NORTHWESTERN UNIVERSITY

Revisiting Software-based Memory Management

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Brian Suchy

EVANSTON, ILLINOIS

September 2022

ABSTRACT

Revisiting Software-based Memory Management

Brian Suchy

Memory management and address translation need significant optimizations in order to not be hindrances in the near future. Currently, plenty of work has started to address issues within the current abstraction of the hardware-software codesign of paging. I argue that a new abstraction is needed in order to properly address this growing issue. I conducted an investigation into reviving software-based abstractions to manage memory with through creating a Tool to Examine and X'form Allocation State, or TEXAS. TEXAS is a software-software codesign to manage memory using the compiler and runtime.

There are several projects that explore TEXAS's capabilities for performing software-based memory management, such as: CARAT CAKE, which uses TEXAS to remove the need for virtual memory along with the supporting hardware like the TLB; CARMOT, which uses TEXAS to provide recommendations to programmers attempting to use complex language extensions like OpenMP; And finally Manufacturing Locality, which uses TEXAS to dynamically convert the temporal access patterns of programs into spatial locality. These projects, and affiliated projects, show the potential of software-based memory management's bright future.

THESIS STATEMENT: *Hardware-software codesign for memory management is becoming difficult and will continue to become worse as data size and complexity increases. With modern advances in compilers paired with the current computing environment, alternative approaches can now be a replacement for a model that is likely doomed to fail in the future. A software-software codesign is a viable alternative for managing memory on large-scale computers using arbitrary languages. In addition to being an alternative, it will enable new optimizations and abstractions that will change how we think about memory management.*

Acknowledgements

I would like to express my deepest gratitude to my advisor Peter Dinda. His immense help, knowledge, and guidance throughout the entire graduate school experience made the PhD process magnificent.

I wish to thank Simone Campanoni and Nikos Hardavellas for their immense contributions to my graduate experience in both time and knowledge.

I also wish to thank Abhishek Bhattacharjee and the rest of my committee for their time and help with bringing this dissertation across the finish line and for offering their guidance.

I would like to thank my co-advisees and colleagues Conor Hetland, Mike Wilkins, Chris Kraemer, Nick Wanninger, and Griffin Dube, Enrico Deiana, Ettore Trainiti, Vijay Kandiah, Tommy McMichen, Brian Homerding, Yian Su, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Brian Tauro, and Kyle Hale for the happy times and good vibes.

Finally, I would like to thank my wife Maria for all the love and support she provided throughout the entire graduate school experience.

THESIS COMMITTEE

Peter Dinda Northwestern University	Committee Chair
Nikos Hardavellas Northwestern University	Committee Member
Simone Campanoni Northwestern University	Committee Member
Abhishek Bhattacharjee Yale University	External Committee Member

Table of Contents

List of Figures

CHAPTER 1

Introduction

Memory management, the foundational system for providing all memory services on a computer, is crucial to the performance and efficiency of computer systems and, by extension, all applications. From allocation to address translation, the memory management system extends into every facet of modern computing and is a crucial component from embedded systems up through exascale computers. As workloads and machines change and scale, the bottlenecks of modern memory management systems, such as hardware-software paging co-design, are becoming increasingly apparent and prevalent. Soon, these issues will threaten to limit performance and efficiency across a wide range of computing. Consequently, research on extending the current memory management model to address these issues has become prominent and continues to expand. It is important to understand that the current memory management model dates back to the 1960s, a time when memory was faster than the processor and the size of it was counted in words.

As expected, the landscape of computing has changed dramatically. Memory is now much slower than the processor and size is counted in terabytes, the system software is now *much* more intimately involved in the computing stack, power and area are now significant factors in microarchitectural design, and hardware now needs to be more flexible[1]. With the changing landscape, it is high time to rethink the abstractions that are the foundations of current memory management systems.

[1]As seen with the rise of FPGAs and ASICs.

I believe that the community should address the problems of current hardware-software co-designs by instead focusing on software-software co-designs, reviving software-based memory management. This renewed approach can remove the need for hardware structures like the translation lookaside buffer (TLB) or pagewalkers, and furthermore allows us to think about managing memory in ways not bound by the traditional model. This idea of managing memory through software is not a new idea. Software-based memory management has had success in the past, most recently explored in the Microsoft Singularity project [**85, 86**]. However, earlier work encountered obstacles such as a lack of protection, being restricted to subsets of managed languages, or simply being too slow compared to the traditional model. The difference today is that modern advances in compiler and systems software research paired with other advancements make software-based memory management feasible. Modern software-based memory management allows us to obtain the benefits of software-based memory management, and can even expose new optimizations. This is done while negating and efficiently addressing many of the obstacles that previous attempts encountered.

If software-based memory management was chosen as the primary form of memory management, there are various immediate benefits available, including (but not limited to):

- Removing or deactivating address translation hardware
- Allowing larger and faster L1 caches
- Enabling memory to be managed with arbitrary granularity instead of only paging
- Dynamic manufacturing of memory locality
- Analysis and understanding of program semantics

1.1. Address Translation: An example of software's potential

An example of the potential for software-based memory management can be seen by looking at address translation hardware. With a modern software-based memory management model, the TLB and paging hardware would no longer be needed. The current co-design between the hardware and kernel that implements paging is becoming unsustainable. Growing TLB capacity is extremely challenging, resulting in level-1 TLBs that have stayed quite small [2] and level-2 TLBs growing only glacially [3]. The growth in reach has also been modest, culminating in perhaps 3 GB in today's TLBs.

While TLB reach remains uncomfortably low, data footprints continue to grow exponentially [**12, 42, 81, 151**], and workloads often suffer TLB misses and long-latency page walks to satisfy data and instruction fetch requests [**30, 31, 55, 72, 82, 96, 135, 153, 174**]. Some workloads have $>$100 misses per thousand instructions. Despite the STLB, these misses still translate into about two page walks per thousand instructions on average and up to as many as 16 for some workloads. Given that page walks consume 54 to 790 cycles, it is easy to see why the cost of address translation continues to be a vexing problem. Previous studies largely corroborate the high cost of TLB misses, and the need to mitigate it has fueled much contemporary research [**30, 31, 32, 55, 72, 82, 96, 135, 153, 174, 25, 15**].

The combined pressure to keep delay, power, and area at bay ultimately hamper the growth of traditional address translation. The hardware structures supporting this model (per-core DTLBs, ITLBs, STLBs, nested TLBs, quad pagewalkers, walker caches) together already require almost as much area as L1 caches [**29**]. A TLB consumes a significant amount of

[2] 64 DTLB entries in modern Intel processors [**88**]

[3] From 1024 STLB entries on 2013's Haswell to 1536 on today's generation

power [**55, 96, 135, 174**], and is a prominent thermal hotspot [**139**]. Early studies find that TLB power consumption is as high as 15-17% of the chip power [**91, 93**]. Subsequent works corroborate these findings, with industry reporting that TLBs consume up to 13% of a core's power [**66, 156**], and later studies estimating that TLBs are responsible for 20-38% of the energy consumed by an L1 cache [**26, 64**]. Increasingly distributed TLBs are no panacea either, as there are significant overheads associated with keeping them coherent [**18, 23, 29, 133, 166, 175**].

Additionally, we have only continued to add more and more complexity to this already complex system through accelerators like GPUs, IOMMUs, and new projects like Apple's M1 processor [**177**] which introduces new issues and complexities.

By removing this hardware, the overall system would have significant savings in power of around $\sim$15% and area roughly the size of an L1 Cache. This would also result in less hardware testing, verification, and bug fixing. Finally, it would eliminate one of the unpredictable aspects of memory accesses, a TLB miss.

As a consequence of decoupling virtual addressing and L1 cache design, the L1 cache size could be significantly increased. In modern systems, the L1 cache is virtually indexed and physically tagged. This introduces issues like synonyms for the same physical page or the explosion of associativity required to increase cache sizes. Currently, x64 typically uses 64 KB, 16-way, caches. Expanding this to 256 KB would currently require **64-way** associativity, which is not feasible due to timing and energy constraints. In contrast, software-based memory management with purely physical addressing would allow for these larger L1 sizes with the same or lower associativity while matching, or surpassing, current latency and energy requirements.

Software-based memory management also lets us rethink the granularity of memory management. Instead of being limited to page granularity[4], the kernel could manage memory by allocations (mallocs), regions (mmaps), or any other scheme. This would allow the programmer or systems developer much more flexibility in the execution and design of a program. Programs would now be free to execute with a granularity more natural to how a program runs.

1.2. Revisiting Software-based Memory Management

Software-based memory management is not a new idea [**137, 19, 43, 108, 140, 85**], with the most recent significant effort being Microsoft's Singularity project [**86**].

The attempts at software-based memory management in the past met various obstacles in their implementation, which led to the projects not taking significant market share in the research or production world. The key issues with the projects of the past were due to protection issues, language restrictions, and most importantly overhead compared to the state-of-the-art at the time. Singularity specifically offered insight into the common issues facing software-based memory management today.

The Singularity project deprecated two priorities important to most successful operating systems: high performance and compatibility with previous systems.[**86**]. These deprecations, although leading to the discoveries the project made, ultimately removed it as a viable alternative to replacing the current memory management system. Because of the decision to instead focus on design and research, Singularity did make a few key insights about the problems facing software-based memory management today. The issues that were highlighted were: run time overhead, the disconnection between different layers of the computing stack, and the need to

[4]Pages are limited to 4 KB, 2 MB, 1 GB sizes on x64

provide backwards compatibility. Each of these issues for Singularity were resolved by ignoring them[5]), creating a new software stack, and using the Sing# programming language.

Singularity, and other software-based memory management projects, showed that to make a feasible software-baased memory management system would require the system to address the problems of protection, language/computing stack restrictions, and overhead. Since it has been 15 years since Singularity attempted software-based memory management, has anything changed that would allow software-based memory management to be feasible?

1.3. Making Software-based Memory Management Feasible with TEXAS

To begin investigating the feasibility of software-based memory management, I developed the Tool to Examine and X'form Allocation State, or TEXAS. TEXAS is a software-based memory management tool that takes advantage of modern compiler advancements to address the problems of protection, language, and overhead problems that plagued earlier approaches to software-based memory management. Using TEXAS, I enabled new abstractions as well as developed[6] new tools that can further help current and future programs.

TEXAS, and my thesis, fits in a broader context of research I have participated in that rethinks interactions between hardware and software interfaces as well as memory management abstractions. This dissertation begins by discussing CARAT, a work that attempts to create software only memory management in place of paging, which guided TEXAS's creation. This is followed by new research thrusts that are being explored using TEXAS as a foundational tool. These projects involve: CARAT CAKE, where virtual memory and paging is completely replaced by a software only memory management system in place of paging; CARMOT, where

[5]Overhead and backwards compatibility were not focused on for Singularity
[6]As well as currently developing.

dynamic observation of programs memory use can be used to automatically generate modern coding abstractions like OpenMP pragmas; and Manufacturing Locality, where access patterns of a program are dynamically converted into spatial locality. Additionally, I briefly discuss other projects not spearheaded by (but still involving) myself that build upon TEXAS concepts, such as: TrackFM, which provides a transparent, and user level, far memory management system; and WARDen, which allows dynamic disabling of cache coherence.

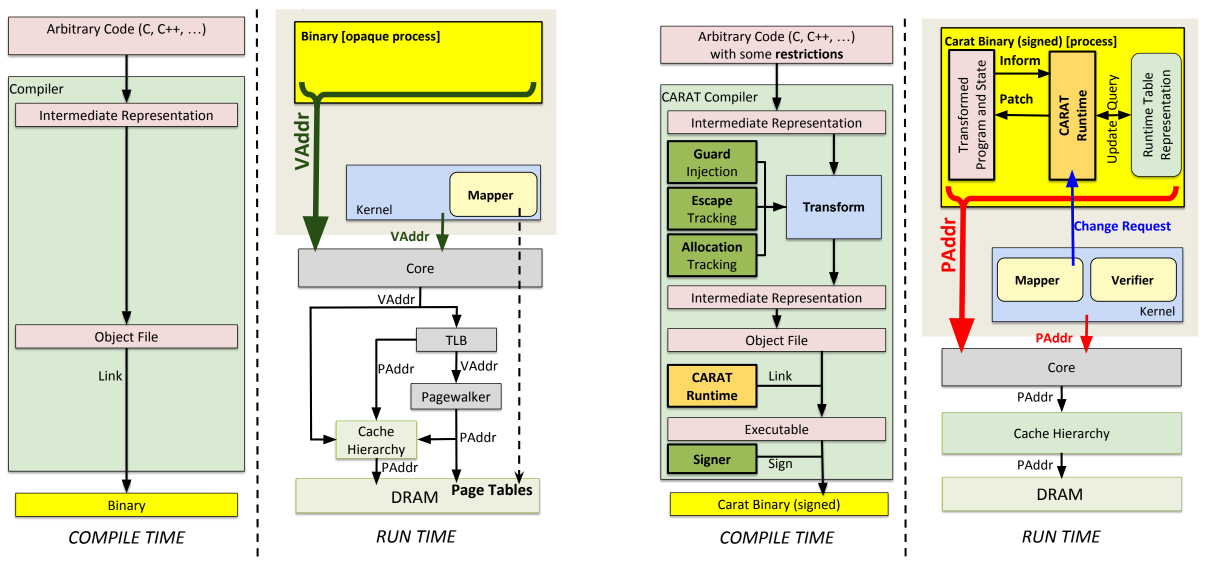
THESIS STATEMENT

Hardware and software co-design for memory management is becoming difficult and will continue to become worse as data size and complexity increases. With modern advances in compilers combined with the current computing environment, alternative approaches can now be explored as a replacement for a model that is likely doomed to fail in the future. A software-software co-design is a viable alternative for managing memory on large-scale computers using arbitrary languages. In addition to being an alternative, it enables new optimizations and abstractions the will change how we think about memory management services.

CHAPTER 2

Investigating Software-based Memory Management with CARAT

The first attempt at addressing the current problems with memory management was Compiler-And Runtime-based Address Translation, or CARAT. CARAT is a re-imagining of virtual memory to be purely software based and eschews the hardware support of TLBs, page walkers, etc. CARAT accomplishes this by making the trade off of adding complexity to the compilation and runtime aspects of a program in exchange for dramatically simplified hardware needs. CARAT can be applied to arbitrary, unmanaged languages. The CARAT prototype is implemented as



(a) Traditional (b) CARAT

Figure 2.1. Comparison of the traditional address translation (paging) model with the CARAT model.

a set of compiler transforms and optimizations in the LLVM toolchain that produces an executable with an embedded runtime that can then run on top of Linux augmented with a kernel module. The prototype seeks to provide the same functionality provided by paging while also being less restrictive in its implementation. Figure 2.1 compares the traditional model and the CARAT model.

2.1. Table of Terminology

The following table contains terminology used to describe the CARAT model and will hopefully clarify any confusion that may be encountered.

Term	**Meaning**
CARAT	The model describing the compiler pass(es) and runtime that replaces paging.
Allocation	Any memory allocation that a program makes. This includes heap allocations (mallocs, global variables, ect), the stack, or anything the program used memory for.
Escape	Any reference to an allocation stored outside of the initial allocation pointer.
Contained Escape	An Escape that is stored within an Allocation.
Guard	A guard surrounding a memory access that ensures proper access permissions.
Memory Region	A contiguous block of memory addresses.
Worst-case memory	The segment of memory in a running program that will require the most amount of time to move.
Allocation tracking	Injected code that will keep track of every allocation of a program.
Escape Tracking	Injected code that will keep track of every escape of a program.
Guard Injection	A compiler pass that inserts guards around memory uses in a program.

Figure 2.2. Terminology of CARAT

2.2. Traditional Model

In the traditional paging address translation model (Figure 2.1(a)), the compilation and linking process is simplified because it can target an abstract virtual address space that is independent of the actual machine's physical address space or how the kernel is currently using it. From the kernel's perspective, the process is mostly opaque, and the kernel's responsibility is to create the illusion of the abstract virtual address space. It does this jointly with the hardware by interposing on each and every memory reference.

Every memory reference, including an instruction fetch, has its virtual address (*VAddr*) translated to a physical address (*PAddr*) which is ultimately the address the memory system uses. On x64 and other processors, the VAddr is also used to immediately begin the cache line lookup in the highest level(s) of the cache hierarchy. Typically, the L1 cache is virtually indexed, but physically tagged.

Address translation happens at page granularity, so instead of translating VAddr$\rightarrow$PAddr, only the bits of the VAddr that contain the virtual page number (*VPN*) are translated to (and replaced by) bits that contain the physical page number (*PPN*): VPN$\rightarrow$PPN. Technically, the translation is VPN$\rightarrow$PTE, where the *PTE* (page table entry) contains both the PPN, access permissions and other metadata about the VPN and PPN.

VPN$\rightarrow$PPN is a mapping from occupied virtual pages in the current virtual address space to the occupied physical pages in the physical address space. This mapping is determined by the kernel and it may change over time. Current systems represent mappings as radix trees. For x64 in particular, this arrangement is a 4-level hierarchy with each level corresponding to 9 bits of the virtual address. The hierarchy allows short-circuiting during traversal to create composite pages (a page can consist of a single base page (4KB in size), 512 base pages (a "large page"), 512^2 base pages (a "huge page"), and (eventually) 512^3 base pages. Intel and AMD do not currently use the entire 64 bit virtual address space, resulting in an apparent mismatch ($9+9+9+9+12 \neq 64$), but there is a model for expanding it that involves adding more levels over time.

The TLB caches translations that have been read from the in-memory page tables, and its extremely high hit rate is essential to providing modern address translation with little performance overhead compared to using physical addressing. When the TLB misses, the pagewalker traverses the in-memory page tables using physical addresses until it finds the relevant PTE and places the mapping into the TLB.

TLBs require substantial chip area and power in order to achieve their necessary extremely low latency. Furthermore, a range of important workloads have been found to have high TLB miss rates, which has spawned a range of research on alternative address translation approaches. While pagewalkers have more relaxed latency requirements, their speed is critical for properly handling such workloads that do not have good enough spatial and temporal locality and thus have high TLB miss rates. In particular, our work is driven by the address translation demands of parallel and high-performance applications (which display these characteristics).

2.3. CARAT Model

In the CARAT model (Figure 2.1(b)), the hardware can be considerably simplified, but the compilation and runtime environment are considerably more complicated. Because only physical addresses are used, the TLB and the pagewalker can be eliminated. In a system that provides both the traditional and CARAT models, the kernel could switch between them with simple hardware support. For example, on x64, physical addressing could be reintroduced, allowing a kernel write to set CR0.PG to zero to disable paging, a capability already present when an x64 processor is run in 32-bit mode.

2.3.1. Compile-time

The compilation process in CARAT involves three additional steps compared to the traditional model. The first, multi-part, step is a set of compiler transformations that serve as the basis for making the executing process both safe and malleable. “Allocation tracking” introduces instrumentation code that invokes the runtime whenever there is a memory allocation. An “allocation” is a broad term in CARAT, and includes both static allocations (e.g., global variables), and dynamic allocations (e.g., mallocs, stack allocs, etc). “Escape tracking” is similar, and introduces instrumentation code that invokes the runtime whenever a pointer is copied (an “escape”) or destroyed. Allocations and escapes from the initial state of the program’s globals are captured at load time.

Conceptually, “guard injection” introduces a guard to every load, store, and call instruction. A “guard” verifies that the physical address about to be used by the instruction is within the restricted set allowed by the kernel and that the appropriate access permissions hold. The kernel essentially provides a dynamic set of “address regions” and their privileges to the CARAT runtime, and a guard checks the address range of the prospective access against this set.

If each relevant instruction truly were guarded, the overhead of CARAT would be abysmal, but CARAT specific optimizations using modern compiler techniques mitigate a significant chunk of this overhead. The CARAT model heavily relies on compiler optimization technology, including new CARAT-specific optimizations, to eliminate, combine, or amortize guards in many situations. An important result is that these transformations are possible for a wide range of programs across many computing areas.

Guarding call instructions is necessary because the call's push of the return address onto the stack could overrun a valid region. Additionally, the prologue and epilogue code the compiler produces for the callee may also perform stack accesses. A call guard verifies that all such 'hidden' stack accesses will be within a legal region. A failed guard involving the stack causes the kernel to be invoked. This provides a mechanism by which the kernel can implement seamless stack expansion, if desired. Beyond this mechanism, a compiler could also ensure that the control flow of calls and returns happens on a separate control flow stack, thus making a stack-oriented attacks, and even return-oriented programming (ROP) attacks, impossible.

The second compilation step involves linking a program with the CARAT runtime that serves as the backend for the guards, the escape tracking, and the allocation tracking code. The runtime is also the interface with the kernel. A key idea is that the runtime can patch every pointer in the program that is affected by a change in mapping the kernel wants to make.

The final compilation step involves signing the resulting binary with the credentials of the compiler toolchain, so that it is easy to validate that a specific compiler made the binary. Alternatively, an attestation model could be used, making it easy to validate that a specific software/hardware stack made the binary. A kernel can then determine whether to trust the binary based on the provenance of the compiler or of the software/hardware stack producing it.

2.3.2. Run-time

CARAT processes, and the kernel, run within a single physical address space and use purely physical addressing. This is a marked difference from the traditional model. At program load time, the kernel first validates the signature on the binary, and then decides whether to trust the compiler or software/hardware stack that built it. It then selects an appropriately-sized region of memory for the code, and initial regions for the program's globals (e.g., data and bss) and stack. Next, it copies the program code (including the CARAT runtime, guards, and allocation/escape tracking) and initialized data into the relevant regions, and then it initializes bss and the stack. It then writes the currently allowed regions into space set aside in the runtime for this purpose. Finally, it invokes the runtime for the first time. This initial *change request* causes the runtime to perform a patch of all global pointers to reflect their initial targets given the layout the kernel has decided on. The kernel then invokes the entry point of the program to get the process running.

During normal execution, allocations and escapes that appear will inform the CARAT runtime which then uses them to update its tracking data structures. Guards are invoked as needed. If a guard fails to validate an address against the kernel-supplied memory regions, it causes the kernel to be invoked, similar to a page fault in the traditional model. Thread creations or other sources of additional stacks are handled readily since these added stacks are allocated in heap memory.

When the kernel decides to change the available regions of physical memory or their permissions, it performs an upcall that forces all running threads to dump their register state onto their stacks. The CARAT runtime then performs a barrier and notifies the kernel that it is safe to change the regions. The kernel then modifies the region set and notifies the runtime. The runtime then resumes every thread. The very next guard will see the changes.

A more significant operation is when the kernel decides to move a range of data pages. As with a region change, the kernel first forces all threads into the CARAT runtime, where they perform a barrier. Note that forcing each thread out also dumps is current register state onto its stack. This makes in-register pointers visible for patching, similar to what is necessary in garbage collection. The kernel next informs the runtime of the page range migration it intends. The runtime uses the source range as a query on its data structures to find all allocations that overlap with the range. If an allocation only partially overlaps the page, the runtime coordinates with the kernel until the kernel either selects a range that does not have an overlapping allocation, or expands the initially selected one to meet the same requirement.

After the source range has been determined, the CARAT runtime queries its data structures (and the register snapshot on the stack) to find all escapes of all allocations in the source range. It then patches each escape with the address the pointer will have once the data in the source range has been moved into the destination range. That is, the CARAT runtime swizzles all pointers affected by the proposed data movement to point to where the data they point to will land once the movement is complete. Next, it informs the kernel that it has finished this task. The kernel then performs the data movement, and resumes all threads of the process.

Note that migrations of pages to/from swap, as well as demand paging from a file, can also be accomplished in this framework. To make a page unavailable, we patch its affected pointers to a physical address that will cause a fault. In x64 systems, one option is to use a non-canonical address. Since the range of non-canonical addresses is vast, the specific non-canonical address can be used to encode different conditions (e.g., swapped, demand-page, “null pointer”, etc).

By using these two kinds of change requests, the kernel can accomplish the main goals of address translation, namely protection and migration of physical pages, without using virtual addresses or the hardware needed to support them.

2.4. CARAT Model Prototype

I have created a prototype of CARAT that implements the CARAT model as described above with some changes that differentiate it from a complete implementation of the model. One thing to note is that the differences from the model only add overhead to the prototype.

2.4.1. Compile-time

Instruction Fetches. How can CARAT guard instruction fetches, which are also memory references? Instead of guarding them, we place minimal restrictions on the code that we can compile which guarantees that the compiler can prove that all control flow is local to the code it produces (including libraries). All control flow is then implemented using PC-relative means (forcing position independence for all code, but not data). The executable is then statically linked, resulting in the code being mobile—the kernel can stop and move the code at any time. This process results in programs being compiled into a dark shadow capsule [**61**].

Self-contained Control Flow. The current prototype restrictions needed for the compiler to be able to assure that all control flow is self-contained (to adhere to the dark shadow capsule model) are the following: (1) No undefined behavior is allowed. When detected, compilation fails. (2) No self-modifying code is permitted. If casts from/to function pointers to/from data pointers are detected, or pointer arithmetic on function pointers is detected, compilation fails. (3) No inline assembly or separate assembly is allowed. These restrictions apply only to user programs.

User-level vs Kernel-level. The biggest difference between the model and the prototype has to do with the kernel components being implemented in user space instead of within the kernel. Examples of this include the use of signals instead of interrupts, patching is done by the user

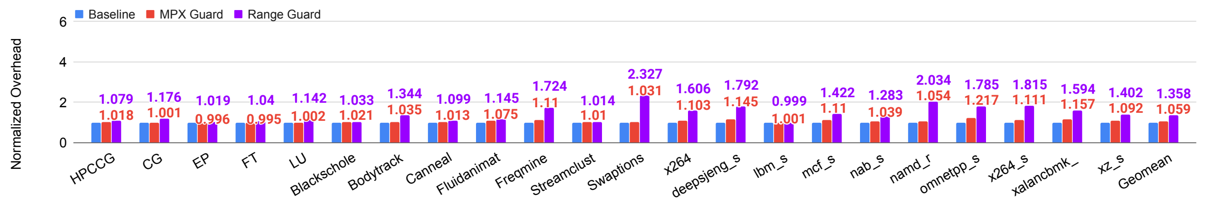


Figure 2.3. Overhead of Protection for CARAT Prototype

program instead of the kernel, what/when to move memory is determined by an adjustable timer, and that the memory moved is always the worst case scenario for the program. Additionally, when performing page movements, the user program will make the new allocation for the moved memory instead of being provided the address of the memory.

Memory Guards. The memory guards used during the run time of the program are also slightly modified compared to the model. There are two ways to implement the guards within the CARAT prototype. The first, which corresponds to the model accurately, places a double branch of LLVM bitcode logic around each memory access in the program. This can be thought of as surrounding each memory access with two 'if' statements, the first making sure the memory is within an upper bound and the second checking the lower bound. The other implementation uses a proxy for the Intel MPX instructions. The Intel MPX instruction performs the same upper and lower bound check, but within a single cycle. Because we do not have access to this instruction within the prototype a proxy instruction that performs a 'Store' instruction is used instead.

2.5. Feasibility

My CARAT case paper [**161**] goes in depth on the various feasibility tests that my prototype of CARAT passes, as well as describing CARAT in more detail. The key results are shown in

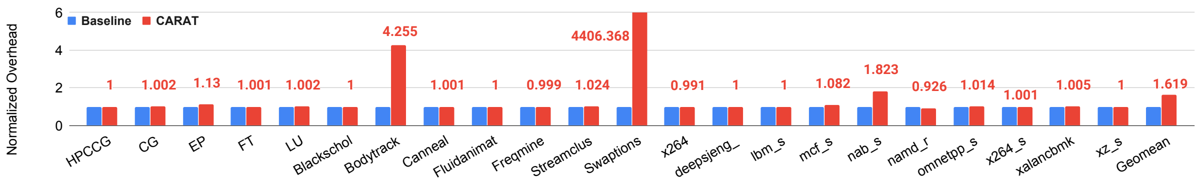


Figure 2.4. Memory Overheads for CARAT Prototype

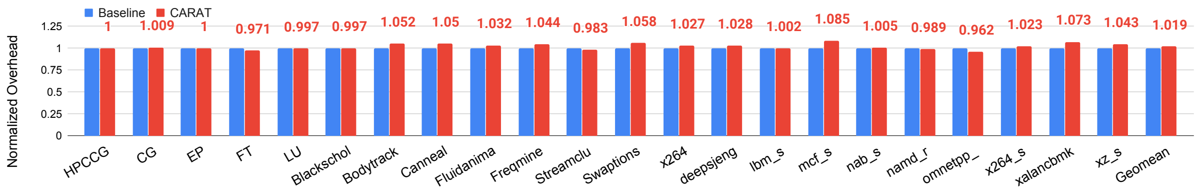


Figure 2.5. Time Overhead for CARAT Prototype

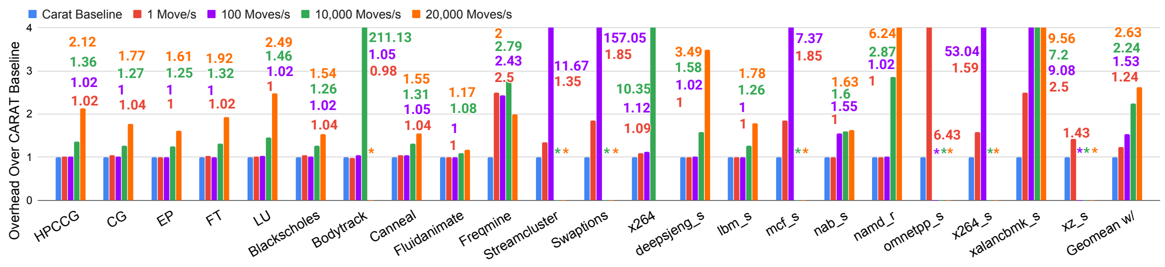


Figure 2.6. Time Overhead to Move Memory for CARAT Prototype

Figures 2.3, 2.4, 2.5, and 2.6. Figure 2.3 describes the time overhead of guarding all memory accesses for each benchmark. Using compiler optimizations, I am able to make the overhead of guarding all memory accesses only 5.9%. Figure 2.4 shows that the extra memory footprint to track all the allocations and escapes of a program. When taking the geomean of this extra memory overhead for all programs, the overhead is only 61.9%. But when looking at the graph, one can see that there is an outlier, Parsec's Swaptions. If we discount Swaption's (by limiting its overhead to be 5x), the overhead drops to 19%.

Perhaps the most important figures (at least for this thesis proposal) are Figures 2.5 and 2.6. Figure 2.5 shows the time overhead of tracking a programs allocations as well as the escaping pointers. The overhead to keep track of this in a program is only 1.9%. Figure 2.6 shows the overhead of moving memory around at constant rates using CARAT. These movements are performed using the worst-case piece of memory of each program. When moving memory at about 100x the typical movement rate of programs (1 Move/s), the overhead incurred is only 24%. The movement rate of 20,000 Moves/s is shown as the *extreme* upper bound on movement rates. None of the benchmarks even *allocate* memory at this rate, but it is included to show that CARAT can still function even under rates that are above any potential situation during run time.

What is even more impressive is that the prototype's overhead of moving memory is actually at least 20x *higher* than the true overhead. This is because of two factors. The first is due to moving the worst-case piece of memory. In order to find the worst case piece of memory, the prototype has to look through all of its allocations and discover the most detrimental piece of memory to move. The second reason for the 20x overhead is due to the prototype being built on top of paging. Once the worst-case piece of memory is selected to be moved, the prototype needs to spend time to find every page of memory that will need to be moved as well. For example, if the piece of memory being moved is on a page that another allocation of the program happens to be on, then we must also recursively perform a movement for that allocation's memory as well.

If the two sources of overhead were removed (the first by allowing any memory to be moved and the second by letting the prototype operate with allocation granularity instead of page granularity), then the overhead of movement at 100x the typical movement rate (1 Move/s) would shrink to be at most 1.2%.

The main idea of the results produced by the CARAT prototype is that it is definitely feasible to manage memory of programs completely in software with acceptable overheads.

CHAPTER 3

TEXAS: A Toolset for Software-Software Co-designed Memory Management

The CARAT project demonstrated that the concept of managing memory in software is, at the bare minimum, a feasible and viable approach. The question to ask now is: What else can be done if memory is managed in software? To answer this, we need a general purpose toolset that takes CARAT runtime and compiler passes, modularizes it, and extracts its core components to make a more general tool.

3.1. Tool for Examining and X'Forming Allocation State (TEXAS)

The Tool for Examining and X'Forming Allocation State, or TEXAS, is the base tool that will be expanded upon to explore new ideas in the scope of software-based memory management. The TEXAS tool, at its core, is taking the CARAT project, breaking it down into its core components to create a modular toolset that can be adapted for other investigations into software-based memory management.

3.1.1. Runtime Components

The first components that make up TEXAS are the runtime abstractions. These abstractions are taken from CARAT as well as being newly created to provide a more holistic view of memory for unmanaged languages.

Term	**Meaning**
Allocation	Metadata for any memory allocation that a program makes. This includes heap allocations, stack allocations, globals, pages, or anything the program uses memory for.
Escape	Any reference to an Allocation stored outside of the initial Allocation pointer.
Allocation Table	A data structure that keeps track of Allocations
Guard	A guard surrounding a memory access that ensures proper access permissions.
Use	An access of an Allocation's memory. (Read/Write)
Connection Map	A graph of Allocations that forms edges based on user defined metrics.
Repacker	The mechanism that TEXAS uses to move Allocations in memory.
SuperMalloc	A large heap allocation that TEXAS assumes memory management over. Allocations within a SuperMalloc also have their memory managed by TEXAS.
State	A user-defined region of code that will collect metadata during dynamic execution of the region.
Allocation tracking	Injected code that will keep track of every Allocation of a program.
Escape Tracking	Injected code that will keep track of every escaping reference to an Allocation.
Guard Injection	A compiler pass that inserts Guards around memory accesses in a program.
Use Tracking	A compiler pass that inserts tracking for all Uses in a program.
TEXAS Loop Creation	A compiler pass that clones loop bodies, and then instruments the cloned body.

Figure 3.1. Terminology for TEXAS

Allocation The first component taken from CARAT is the concept of an Allocation. In CARAT, and later in CARAT CAKE, an Allocation is used as metadata that contains information needed to perform memory movement and protection on active Allocations. To generalize this, Allocations now serve as metadata for arbitrarily sized pieces of memory. These can still be Allocations in the sense of CARAT's work, but the concept is now expanded to fit arbitrary uses.

On top of containing CARAT information such as Escapes, affiliated pointers, and length, TEXAS Allocations can contain other pieces of metadata including, but not limited to, an allocations *vertex id* in the Connection Map, its memory alignment, and its parent SuperMalloc.

For the following chapters, Allocations are used in various ways. For example, in TrackFM (§ 7.1) Allocations are treated as arbitrarily uniformly-sized pages that can contain multiple (or subsets of) traditional "Allocations" within them.

Allocation Table/Map The next component taken from CARAT is the Allocation Table, or Allocation Map. In CARAT, this global data structure contained information about *active* Allocations. For TEXAS, this data structure is still used to map memory addresses to Allocations, but also has extended functionality and metadata including (but not limited to): getting the memory footprint of a program, estimating memory fragmentation, and invoking the generation of a Connection Map. Additionally, the Allocation Table is no longer required to be a single global table and can instead be an arbitrary number of tables that operate on a per-thread basis or even a per-State (§ 3.1.1) basis. Also, the Allocations that exist in an Allocation Table no longer need to be active.

Guards and Use Tracking The final runtime component taken from CARAT is the Guard. In CARAT/CARAT CAKE a Guard is invoked on memory accesses to ensure proper read/write permissions. In TEXAS, a Guard can be generalized to be more flexible. "Use Tracking", and the term Use, in TEXAS is defined in a similar manner to the terminology used in LLVM. A Use in LLVM is defined as an access of a Value by another instruction[1]. Consequently, in TEXAS a Use is defined as an access of an Allocation by an instruction. A Guard can be seen as a variant

[1]A Use in LLVM is defined as an edge connecting a Value to a User.

of a Use, in which the goal of Guard Injection is to only inject a Guard for the temporally first Use of an Allocation.

Uses allow more general tasks than guards. For example, in CARMOT (§ 5), the Uses will be used in the creation of Computational Spoors. In TrackFM (§ 7.1), the Uses will be used in a similar fashion to CARAT Guards, but for ensuring an Allocation is in local memory before it is accessed.

Connection Map The first new runtime component is the Connection Map. This data structure is implemented as an Adjacency List from the Boost [**149**] library. The Connection Map is used to form a graph that connects Allocations to one another. The edges that this graph forms can be determined in various ways. TEXAS currently has three ways of forming edges.

The Allocation-to-Allocation Connectivity Mapping, or AACM, method uses the Escapes and Allocation Map to generate edges of Allocations that point to other Allocations. In effect, this Connection Map dynamically reconstructs data structures in a running program. For example, a linked-list or tree data structure can be reconstructed by TEXAS allowing TEXAS to make observations proceeded by transformations.

The Access Pattern Method, or APM, uses Use Tracking to form temporal edges between Allocations. By tracking the dynamic use of memory, TEXAS can also form edges based on how close in time one Allocation was used with respect to another.

The Spatial Method, or SM, uses an Allocation Table to form spatial edges between Allocations. Using the Allocation Table, TEXAS can form the edges between Allocations that are within a certain range of addresses from one another *or* may share other spatial characteristics with one another such as potential false sharing.

Repacker The Repacker is the component responsible for moving/patching Allocations in memory. When performing a move of an Allocation(s), the Repacker is responsible for patching all Escapes of the affiliated Allocation(s) as well as acquiring ownership of the memory management of the Allocations through the creation of a SuperMalloc (§ 3.1.1).

In its current state, the Repacker can take in an Allocation Table *or* a Connection Map and repack the affiliated Allocations using heuristics to X'form the given Allocation Table/Connection Map. The current repacking heuristics in TEXAS include: defragmentation, in which all the affiliated Allocations are repacked in memory to reduce external fragmentation; BFS/DFS repack, in which Allocations are repacked using the Connection Map, built with AACM, and selecting the order of nodes based on breadth/depth first navigation of the graph; and Temporal repacking, built with APM, which repacks the Allocations using the Connection Map, based on weighted temporal edges, to order Allocations by temporal closeness[2] of Allocations.

SuperMalloc A SuperMalloc is a (typically) large single allocated segment of memory, of variable length, that can contain many Allocations. TEXAS assumes the memory management of any Allocations that exist within a SuperMalloc. Upon placement of an Allocation into a SuperMalloc, the old memory the Allocation previously existed at will be *free()*'d and the Allocation will be promoted into having the SuperMalloc assume responsibility for it. TEXAS still is compatible with other memory managers such as libc (malloc) or jemalloc. Note, although demotion of an Allocation's management back to one of this memory managers is possible, TEXAS does not currently perform these demotions.

For example, if a user wishes to defragment memory, a single SuperMalloc can be created the size of an Allocation Table's memory footprint. After creation, all the Allocations within an

[2]Closeness is a TEXAS user-defined metric

Allocation Table can be moved with a Repacker into the new SuperMalloc perfectly arranged one after another in memory[3]. Other common related terms for a SuperMalloc include memory pools or allocation arenas.

State The final new major component of the TEXAS runtime is the concept of States. A State in TEXAS is a statically-defined “region” of a program’s dynamic execution. More plainly, a State can be thought of as a toggle with a unique identifier that can enable and disable certain aspects of the TEXAS runtime during a program’s execution. Using the unique identifier, the metadata, information, and actions that TEXAS builds/takes can be associated with the identifiers to give subsets of information about a program.

For example, if a user wanted to have an understanding of what a single loop is doing in their program, they could create a State that sets its enable and disable toggles to be wrapping the loop as shown in Figure 3.2.

[3]A SuperMalloc operates as a bump allocator by default.

```
---BEFORE---

int work(){
    int y = 0;
    for (i = 0; i < 10; ++i){
        y += i;
    }
    return y;
}

---AFTER---

int work(){
    int y = 0;
    int StateID = TEXAS_Get_State(); //Begin tracking State
    for (i = 0; i < 10; ++i){
        y += i;
    }
    TEXAS_End_State(StateID); //End tracking State
    return y;
}
```

Figure 3.2. Example of creating a TEXAS State to collect metadata about a loop.

3.1.2. Compiler Components

The next components that make up TEXAS are the compiler passes. These passes are also taken from CARAT as well as newly created to provide users with simple transformations to build upon or use wholesale for their projects.

Allocation and Escape Tracking The Allocation and Escape tracking compiler passes are effectively identical to the CARAT version and add instrumentation for tracking all Allocations made as well as their potential Escape creations. One notable difference is that now Allocation tracking has more toggles that allow control over what types of Allocations to track, such as Stack, mallocs, Globals, heap, SuperMallocs, and more.

Guard Injection and Use Tracking The Guard Injection pass operates effectively identically to CARAT's version. However, there is also an additional new compiler pass that is created from Guard injection for performing Use tracking. Use tracking is an unoptimized Guard Injection that injects calls at every memory use sans hoisting or redundancy removal. This will in effect allow a temporal accurate portrayal of memory uses to be gleaned from a program. For example, Use Tracking can generate a memory reference trace of a program.

TEXAS Loop Creation The TEXAS Loop Creation is a new pass that transforms (designated) loops of a program. The transform creates a clone of the loop body, allowing TEXAS to instrument one of the bodies and letting one run without instrumentation. A visualization of this kind of transformation can be seen in Figure 6.6, in which the loop is converted into an inspector-executor that dynamically repacks memory.

3.2. Using TEXAS to Revisit Software-based Memory Management

Using TEXAS, we can now explore ideas for examining and transforming memory. To explore the potential of TEXAS, there are four distinct paths that all make use of TEXAS to accomplish distinct tasks within the scope of memory management. These projects are visualized in figure 3.3, which shows how the projects connect to one another.

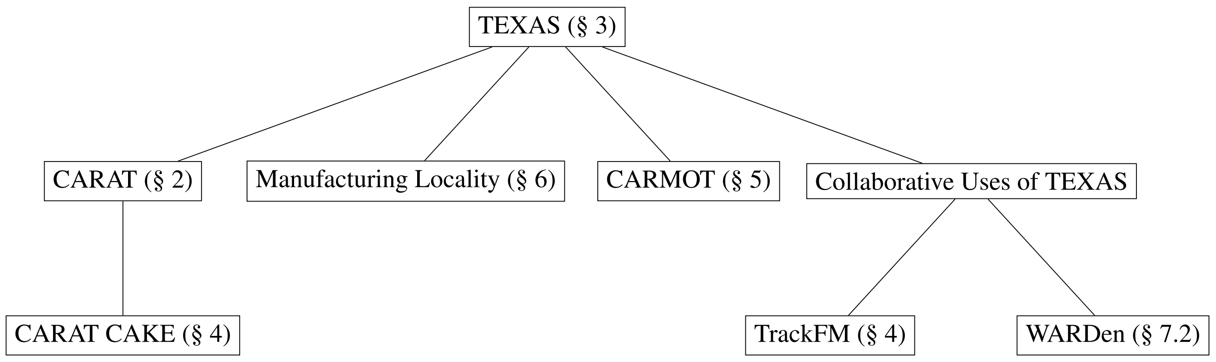


Figure 3.3. Project Connectivity Graph of Thesis Work

3.2.1. CARAT CAKE

The first path is the development[4] of CARAT CAKE. While we have made the case for Compiler-And Runtime-based Address Translation (CARAT), its evaluation was based on a user-level prototype. We now incorporate CARAT into a kernel, forming Compiler- And Runtime-based Address Translation for CollAborative Kernel Environments (CARAT CAKE). In our implementation, a Linux-compatible x64 process abstraction can be based either on CARAT CAKE, or on a sophisticated paging implementation. Implementing CARAT CAKE involves kernel changes and compiler optimizations/transformations that must work on all code in the system, including kernel code.

We evaluate CARAT CAKE in comparison with paging and find that CARAT CAKE is able to achieve the functionality of paging (protection, mapping, and movement properties) with minimal overhead. In turn, CARAT CAKE uses TEXAS in its implementation to enable the

[4]or rather baking

new benefits for systems including energy savings, larger L1 caches, and arbitrary granularity memory management.

3.2.2. CARMOT

CARMOT (§ 5), the second path, uses TEXAS to assist programmers in forming modern day programming abstractions. Modern programming languages offer abstractions that simplify software development and allow hardware to reach its full potential. These abstractions range from the well-established OpenMP framework to newer C++ features like smart pointers. To properly use these abstractions in an existing codebase, programmers must determine how a given source code region interacts with the entire program state (i.e., the program's variables and memory locations). We call this process Computational Spoor Discovery (CSD). Without tool support for CSD, a programmer's only option is to manually study the *entire* codebase.

We propose a profile-based approach that automates CSD and provides abstraction recommendations to programmers. Because a profile-based approach incurs an impractical overhead, we introduce the Compiler and Runtime Memory Observation Tool (CARMOT), compiler and runtime codesign built-on TEXAS. Using CARMOT, we reduce the overhead of CSD by two orders of magnitude, making CSD practical. We show that CARMOT's recommendations achieve the same speedup as hand-tuned OpenMP directives and avoid memory leaks with C++ smart pointers. From this, we argue that CSD tools, such as CARMOT, can provide support for the rich ecosystem of modern programming language abstractions.

3.2.3. Manufacturing Locality

The next path is Manufacturing Locality (§ 6). Programs written in C/C++ require immense care in the placement of memory in order to make best use of cache and memory. Typically, this requires that programmers have a deep understanding of not only the systems architecture, but also how their chosen allocator makes use of the memory system. Unfortunately, many programmers are leaving significant performance gains, as well as wasted memory space due to fragmentation, on the table. Even with an understanding of how a program uses memory, the developer can determine these optimal static placements occasionally. Additionally, static placement of memory may only be optimal occasionally during the dynamic execution of a program.

To address this, we introduce Manufacturing Locality. Manufacturing Locality extends the TEXAS runtime to enable a program to dynamically repack memory in order to best[5] match the access pattern of a program with its memory layout throughout execution. Manufacturing Locality builds a graph-based understanding of memory, and its contained Allocations, in order to perform transformations based on defragmentation, access pattern to layout matching, and BFS/DFS search patterns.

3.2.4. Collaborative Uses of TEXAS

The final path shows how TEXAS is already being used in collaborative works with other researchers. In TrackFM (§ 7.1), TEXAS's ideas are taken to create a far memory management system that operates at user level in units of page-like objects[6], but is completely transparent

[5]heuristically

[6]These objects are a set size, but not limited to 4KB, 2MB, or 1GB.

to the programmer. In WARDen (§ 7.2), TEXAS's extension CARMOT (§ 5) can be used to detect regions of memory that can dynamically have their cache coherence disabled.

CHAPTER 4

CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation

CARAT CAKE, or Compiler- And Runtime-based Address Translation for CollAborative Kernel Environments, is the large step that takes the idea of CARAT and implements it as a full system implementation from the kernel up that executes Linux applications on an x86 processor. More specifically, CARAT CAKE extends an existing single address space OS, Nautilus, with a Linux-compatible process abstraction, underpinned by both CARAT CAKE and paging. Compiler transformations (implemented in LLVM) are applied both to the kernel itself, and to each user program. User programs are then linked in a specialized way, and signed to attest that CARAT CAKE's compiler toolchain produced them. Nautilus can then load signed user programs directly into the physical address space where they coexist with the kernel, running in kernel mode. Protection of the kernel and other processes is achieved by the compiler toolchain introducing protection checks that work cooperatively with the kernel. Memory object migration is possible for both kernel and process objects because the compiler toolchain introduces allocation and pointer escape tracking throughout all codebases. The kernel can always compact/defragment memory, as is needed given the single physical address space.

The publication of CARAT CAKE[**160**] contributed the following:

- We show how to (and do) extend the previously published CARAT concept into CARAT CAKE, a full-blown alternative to paging within a kernel.

- Described the design and implementation of the CARAT CAKE prototype, drawing observations about the challenges of this form of memory management within the kernel.
- Created a Linux-compatible process abstraction on top of CARAT CAKE, drawing observations about the challenges of separate compilation and kernel support. CARAT CAKE also implements the process abstraction using a paging codebase that leverages modern x64 hardware support features for improving paging performance.
- Evaluated CARAT CAKE's overheads in comparison with paging (and with paging on Linux), as well as the software engineering effort involved. The evaluation shows that CARAT CAKE is a viable alternative to paging.
- Described new benefits made possible with CARAT CAKE centered around microarchitecture and software benefits.

The original CARAT user-level prototype is limited because it *emulates* what software-based memory management would look like. Additionally, it is built on top of the Linux process abstraction, paging, and the MMU rather than *driving* memory management *within* the kernel. In contrast, CARAT CAKE lets us put the CARAT concept into the driver's seat and explore the compiler-runtime-kernel design space without constraints. In contrast, CARAT CAKE lets us put the CARAT concept into the driver's seat and explore the compiler-runtime-kernel design space without constraints. CARAT CAKE also forces us to consider how kernel abstractions such as processes are to be implemented on top of CARAT CAKE's mechanisms instead of paging.

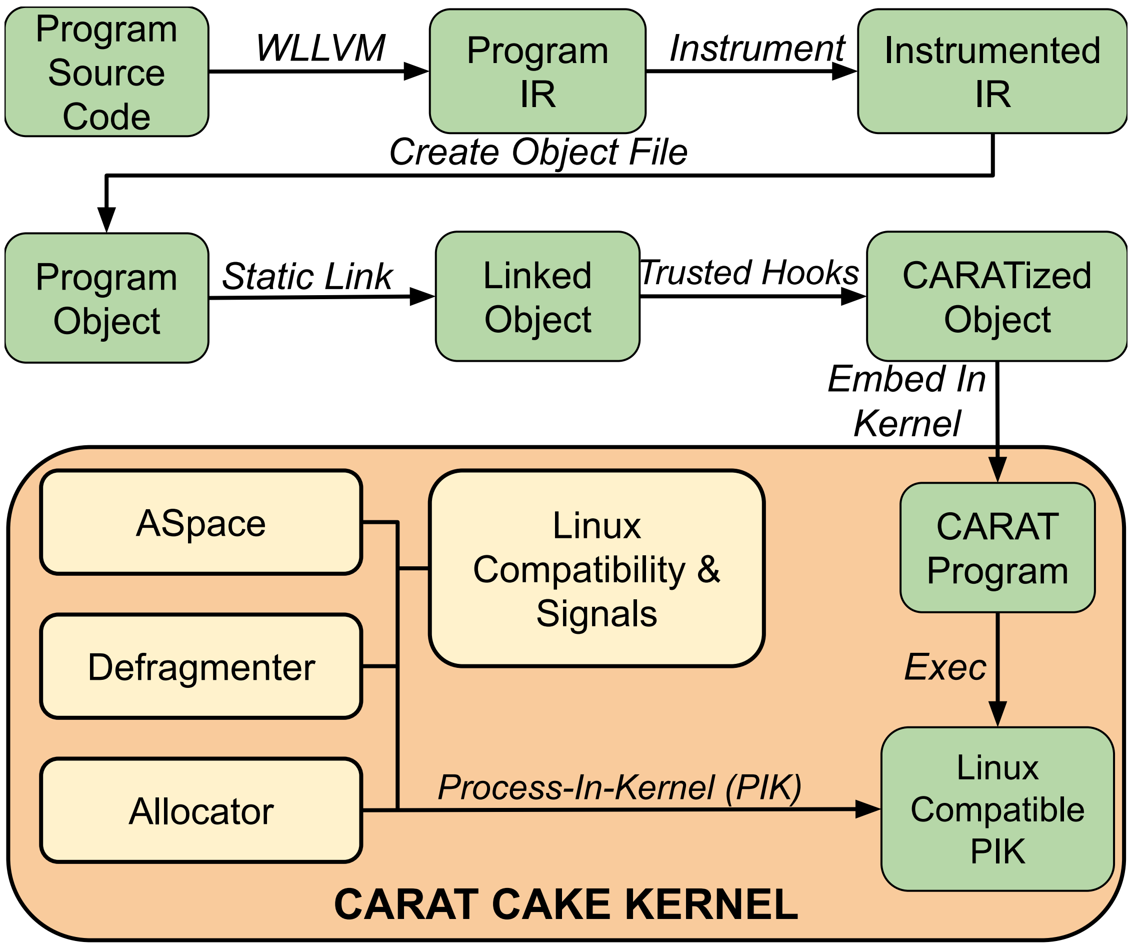


Figure 4.1. CARAT CAKE System Diagram.

4.1. System and Design Choices

Figure 5.5 shows the structure of CARAT CAKE, with the top half comprising the compilation process (for user programs) and the bottom half comprising the kernel elements. The relevant sections of our presentation are noted.

CARAT CAKE is centered around two major components of any virtual memory abstraction: *mapping* and *protection*. The kernel and compiler work in tandem to provide protection and mapping functionality for execution of user programs. The compiler performs analysis and transformation to propel Allocation and Escape Tracking within the kernel and user programs in order to support memory mapping and movement. The compiler also performs Guard Injections to propel kernel protection checks to support protection of the kernel and user processes. The kernel manages a single physical address space in which all code and data coexist.

The compiler, runtime, and kernel are coupled via a set of basic abstractions that allow the kernel to *manage* the physical address space. The kernel allocates Memory Regions and groups them into ASpaces, which the kernel can delegate, expand, or assign to entities requiring memory. In CARAT CAKE, these entities include the kernel itself and *processes*—building the well-known user-space and kernel-space layers. The compiler's instrumentation of the kernel and user programs, combined with the runtime, provides memory tracking and protections *per* ASpace. Memory tracking is also applied to the kernel itself. Memory is managed at the level of Memory Regions. Unlike a paging system, these can be of arbitrary size and thus external fragmentation is a concern. Protections are also managed at the granularity of Memory Regions.

Because the compiler performs protections and tracking via static analysis and transformation of application code, user-level developers can seamlessly develop apps for CARAT CAKE ignorant to the CARAT-based system underneath. The kernel builds stacks, user heap(s), etc, for a process by chunking physical memory directly without address translation. The kernel-level developer can largely ignore CARAT CAKE, unless they are working on memory management itself.

4.2. Compiler

The CARAT CAKE compiler instruments both user *and* kernel code to track Allocations and Escapes and to guard memory references in user code. In contrast to the compiler described in CARAT, our compiler intertwines tracking and protections with kernel-level permissions. We extend Allocation and Escape tracking transforms and apply the Address Checking for Data Custody (AC/DC) data-flow analysis from the prior work, loop invariant analysis, and scalar evolution analysis to elide redundant Guard Injections. Additionally, we generalize the compiler so it can understand *how* to manage or optimize a program's memory management via static analysis and instrumentation at the IR level. This is especially important when optimizing for both user and kernel code, where underlying assumptions about the semantics or safety of memory accesses can differ. We also exploit invariants that we can derive by analyzing the trusted computing base (TCB) for the kernel and integrating them into the CARAT CAKE compiler.

CARAT CAKE's compiler is also now responsible for *actually* enforcing protections for the computing environment; whereas, the CARAT paper simply required emulating the overhead and protection encountered. Because of this, the transformations the compiler enacts are more complex than single bounds checks of the previous work.

One significant change to the compilation flow is how it leverages knowledge of Memory Region semantics in the kernel. If the compiler can guarantee that an instruction in the IR that references memory is safe given the confines of the process, the compiler can optimize away its guard. More specifically, the compiler can elide guards for the following categories of references: 1) explicit stack locations in the IR, 2) global variables, and 3) memory received from a library allocator (e.g. `mallocs`). (1) represents references within the bounds of the stack the kernel itself sets up and hands to the program, while (2) represents a section of the executable that the kernel will load and verify for the program (e.g., `.data`). For (3), the memory ultimately derives from a region (heap/`mmap`) that the kernel allocates and controls

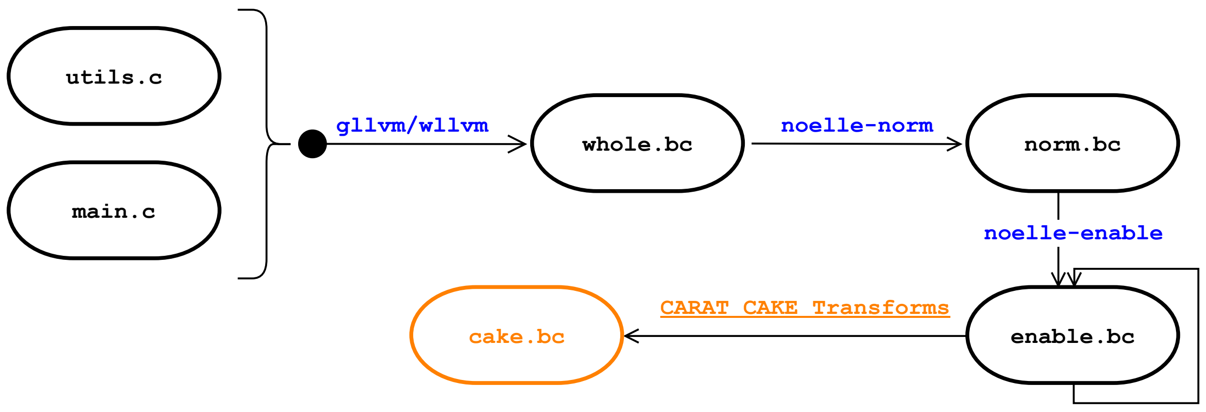


Figure 4.2. Example build procedure of a codebase into a CARATized program using NOELLE.

who it gives access to. To accomplish this efficiently, the compiler passes that inject the guards leverage NOELLE's Program Dependence Graph (PDG) extensively, particularly by enabling dependence analysis, and loop analysis information across the entire program (31 forms of alias analysis at this time).

NOELLE also enables induction variable-based optimizations in place of scalar evolution optimizations. In a loop, or nested loops, NOELLE finds the induction variable(s) and CARAT CAKE can use them to compute the bounds that an IR memory instruction uses, allowing CARAT CAKE to enforce protection using these bounds. When the induction variable analysis provided by NOELLE is not sufficient, we revert to using scalar evolution-based protection mechanisms. NOELLE's induction variable optimization enables the protection optimization to be even faster than the scalar evolution optimization; however, the applicability of induction variable-based optimization is a subset of what is provided by scalar evolution.

4.2.1. User Program Compilation

User programs are compiled and linked separately for execution within the kernel. We apply whole-program compilation and optimization to *all* targets—the entire user program and all its dependencies are transformed (Figure 4.2). First, we perform whole program compilation of the application code and fetch the bitcode of the application, which is passed into a set of NOELLE normalization passes, followed by NOELLE's enabler passes, which normalize the code for instrumentation until a fixed-point is reached. Next, we apply the protections pass and the tracking pass. The final bitcode generated at the end of this flow is prepared to be compiled into object code and linked specially for the kernel. These last stages of compilation and user program interaction with the kernel via LCP and the kernel process abstraction is described in §4.6.

4.2.2. Kernel Compilation

To compile the kernel itself, we use most of the compilation flow used for the user program. In particular, we apply whole program compilation, and NOELLE's normalization and enabler passes to the entire kernel. However, we apply only the tracking pass, which enables the kernel to manage its own memory using CARAT CAKE. The kernel code has no guards injected by default and hence behaves much like a monolithic kernel with paging. Additionally, because the kernel is a part of the TCB, CARAT CAKE can allow the kernel to disable tracking for certain parts of the kernel. This feature gives TCB members more control, when the kernel specifies that a section of kernel code need not be tracked, it can safely take responsibility for that section's memory management.

4.3. Kernel

The Nautilus kernel has been augmented in the following ways. Details of processes, which are not native Nautilus entities, are given in §4.6. Here, a process can be thought of as an address space combined with a thread group.

4.3.1. Specialized ASpace

A CARAT CAKE ASpace comprises a set of Memory Regions, which incorporate permissions and represent constructs of the user program such as the stack, heap, `.text` section of the executable, etc. A CARAT CAKE ASpace also contains a set of threads that are currently assigned to it. This is necessary because the context of these threads (stack, and registers) need to be patched on a memory move. The kernel exists as a Memory Region mapped into each ASpace but is only accessible from the user process during a valid "front door" or "back door" entry (§4.6).

4.3.2. Tracking

To implement the runtime side of tracking, the compiler's tracking callbacks drive edits to the *AllocationTable*, a data structure containing a mapping between initialization pointers and Allocations. Each CARAT CAKE ASpace contains a local AllocationTable that tracks allocations within the ASpace's Memory Regions. In addition to storing metadata about the Allocation, the runtime must also track Escapes to those Allocations and store them as metadata within the corresponding ASpace. Each compiler-instrumented Escape invokes the runtime to track it and map it to its corresponding Allocation, establish the reverse mapping in the Allocation's Escape Set.

Runtime tracking of all Allocations and Escapes allows CARAT CAKE to manage memory at the Allocation granularity, where memory movement, defragmentation, remapping Memory Regions, etc. can occur at this granularity. When the kernel CARAT CAKE runtime manages this memory across all existing ASpaces, it effectively manages the entire physical address space of the machine.

4.3.3. Protection

Conceptually, a protection check of an address, invoked by a Guard Injection, or Guard, determines if said address is a member of the set of Memory Regions within the ASpace. More specifically, it requires the CARAT CAKE runtime to perform a lookup in the ASpace using the memory addresses being accessed as keys. Additionally, a Guard Injection compares the requested operation involving address against the permissions of the region that it belongs to.

In conjunction with each Guard's complexity, Guard Injections also dominate both the proportion of instrumentation at compile-time and invocations of the runtime. The real execution time of a region lookup can worsen as the number of regions increases, a real possibility for processes dynamically allocating a large amount of memory, for example. Consequently, each individual Guard must *also* be optimized for performance at run time.

We apply an optimization to boost this performance. In particular, addresses are vetted in a simple hierarchical manner within a Guard, where each level of computation within the Guard is more expansive. In CARAT CAKE, the runtime can check if an address belongs to commonly referenced Memory Regions of the ASpace such as the stack or sections of the executable. This heuristic exploits the understanding that a large portion of memory accesses and references interact with the stack or global state of a program. Second, if further processing needs to occur to find the Memory Region for an address, the CARAT CAKE runtime performs a lookup across the entire set of Memory Regions of the ASpace.

4.3.4. Movement

Any memory management scheme (paging, CARAT, or other systems) must support memory movement to improve inefficiencies that accumulate over time. A paging scheme solves physical memory movement by heavily relying on the ability to manipulate the virtual to physical address mappings. CARAT CAKE must do so by actually moving memory during each update. Fundamentally, this can be accomplished because the CARAT CAKE system has a omnipotent view of all the Allocations that are within a given slice of memory that is being moved, as well as all Escapes from these Allocations.

Because CARAT CAKE can move memory at an Allocation granularity, this enables the system to perform a *hierarchy* of different kinds of moves, with each layer relying on the last. At the finest granularity, unlike paging, CARAT CAKE can move individual Allocations. One layer up, CARAT CAKE can move regions, by moving all the Allocations within a region. One more layer up, CARAT CAKE can move processes, by moving all the regions within a process. The CARAT CAKE runtime can even move the entire kernel if necessary because all kernel code is tracked.

At a first glance, one may argue that an important benefit of paging is that memory can be moved “lazily,” by invalidating pages, faulting on an invalid virtual address, and walking the page tables to fetch the correct pages to use. However, implicit in this “lazy” mechanism is extensive, mandatory address translation for each access, which CARAT CAKE completely avoids. Without mappings, moving an Allocation in CARAT CAKE generates a new destination address for said Allocation and triggers a *patch* of all objects or instructions in the program referencing the Allocation (i.e. pointers). In CARAT CAKE, these references arise from pointers to Allocations that are stored to memory, the Escapes. Essentially, CARAT CAKE performs this change in mapping "eagerly."

This flavor of memory movement is enabled by tracking Allocations and each of their Escapes, as described previously. Patching following a movement simplifies to the runtime simply pointing each Escape to the Allocation's new address. Unfortunately, there are caveats to this approach; because of register allocation by the compiler back-end, an Allocation may escape to a register or to a spilled location on the stack. Consequently, the CARAT CAKE runtime scans the program stack and register state to patch such escapes, similar to a register and stack scan in a conservative garbage collector.

4.3.5. Defragmentation

Because no virtual to physical address mapping exists in CARAT CAKE, it is necessary to address fragmentation through data movement. As shown in Figure 4.3, defragmentation exploits the hierarchy of movements described above. To defragment a Region, we rearrange the Allocations within it. To defragment a process/ASpace, we rearrange the Regions within it. To defragment all memory, we rearrange all processes/ASpaces. Note that the entire process shown does not need to take place; instead, each step can be independently run, or the step-by-step processing can terminate early. Executing all of the steps in order accomplishes a global defragmentation of memory at fine granularity.

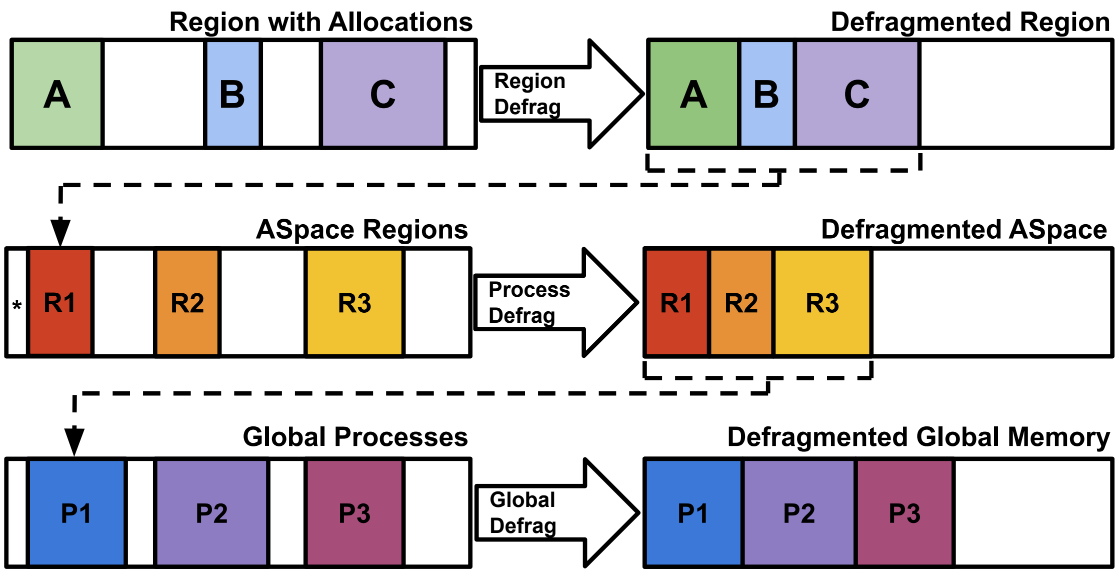


Figure 4.3. CARAT CAKE defragments hierarchically.

Defragmenting a single Region packs the Allocations within the region to the beginning of the Region. For instance, if this step is invoked by the kernel memory allocator, the pointer to the end of the last Allocation (Allocation *C* in the figure) now points to the largest possible free block available within the Region. In the case of global defragmentation, the Region can be packed in this way before moving on to packing processes/ASpaces.

A notable feature highlighted in Figure 4.3 is the ability to move a Region into an overlapping free memory chunk, marked by an * next to *R1* in the figure. Unlike paging, these chunks can be of any granularity. More generally, movement can occur at arbitrary granularity.

4.4. Other Implementation Aspects

Several other elements are necessary background for understanding our evaluation and its limitations.

4.4.1. Granularity and Alignment

Paging enforces protection and allows movement of memory at page granularity. In contrast, CARAT CAKE can in principle operate down to byte granularity. In practice, CARAT CAKE operates at different granularities depending on the context. Protections are enforced on the relevant Region's granularity. Region protections operate in a similar fashion to page protections, with the key difference being that the Region is arbitrary in size. Tracking Allocations happens at the granularity of individual Allocation sizes. Moving memory happens at all levels of granularity including Allocations, Regions, and ASpaces.

4.4.2. Address Space Abstraction Data Structures

In CARAT CAKE, Regions can be allocated and delegated for processes to use by adding them to a process's ASpace. The ASpace abstraction captures a memory map in a manner that is independent of whether CARAT CAKE or paging is being used. The memory map consists of a set of Regions, each of which includes the virtual and physical start addresses, and length, as well as protection bits (read/write/exec/kernel/etc). Because the speed of finding the relevant Region for a virtual address is critical for all ASpace implementations, the data structure is pluggable. Currently, red-black trees (similar to Linux), splay trees [**155**], and linked lists are available. The prototype uses a red-black tree to implement many of its internal data structures. Within the ASpace, red-black trees are used to keep track of memory regions, Allocations (Allocation Table), and Escapes (Escape Set).

4.4.3. Library Allocators

In a general CARAT system, library allocators would be designed around the assumptions of a CARATized process, a CARAT address space, and a region-based design. However, to prototype CARAT CAKE and generate comparable results against paging in Nautilus and Linux, we cannot compare existing library allocators (namely *libc* `malloc`) against a custom allocator for CARAT CAKE. Instead, we allow all CARATized user programs to use `malloc`, and we conform to the assumptions that *libc* `malloc` makes—most notably a virtual address space, where the heap exists as a logically contiguous chunk of memory, and memory management system calls such as `brk`, `sbrk`, and `mmap`. In order to adhere to these assumptions, CARAT CAKE allocates and expands each heap's memory as a contiguous Region of physical memory, which mimics the invariants assumed by `malloc`. Memory management system calls are handled by the Linux compatibility layer in Nautilus (§4.6). If the internal state of the library allocator was visible/mutable to CARAT CAKE, this limitation would cease to exist.

4.4.4. Tracking Stack Allocations

The CARAT CAKE prototype does not individually track each stack variable; the entire stack itself is treated as a singular Allocation. This does restrict the stack in CARAT CAKE to being a single contiguous chunk of memory, but it can be expanded (moving it if necessary) by the CARAT CAKE runtime.

4.4.5. Resolving the Race between Guards and Protection Changes

A protection change and the execution of a Guard might race. This problem is exacerbated when the compiler has optimized the Guard, because such optimizations assume that the permissions are invariant for the duration of the guarded code block. To account for this, the system allows for limited permission-changing functionality via a “no turning back” permissions model. When a Guard is invoked and the permission check succeeds, we store the permissions that have been allowed for the affected region. A subsequent protection change may only downgrade permissions (e.g., going from read/write to read-only is allowed). This is not an intrinsic limitation—the compiler could introduce the code to “release” the region.

4.5. Paging Alternative

While we do compare performance with Linux, there are of course many confounding factors given that it is an entirely different, and much larger codebase. To control for the difference, we have built a substantial and performant implementation of the ASpace abstraction within Nautilus using x64 paging.

Each address space has its own distinct page table hierarchy that is based on the 4-level x64 paging model. Page table mappings can be constructed eagerly or lazily on demand. In addition to the basic 4 KB pages, we also support large (2 MB) and huge (1 GB) pages. Nautilus uses buddy system allocation and a side-effect of this is that allocations of physical memory are aligned to their own size. As a consequence, our paging implementation has many more opportunities to use larger pages, and it aggressively uses them. Thus we maximize the reach of existing TLBs. We also minimize the cost of TLB flushes by implementing Intel’s Process

Context Identifier (PCID) feature. Using PCID, it is not necessary to flush TLB content on a context switch. IPIs are used to implement remote TLB shootdowns when needed.

4.6. Linux Compatible Process (LCP)

The Linux compatible process (LCP) implementation allows for separate compilation and linking of the application and kernel, much like the Linux user-level model. However, the separately compiled application executable is dynamically loaded and run as part of the kernel. Unlike a kernel module, however, it is not linked to the kernel, but rather runs within a specialized kernel-mode process abstraction that emulates the Linux system call and signal interfaces. More details about our process abstraction and LCP can be found in an earlier paper [**113**]. The "PIK" approach described in that paper for executing OpenMP parallel programs as components of the Nautilus kernel provides neither CARAT- nor paging-based protections or memory management.

4.6.1. Code Generation and Linking

The CARAT CAKE compilation process adds tracking and protection to the user program at the LLVM IR level. When we build the program for paging, these steps are simply not done. The ordinary Linux user-level build process and the LCP build process are virtually identical. Whether we target CARAT CAKE or paging, we must handle back-end code generation and linking slightly differently. These details are hidden in wrappers for `cc` and `ld`.

Code generation uses position-independence (`-fPIE`) to allow the resulting executable to be loadable into any physical address, and to be movable during execution, if necessary. A custom linker script is used to integrate all components via static linking.[1] The linker script preserves the position-independence of the entire linked executable (creating a "static PIE" executable). The compiler, C, and C++ runtime startup code (e.g., `crt0`) is integrated carefully, and with an assumption that the kernel will be providing a "pre-start" environment for it. A special multiboot2-like header is to simplify the ELF load process. This header also contains the attestation signature for CARAT CAKE.

4.6.2. Process-in-Kernel Abstraction

Nautilus has been extended with a process abstraction that combines a kernel thread group, an ASpace (either CARAT CAKE or paging), and (optionally) a custom allocator. A special loader brings the executable image into physical memory at any convenient location, and initializes BSS/TBSS/etc, as well as an initial stack+heap, all allocated in physical memory. A process launch begins in an initial thread that runs a wrapper function (the "pre-start" code) that completes the setup of the process before invoking the user's thread function. Child threads start similarly, and then join their parent's ASpace.

4.6.3. CARAT CAKE and the Trusted Back Door

Code introduced via CARAT CAKE compilation, for example protection checks, tracking, and (optionally) allocation, invokes the kernel-level CARAT CAKE run-time via a function and data table that is advertised to the process somewhat like a Linux vDSO. However, the compilation process assures that only the injected code has access to this trusted backdoor into the kernel. Because no system calls or other boundary crossings are involved in using the trusted backdoor,

[1]All code ultimately linked into the executable must have been transformed by the CARAT CAKE compilation process, or must have separate attestation of that protection and tracking has been manually added.

the run-time operation of CARAT CAKE is a unified whole across all processes and the kernel. The CARAT CAKE runtime uses this feature to quickly invoke the kernel without a system call.

4.6.4. Linux Compatibility and the Untrusted Front Door

To allow Linux programs to run, the process abstraction must provide compatibility with Linux system calls and signals, as well as some of the expected userspace environment. We implemented a subset of this functionality for CARAT CAKE. Our system call interface uses the `syscall` instruction (or `int 0x80`), not the vDSO. Unlike in Linux, a system call in Nautilus happens in the same address space, at the same privilege level (which complicates the return from system call slightly), and using the same stack as the calling thread (red zone is avoided for both mechanisms).

The most important system calls (i.e. those used by the C runtime and by library allocators – e.g. `malloc`) are largely implemented while other, more sparingly used Linux syscalls are stubbed so that we can see all activity, and respond, by default, with an error. A Linux-compatible signal installation (e.g. `sigaction()`), signal assertion requests (e.g. `kill()`), and signal delivery mechanisms were added to Nautilus. The latter required substantial modifications to low-level thread context-switch processing. Again, the focus in this prototype was on signals required to execute our benchmarks.

4.7. Evaluation

We now evaluate the CARAT CAKE prototype in terms of performance and in terms of the engineering effort necessary.

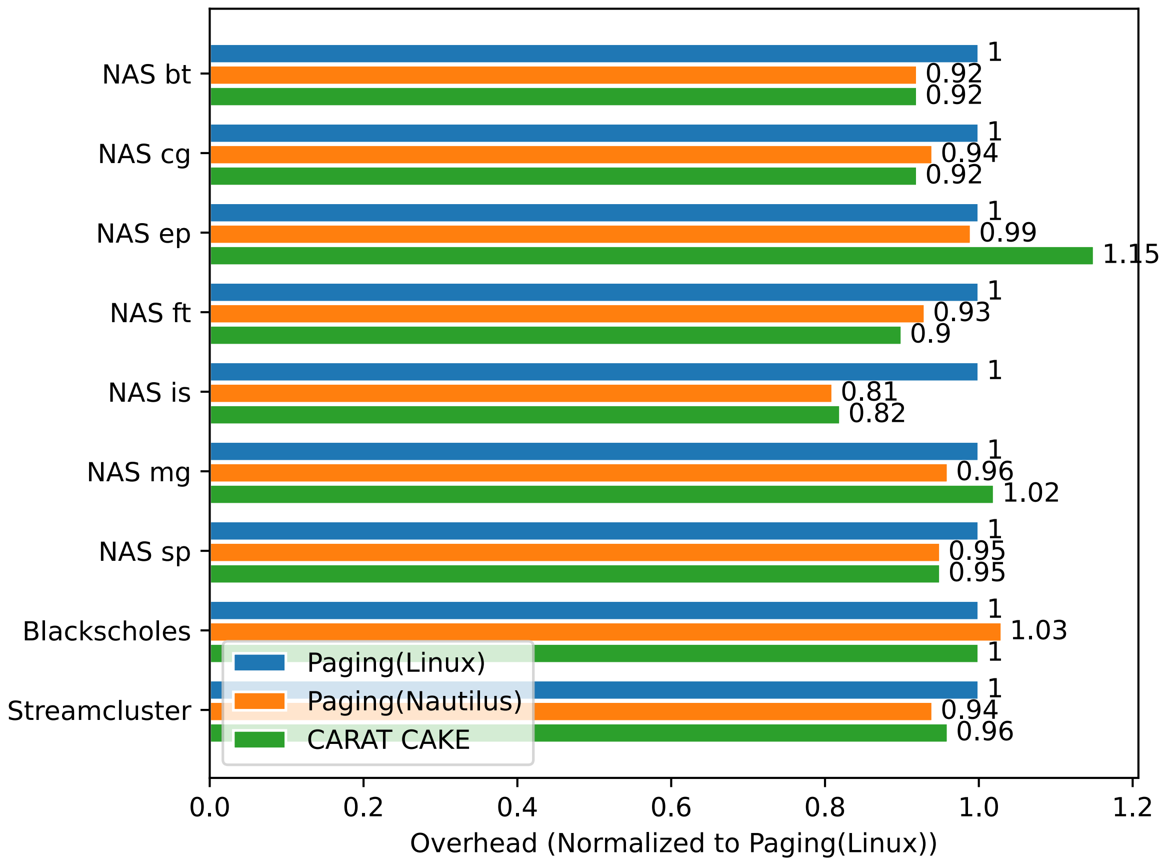


Figure 4.4. CARAT CAKE has comparable run time overheads.

Steady-state Overhead is Similar to Linux. By far the most common situation for a workload is steady-state operation, in which the kernel is making few changes to the virtual to physical mapping (for paging), physical location of allocations (for CARAT CAKE), or protections. Steady-state operation is where we would expect to see maximum performance and energy gains in future hardware that would eliminate hardware support for paging or allow it to be disabled.

Figure 4.4 compares the measured performance of our benchmarks on our target platform using Linux as a baseline for normalization. As can be seen, CARAT CAKE and paging in

Nautilus are comparable to Linux.[2] This is the main takeaway—the tracking and protection overheads from the compiler-injected CARAT CAKE code in the kernel and the user program prove to be quite small in practice.

It is important to understand that CARAT CAKE is still using paging here because of the inability to deactivate it on x64. While the identity-mapped page tables used are at 1 GB granularity and thus TLB misses are rare in steady state, CARAT CAKE is still paying the cost of having a TLB in the first place, namely on control path length in the logic of the processor and in virtual addressing-imposed restrictions on the L1 cache design. Despite this, its overhead is comparable to that of the two different paging implementations.

Memory Movement Costs are Reasonable. Unlike paging, the fact that CARAT CAKE uses physical addressing means that the need to be able to migrate allocations is inevitable. For example, a failing allocation will need to be followed by a defragmentation, similar to a garbage collection pass in a managed language.

To measure the effect on performance of such migrations, we developed a tool that competitively "peppers" normal execution of a benchmark with migrations. More specifically, $pepper(rate, nodes)$ is a separate thread that maintains a linked list of $nodes$ elements. It wakes every $\frac{1}{rate}$ seconds and migrates the linked list, element by element, to a new memory region. The benchmark sees a pause while this is accomplished. This is then measured as a slowdown compared to the "unpeppered" benchmark.

[2]Indeed, both Nautilus paging and CARAT CAKE exhibit similar and slightly better performance than Linux paging. Our point here is not to "beat" Linux, but rather to evaluate whether CARAT CAKE is a viable alternative to paging.

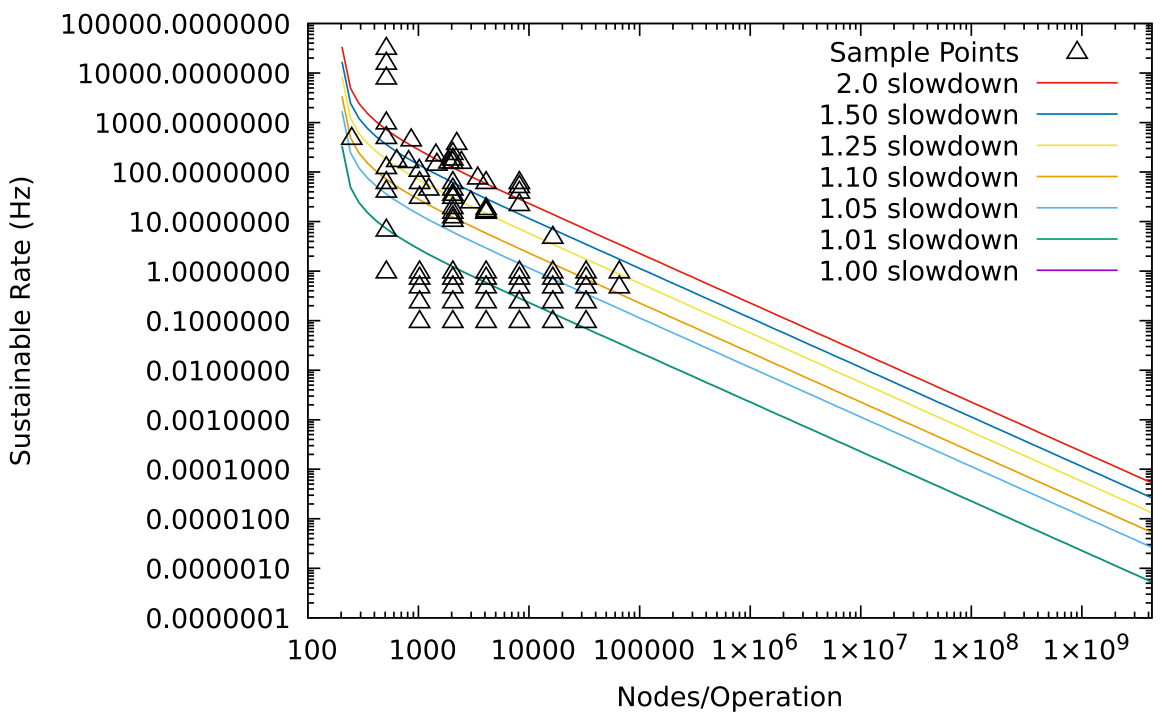


Figure 4.5. Possible $rate$ and $nodes$ (size) combinations given various constraints on application slowdown (NAS IS).

Note that this processing involves finding and patching $nodes$ pointer escapes, moving the actual data, and the synchronization overhead (a world stop/start across 64 cores). The slowdown that the system (the benchmark in these measurements) experiences is dominated by synchronization at high rates (the measured maximum possible rate is ~26 KHz). At lower rates, the pointer escape finding/patching and memory movement, particularly the former, dominate—this is the regime the system will mostly operate in. In our pepper data collection, we sample the space of $rate$ and $nodes$.

Compactly presenting the pepper results is challenging because of their inherently 3D nature and because slowdown might be better thought of as a constraint. To do so, we fitted a

Benchmark	**Num. Allocations**	**Max Escapes**	**Pointer Sparsity (φ)**
pepper (linked list)	*nodes*	*nodes*	**8 B/ptr**
Nautilus Kernel	*944*	*34K*	*105 B/ptr*
Streamcluster	8.9K	66	2 MB/ptr
Blackscholes	36	25	26 MB/ptr
SP	149	1	83 MB/ptr
MG	247K	494K	921 B/ptr
FT	70	27	16 MB/ptr
EP	82	1	2 MB/ptr
CG	67	1	62 MB/ptr

Figure 4.6. Many programs display high pointer sparsity (φ).

physically-inspired model, specifically

$$slowdown(rate, nodes) = 1 + (\alpha + \beta \times nodes) \times rate$$

which includes the costs described above, using regression. The resulting fit for α and β exhibits $R^2 = 0.9924$. The model then allows us to create characteristic curves in which we constrain slowdown and then sweep $nodes$ to project the maximum possible $rate$ that can be achieved. Figure 4.5 shows these characteristics for the IS benchmark. To interpret the figure, choose a desired slowdown. The corresponding curve divides the space of rates and nodes into two: the combinations below the curve are possible. The key point, however, is that with a reasonable constraint on overhead, 10% for example, to match the gains seen in Figure 4.4, quite high migration levels can be sustained. Also quite large migrations can be sustained at lower rates.

Ideally, the duration of migration (and thus performance impacts on applications) would be limited only by the `memcpy()` performance. How close we can get to this limit is determined by the *pointer sparsity*, φ, which we define as the ratio of the amount of data moved to the number of pointers that are updated. As φ increases, we approach the `memcpy()` limit.

Component	**Lines of code** Paging	**(Engineering Effort)** CARAT CAKE
Compiler		
Tracking		2,066
Protection		1,563
Build changes		50
Compiler total		3,679
Kernel		
Paging	3,250	
Allocator changes		300
Tracking runtime		2,662
Migration support		949
Heap/stack expansion	100	100
Defragmentation		100
Kernel total	3,350	4,111
Compiler (Reusable)		
Building	(Average)	(Heavy)
Optimizing	(Average)	(Heavy)
Architecture		
Memory management	(Heavy)	(Minimal/None)
Other	(Average)	(Average)
Total	3,350	7,790

Figure 4.7. Breakdown of implementation sizes.

In the pepper results (Figure 4.5), we deliberately consider a *low* sparsity move ($\varphi = 8$ for a 64-bit pointer linked list being moved.) Figure 4.6 puts pepper in context by showing φ for the other code we consider, including the kernel itself. As can be seen, many benchmarks display very high φ. If we were to move one of these instead of a linked list, the curves shown in Figure 4.5 would shift upward.

Engineering Effort is Likely Comparable to Paging. Converting a kernel that already extensively and intrinsically uses paging to use CARAT CAKE may be quite challenging. We cannot provide direct input on that challenge because we started with a kernel that effectively uses physical addressing (identity-mapped paging). However, we can compare and contrast the effort involved in adding CARAT CAKE and paging to that kernel. Figure 4.7 breaks down the code sizes involved for both approaches. The shared code (ASpace, LCP, etc) is not included.

As can be seen, the implementation costs are similar (within a factor of two in LoC), but the cost is shifted to the kernel for paging and to the compiler for CARAT CAKE.

The figure also compares qualitative aspects that cannot be easily captured in LoC, such as the reliance of each approach on the compilation frameworks and underlying architectures, and the effort involved in these.

CARAT CAKE requires the expansion of the software TCB, particularly in additions to the the compiler, as well as compilation libraries used for optimization, such as NOELLE. These efforts may introduce new attack vectors via bugs in the compilation framework being used. With paging, if there is a compiler bug, the consequence is a kernel with a likely random bug. With CARAT CAKE, because we rely on the compiler for protection, a compiler bug may be more easily used to subvert protection. Because of this, CARAT CAKE demands larger engineering investment into the compiler's development and verification.

On the flip side, because CARAT CAKE avoids memory management hardware, this also removes attack vectors that might be present in the hardware TCB. Note further that bugs/attack vectors found in hardware require much more effort to patch than software bugs due to the physical nature of the hardware. Indeed in some cases the bugs may not be patchable at all, such as we have experienced with the Spectre and Meltdown vulnerabilities [**111, 99**]. Because of this, in a future hardware system that used CARAT CAKE the architectural engineering effort and verification would be significantly decreased compared to paging.

This proposed shift of engineering effort from hardware to software also fits well into the current and developing landscape. Microarchitecture development efforts, save for a few up and comers like OpenPiton, are *closed* development efforts under the control of industry. At the same time, they are cornerstones to the TCB, something made clear by recent vulnerabilities.

In contrast, compiler development efforts are often *open* projects (LLVM being the popular example) that have *significantly* more participants. Arguably the number of participants is also increasing at a much faster rate than with microarchitecture. That the engineering effort of CARAT CAKE is clearly tractable, and can leverage this milieu is compelling.

4.8. Contributors

This project was accomplished through a large effort beyond just myself for many components of this work.

Souradip Ghosh, Drew Kersnar, and Gaurav Chaudhary were responsible for development of the compiler and runtime as well as the supporting framework. Siyuan Chai and Zhen Huang were responsible for the CARAT Aspace abstraction that interfaced with the CARAT CAKE runtime. Aaron Nelson and Michael Cuevas were responsible for the Linux Compatible Process In Kernel work as well as Signals. Alex Bernat was responsible for work on the defragmentor and the allocator. Additionally, myself, Nikos Hardevellas, Simone Campanoni, and Peter Dinda were responsible for various overlapping parts of this project as well as offering guidance.

CHAPTER 5

CARMOT: Low Cost Computational Spoor Discovery with the Compiler And Runtime Memory Observation Tool

Another use case for TEXAS is for simply examining the usage of memory that a program makes. Programming languages evolve to give programmers powerful abstractions that improve performance, energy savings, and code clarity. Examples include OpenMP features, and C++ smart pointers. Programmers often struggle to use abstractions properly because of the implicit requirement that the programmer must be omniscient regarding the behavior of the whole program, especially of the region where the abstraction is to be applied. This results in programmer errors that lead to both performance and correctness issues.

For the most part, programmers are left alone in the dark with the complex task of properly using programming languages abstractions [**158**]. While some existing tools specifically target OpenMP [**20, 141, 110, 130, 40, 121**], they either use limited static analysis approaches, or over-specialized dynamic strategies that are opaque to the programmer, suffer from false positives, and are limited to a tiny subset of OpenMP. We lack a general approach to assist programmers in using a wider and more complex set of abstractions.

This is the first paper observing that many abstractions rely on a common piece of information related to the access pattern of program's variables and memory locations, in other words the program state. Our approach studies this access pattern for the code region where the abstraction is to be applied. Texas defines a new concept that summarizes the impact of this access pattern, which it calls the *computational spoor*[1]. The computational spoor describes: (a) which, where, and how variables and memory locations are used in a code region, (b) how data flows across code region boundaries, and (c) the reachability relationships between different memory locations. Intuitively, the computational spoor of a code region assists programmers by formalizing the mental process they employ when thinking about abstractions. For example, when parallelizing a program's loop using OpenMP, a programmer needs to understand which variables and memory locations are only read, which are written, and in what order. Only read variables and memory locations are shared across parallelizing threads, while variables and memory locations that are written have to be privatized, reduced, or enclosed in a critical section depending on whether their value flows across loop iterations and the type of operations performed on them. The computational spoor formally characterizes this information and presents it to programmers in human-readable form by reporting source code level information in the form of instances of the target abstractions that need to be considered by programmers as recommendations.

Texas refer to the process of generating the computational spoor as *Computational Spoor Discovery (CSD)*. Today, no tools are capable of performing CSD. Even worse, building a tool capable of automating CSD is challenging for two reasons. First, automating CSD with only static code analyses is a challenge due to potential memory aliasing and caller-callee relations that are unknown at compile time. Second, performing CSD at run-time is challenging because of its computation and memory requirements. This is because CSD requires tracking the whole memory of a program as well as all variables and all their run-time accesses performed within the abstraction's code region. Notice that although tools exist to track the memory of a program,

[1]***spoor*** · a track, a trail, or scent, especially of a wild animal.

such as Valgrind [**128**] and AddressSanitizer [**152**], they cannot perform CSD because they only track memory accesses. In other words, they ignore accesses to function's variables (allowing them to operate with low overhead), which form the vast bulk of program state accesses (Section 5.1.3). But these are needed for CSD. Further, existing tools have no need to integrate information about accesses, which is required for CSD, and therefore they do not need to preserve access information (allowing them to operate with low memory). This paper describes the first system capable of automating CSD.

TEXAS is heavily utilized to implement CSD in a specialized system called *Compiler And Runtime Memory Observation Tool (CARMOT)*. CARMOT has three components:

- An LLVM-based compiler that includes CSD-specific static code analyses and optimizations. These optimizations keep the CSD overhead manageable even for a large codebase. Furthermore, the CARMOT compiler instruments the target program for the profiling phase of CSD.
- A variant of TEXAS's runtime co-designed with CARMOT's compiler to efficiently perform CSD by tracking all allocations, deallocations, and memory changes during program execution following compiler's directives.
- A custom binary instrumentation tool based on Pin [**112**] that performs CSD when program execution extends into pre-compiled libraries.

To demonstrate the power and flexibility of the computational spoor, CARMOT is applied to five important abstractions that range from traditional to emerging. The abstractions are: OpenMP critical section, OpenMP parallel for, OpenMP task, C++ smart pointers, and the Input-Output-State abstractions needed by the STATS compiler for non-deterministic programs [**60**]. A programmer invokes CARMOT specifying an abstraction they would like to use on a given code region (e.g., OpenMP parallel for). CARMOT generates recommendations to programmers by synthesizing the target abstraction using the proper abstraction's parameters that reflect the program's run that was profiled (e.g., the various critical sections, the variables that need to be shared between threads, and which objects to clone). Section 5.3.1 explains how to combine computational spoors generated using different runs and inputs of a program. Furthermore, programmers can also use CARMOT to verify the correctness, or to improve the performance, of existing uses of an abstraction for specific runs of the target program. CARMOT demonstrates the power of its recommendations by automatically verifying the correctness of existing pragmas, and recommending new pragmas for the aforementioned OpenMP abstractions on the most relevant benchmarks from PARSEC, SPEC CPU 2017, and NAS using the inputs available in these benchmark suites. On average across all benchmarks, the pragmas that result from CARMOT's recommendations speed up program execution by $8\times$ compared to the original, serial version, and they match or outperform the manually (and labor-intensive) parallelized version. CARMOT also shows how it can be used to find and break cycles in reference counting garbage collection used by C++ smart pointers, so programmers can safely use this abstraction without introducing memory leaks. For example, CARMOT was able to easily discover a complex reference cycle in the SPEC CPU 2017 benchmark `nab` that spans across multiple files and functions. Finally, the STATS abstraction automatically generated by CARMOT in a few minutes outperforms those that the STATS authors manually defined after six months of work.

STATS is a compiler optimization that optimizes non-deterministic applications by observing the memory state of the program in order to 'fast-forward' certain computations of the application. This project was published in ASPLOS 2018 [**59**] as well as a follow up characterization paper in ISPASS [**58**]. One of the current limitations of STATS is the fact that programs that wish to use STATS must be manually **and correctly** instrumented to capture the state space of the program. This causes issues for any program that has many files or significant complexity.

5.1. Background

Programming languages constantly evolve to include new abstractions that enable programmers to take advantage of the latest hardware while developing easier-to-maintain code. However, *actually using* new abstractions in a large codebase is challenging and error-prone due to the lack of tools to assist programmers. We describe three use cases exploring five different abstractions and how they support and challenge programmers. We show how the computational spoor is the common information necessary to build tools that help programmers use these abstractions. Then, we discuss why other memory tracking tools cannot be used to compute the computational spoor.

5.1.1. Program Parallelization and Synchronization

Writing performant parallel code often requires programmers to (i) declare concurrency, (ii) synchronize threads, and (iii) clone memory locations and variables to make them thread-private. However, writing parallel code using low-level APIs like pthreads increases development and maintenance costs. OpenMP includes high-level abstractions that make synchronization and cloning easier for programmers. For example, OpenMP includes directives to synchronize parallel accesses (*#pragma omp critical/ordered*), parallelize loops (*#pragma omp parallel for*), and asynchronously execute units of computation (*#pragma omp task*).

To understand the challenge of using high-level abstractions, let us consider the *#pragma omp critical* pragma. This pragma provides coarse-grain synchronization, allowing only one thread at a time into the section to update the data referenced therein. To decide which code needs to be wrapped in a critical section, programmers must precisely understand how the program state (variables and memory locations) is accessed (read/written) potentially by all source code statements. If the critical section does not include all the necessary code, then the generated parallel code will be incorrect. Conversely, if the critical section includes unnecessary statements (i.e., they operate on unshared data), the resulting program will exhibit less concurrency, lowering performance and scalability, and potentially defeating the ultimate goal of using OpenMP.

Another high-level abstraction is *#pragma omp parallel for*, which distributes the iterations of a loop between the available cores of the underlying architecture. This pragma asks programmers to declare variables as *private* or *shared* to indicate whether they need to be cloned or shared between threads. Furthermore, programmers must make more decisions for variables declared as *private* about how the private copies need to interact with the code outside the target loop (e.g., *first private*). To use this abstraction, programmers again must understand how the program state is accessed by the target code region (a loop in this case). When programmers fail to identify all variables that need to be private, the result is an incorrect program with data races. Also, when programmers clone variables that could have been shared between threads, the result is an inefficient program with unnecessary copies.

Modern OpenMP also offers another high-level abstraction, *#pragma omp task*, to asynchronously execute a code region that represents a unit of computation. Tasks can depend on each other (e.g., the output of task 1 is used by task 2). Such dependences are expressed with the *depend((in—out) : variables)* attribute. To correctly generate task dependences, programmers must understand how the program state is accessed in every task code region and how the program state in each task interacts with the others. Forgetting dependences leads to an incorrect parallel program.

5.1.2. Managing Dynamic Memory

C++ programmers used to manually manage the dynamic memory of a program. To help with this task, modern C++ standards (as of C++11) have added the smart pointer abstraction. Smart pointers manage dynamic memory using reference counting, which tracks the number of pointers to an object and deletes it when the count drops to zero. Unfortunately, using smart pointers can lead to memory leaks when there are cycles in the reference counting graph of allocations. Programmers have limited tool support to detect when cycles occur and no support to help break them. This is particularly challenging when reference cycles cross many functions and source code files, making their detection done by a programmer challenging.

5.1.3. Declaring *State Dependences* with STATS

STATS [**60**] is a compiler for parallelizing non-deterministic programs. STATS requires a programmer to follow a given code structure (i.e., the Input-Output-State abstraction of STATS), which makes the compiler aware of the STATS *state dependence* (i.e., a Read-After-Write (RAW) dependence that can be satisfied in a different way following the STATS execution model). To do so, a programmer needs to classify the subset of the program state accessed by the code region where STATS will operate into three classes: 1) Input class (program state that is only read), 2) Output class (program state that is written first), 3) State class (program state that is read first and then written). Understanding which part of the program state goes into these three classes is challenging and often requires a programmer to understand the behavior of the entire program. Misclassifications lead to either performance degradation or an incorrect program.

5.1.4. Computational Spoor Discovery

All above abstractions require programmers to answer the following questions: “How does a code region interact with the program state? Where, when, and how did the region affect it?” To answer these questions we use the computational spoor, which conveniently summarizes the knowledge of how the program state is affected by a code region. We now give an intuition about how to collect and apply the computational spoor to use the abstractions *#pragma omp critical/ordered* and *#pragma omp parallel for*. Section 5.2.4 formalizes this process for the other abstractions. Consider the loop in Figure 5.1 to be the code region where a programmer wants to apply *#pragma omp parallel for*. CSD would classify the program state affected by the loop’s body as follows: variables *a* and *b* are only read, variables *x* and *i* are always written before being read, and variable *y* is read and then written. With this information, CARMOT generates the *#pragma omp parallel for* with the correct attributes and critical section. In more details, CARMOT lists variables *a* and *b* as *shared* because they are only read, hence multiple threads can read their value at the same time without making extra copies. Variables *x* and *i* are instead declared as *private* because their value changes throughout loop iterations, hence

```
1  int work(int a, int b){
2    int i, x, y;
3    #pragma carmot roi{
4      y = 42;
5      for (i = 0; i < 10; ++i){
6        #pragma carmot roi{
7          x = i/(a + b);
8          y /= a*x + b;
9        } // end roi
10     }
11   } // end roi
12   return y;
13 }
```

Figure 5.1. CARMOT automatically builds the computational spoor containing the information to parallelize this for-loop.

multiple threads must have private copies to work on. Finally, variable y introduces a RAW loop-carried data dependence and cannot be put into a *reduction* clause because of the division operation performed on it. So, CARMOT recommends wrapping the statement that uses y (i.e., line 8) in *#pragma omp ordered* because the division operation is not commutative, hence the order of operations must be preserved.

5.1.5. Overhead of Computational Spoor Discovery

CSD requires tracking of significantly more program state accesses than what prior tools had to handle. These extra events create overhead that prior tools do not have. This is why our approach requires a more involved compiler-based solution including several CSD-specific compiler and runtime optimizations. In more detail, on top of tracking all memory accesses like prior tools do [**152**], CSD requires tracking of all accesses to all function's variables of a target code region. To understand the impact of tracking function's variables, we measured the increase in accesses that must be tracked when function's variables are also considered (on top

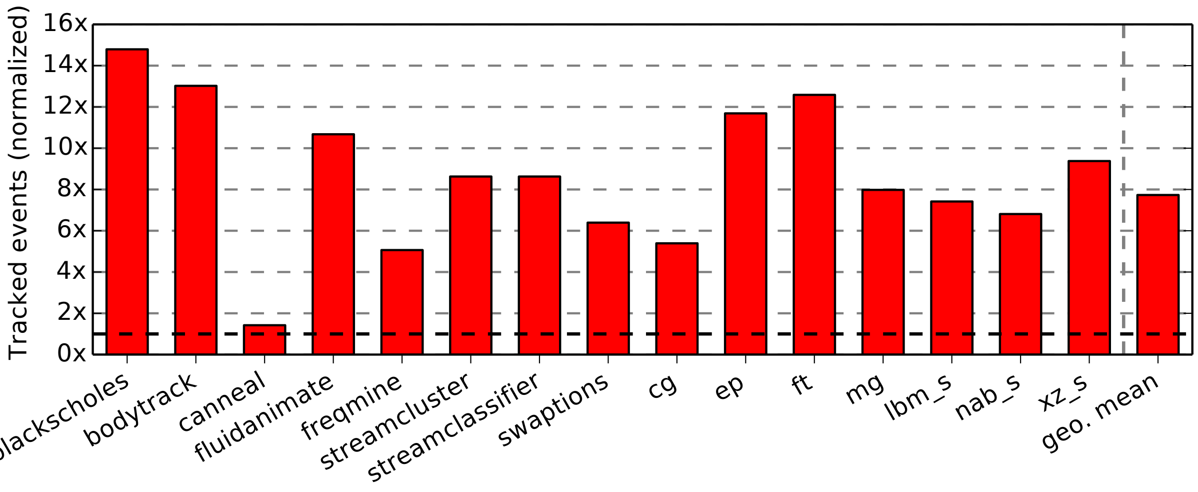


Figure 5.2. CSD requires tracking $8\times$ (geo. mean) more memory-related events than tools like AddressSanitizer.

of memory accesses). Figure 5.2 quantifies this increase. On average, performing CSD requires tracking $8\times$ more events than what prior tools track (the baseline $1\times$ is the number of memory accesses).

The reason why prior tools did not have to track function's variables is that their goal is to validate memory accesses. Hence, these tools are able to invoke many general-purpose compiler optimizations, which would make CSD impossible. An example of optimization that these tools can make use of is mem2reg (among many others prior tools can rely on), as they can tolerate an ambiguous mapping between the source code variables and the compiler's IR variables. Because the end goal and requirements (such as the reporting of source code level information of where variables have been declared) of CSD are substantially different than the memory correctness tools, many optimizations (like mem2reg) can not be generally applied. This combined with the need for tracking function's variable accesses results in a significant number of program state accesses that CSD-tools have to track.

5.2. What is a Computational Spoor?

A computational spoor is comprised of three components (Figure 5.3): **Sets** to classify the program state, **Use-callstacks** to contextualize computation, **Graph** to represent reachability relationships within the program state. In this section we describe these components and how they are used to automatically generate the abstractions of Sections 5.1.1— 5.1.3.

5.2.1. The Computational Spoor and its Sets

We define *program state* as the set of memory locations (stack and heap) and variables (both local and global) of a program at the source code level. Also, we define as Region of Interest (*ROI*) a single-entry single-exit (SESE) code region [**89**]. Examples of an SESE are a loop, an if-then-else code block with its join node, a single statement, or a function. A computational spoor is related to an ROI and contains information about how the program state is read and/or written by that ROI.

A computational spoor classifies the ROI's access of the program state into four **Sets**. Each set indicates how an ROI in the source code interacts with (i.e., reads/writes) the program state members (e.g., memory locations). The sets that comprise a computational spoor for a dynamically invoked ROI Z are:

Input set: Members read by a dynamic invocation of Z before being written by any invocation of Z. This set represents the input of Z as these data are generated by the code outside Z and consumed by Z.

Output set: Members written in a dynamic invocation of Z and read outside Z. This set represents the output of Z as this data is generated by Z and consumed by the code outside Z.

Cloneable set: Members written by more than one invocation of *Z* where no subsequent invocation reads them before overwriting them. This set represents data locations reused by invocations of *Z* without triggering RAW data dependences.

Transfer set: Members written by an invocation of *Z* and then read by a subsequent invocation of *Z* before any potential overwrites. This set represents the data generated by an invocation of *Z* and consumed by a subsequent invocation of *Z*, triggering a RAW data dependence.

Three pieces of information are necessary to classify members of the program state in the correct set of a computational spoor. First, we need to know *where* members of the program state are allocated in the source code. Second, we need the *context* of such allocations. As context we use the callstacks that lead to the code statements that performed such allocations. The context of allocations is necessary because the same static code statement that generates members of the program state can be used in different parts of the program, and the programmer must be able to distinguish them. For example, custom allocators are widely used in large codebases, without knowing the callstack all allocations would look like they are coming from the allocation statement in the custom allocator, which is not useful information for a programmer. Third, we need to *record every read and write* the computational spoor's ROI performs on every member of the program state to classify them correctly. We call these accesses *uses* of the program state.

Finally, members of the program's state need to be evaluated independently between different ROIs. This is because computational spoors of different ROIs might classify the same member of the program state differently. For example, in Listing 5.1 there are two ROIs. For the innermost ROI, variable *y* belongs to the Input, Output, and Transfer sets of this ROI's computational spoor, following the sets definitions. However, the same variable *y* belongs only to the

Output and Cloneable sets of the outermost ROI's computational spoor since *y* is always written before being read. Evaluating potentially multiple times the same member of a program's state increases the computational demand for CSD when multiple ROIs are considered, which is the common use-case.

5.2.2. The Computational Spoor and its Use-Callstacks

The program statements in an ROI (i.e., the uses of the program state in that ROI) can be executed multiple times from different parts of a program. To take this into account, we record the callstack of each invocation of these statements. We refer to these statements and their recorded callstacks as **Use-callstacks** of a computational spoor (Figure 5.3). Knowing **Use-callstacks** enables us to disambiguate a static statement when invoked from different parts of the program, which can lead to a program state member being classified in different **Sets** of a computational spoor. In more detail, consider an ROI which contains two calls to the same function, which has multiple input arguments. Suppose that function overwrites one argument with the value of another. If these arguments alias (i.e., they are the same program state member), the program state member will belong in the Transfer set. Otherwise, such member does not belong to the Transfer set. To differentiate between these two cases we must understand how the function was called. We gain this information from the **Use-callstacks**.

5.2.3. The Computational Spoor and its Graph

The computational spoor **Sets** characterize the data transfers within and across an ROI, while **Use-callstacks** contextualize data accesses. While both types of information are necessary to generate abstractions, they are not enough: they do not include information about how a given

Abstraction	**Computational Spoor**		
	Sets (I,O,C,T)	**Use-callstacks**	**Graph**
OMP parallel for (and critical/ordered)	Yes	Yes	No
OMP task	Yes	No	No
Smart Pointers	Yes	No	Yes
STATS	Yes	No	No

Figure 5.3. Different abstractions need different parts of an ROI's computational spoor.

member can be reached, through which pointer chains. For example, a program state member may be a pointer to another one and the Sets or Uses callstacks do not include such information. For use cases such as Smart Pointers (Section 5.1.2), tracking reference information is vital. Therefore, the computational spoor also includes a directed **Graph** where nodes are program state members allocated within the computational spoor's ROI and edges are references that point to other program state members.

5.2.4. From Computational Spoor to Abstractions

An ROI's computational spoor is used to generate recommendations for programming language abstractions. Programmers declare the abstraction to apply to a given ROI to CARMOT. Then, CARMOT uses a ROI's computational spoor to automatically generate, when possible, new source code with the requested abstraction in it and customized it with the correct attributes. When source code cannot be generated, essential information about how to change the code is reported. Figure 5.3 illustrates which parts of a computational spoor are necessary to generate each abstraction.

Program parallelization and synchronization. To generate a *#pragma omp parallel for* with the correct attributes CARMOT uses the **Sets** of the computational spoor as follows. For every element e in the Cloneable set, CARMOT extracts the callstack for the element's allocation.

These program state members and their callstacks tell us what needs to be cloned to remove WAR and WAW data dependences between invocations of the related ROI (i.e., the body of a loop). If e is a variable, then CARMOT privatizes it in the generated pragma. Variables that are also in the Output set are declared as *lastprivate*, since they can be read after the ROI. Similarly, variables that are also in the Input set are declared as *firstprivate*, since they were first read inside the ROI. If e is a memory location, then CARMOT advises programmers to clone the related memory object (CARMOT's output provides the allocation site and its callstack to help programmers understanding how to perform the cloning) and use the OpenMP API *omp_get_thread_num()* to access the correct clone of that allocation in the ROI. Elements that belong only to the Input set are declared as *shared* in the pragma because they are only read. Finally, For each element e in the Transfer set CARMOT retrieves its **Use-callstacks**. If e is a variable, then CARMOT checks each use of e to understand if the computation performed on e is reducible (i.e., the statement uses one of the OpenMP-supported reduction operators such as $+$). If the computation is reducible, then CARMOT includes e and the supported operation in the *reduction(operator:variable)* attribute. Otherwise, all statements that access e are wrapped in a *#pragma omp critical* or *#pragma omp ordered* section. Note that CARMOT leaves the decision as to which abstraction to use, either critical or ordered, to programmers as they know whether it is necessary to preserve the loop iteration order.

CARMOT generates dependences for *#pragma omp task* as follows. The Input and Output sets of a computational spoor are mapped to the *depend* attribute of *#pragma omp task*. All elements e of the program state in the Input set are declared as *depend(in:*e*)*. Similarly, all elements e of the program state in the Output set are declared as *depend(out:*e*)*.

Managing dynamic memory. Cycles between program state members allocated in an ROI's computational spoor are detected using the computational spoor **Graph**, which tracks references between program state members. Our tool can also suggest to programmers which references should become weak pointers[2] to break all detected reference cycles. CARMOT detects which reference in a cycle needs to become a weak pointer by identifying the node in that cycle that has the oldest access time. This enables programmers to gradually port single modules within a large codebase to use smart pointers without introducing cycles.

Declaring *state dependences* with STATS. The STATS abstraction Input-Output-State can be mapped directly from the **Sets** of an ROI's computational spoor. Elements classified in the Input, Output, or Transfer sets are respectively mapped to the Input, Output, State classes of the STATS abstraction. The STATS abstraction requires the target ROI to be explicitly moved into a separate function; hence, elements in the Cloneable set are declared locally in that function. This localization enables the STATS compiler to spawn independent parallel threads to execute the related ROI.

5.3. CARMOT

We designed and implemented CARMOT to perform computational spoor discovery (CSD) automatically and efficiently. It does so by profiling the execution of the target program and generating the computational spoor for that execution. The computational spoor is then used to automatically generate source code information for the programming language abstractions described in Sections 5.1.1– 5.1.3. After introducing how CARMOT's design performs CSD, this section describes the internals of CARMOT's compiler, Pintool, and runtime.

[2]A weak pointer is a reference to an allocation that does not increment the allocation reference counter.

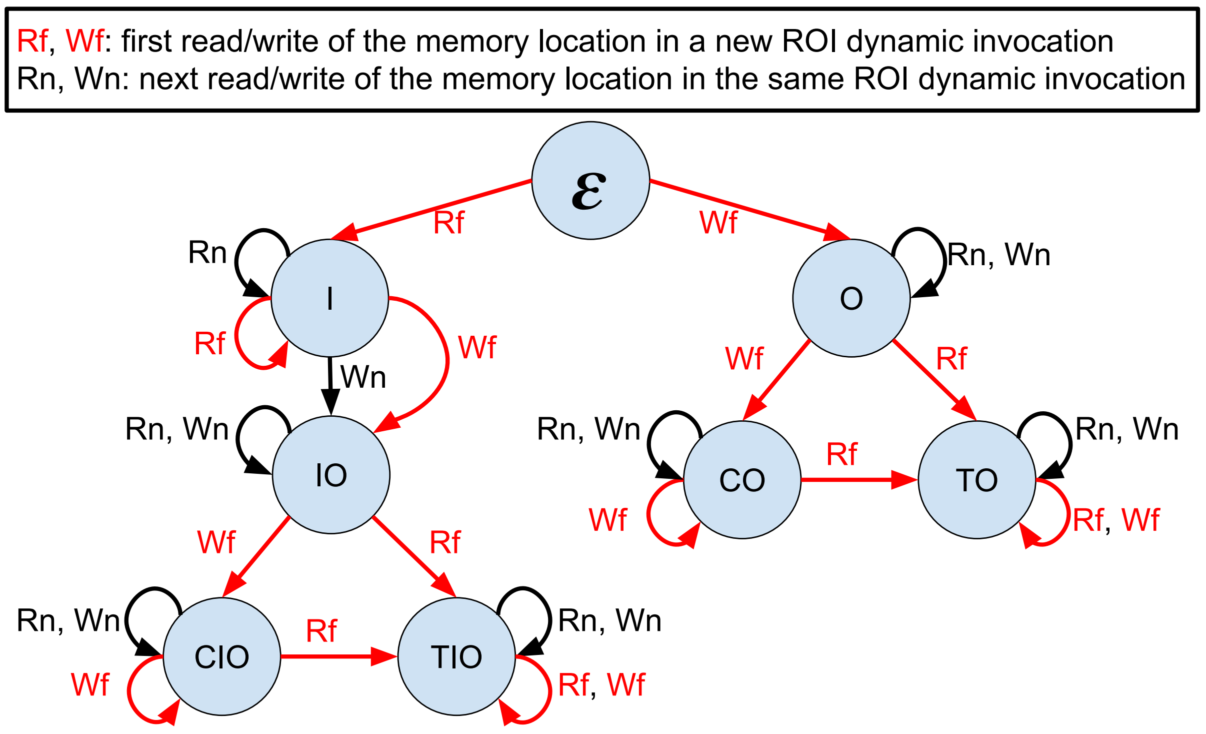

Rf, Wf: first read/write of the memory location in a new ROI dynamic invocation
Rn, Wn: next read/write of the memory location in the same ROI dynamic invocation
ε
Rf
Wf
I
Rn
Rf
Wn
Wf
O
Rn, Wn
Wf
Rf
IO
Rn, Wn
Wf
Rf
CO
Rn, Wn
Wf
Rf
TO
Rn, Wn
Rf, Wf
CIO
Rn, Wn
Wf
Rf
TIO
Rn, Wn
Rf, Wf

Figure 5.4. Each member of the program state can be classified in its corresponding computational spoor set(s) following a Finite State Automaton (FSA).

5.3.1. Computational Spoor Discovery with CARMOT

CARMOT classifies the program state members of an ROI independently of other ROIs. When a program state member (e.g., variable) is accessed within an ROI, CARMOT classifies it into the computational spoor sets (Section 5.2.1) of that ROI following the Finite State Automaton (FSA) shown in Figure 5.4. Each program state member has an instance of this FSA. All program state members start in the ε state. A program state member is added to the computational spoor of an ROI Z upon its first access within Z. Subsequent accesses of a member in Z might change its FSA's state for Z. At the end of a program's execution, the final state of a member's FSA for Z reflects the set (or sets) that the program state member belongs to with respect to

ROI *Z*. In more detail, if the terminal FSA state includes an `I`, `O`, `C`, and/or `T`, then the related program state member belongs to the Input, Output, Cloneable, and/or Transfer set respectively. Finally, note that a member can never be both in the Cloneable and Transfer sets ($C \cap T = \emptyset$).

Let us consider the loop in Listing 5.1 focusing on the innermost ROI and the program state member variable *y*. The first operation on *y* is a read, hence *y* transitions from the ε state of Figure 5.4 to the `I` state. While in `I`, a write to *y* in the same dynamic invocation of *Z* (W_n) causes *y* to transition into `IO`. Variable *y* will only depart from `IO` if a subsequent invocation of *Z* happens (i.e., R_f or W_f). In *Z*'s subsequent invocation of our example a read happens (R_f). At this point, *y* would transition from `IO` to `TIO`. There is no way out of the `TIO` state, so when the program finishes, CARMOT classifies *y* in the Transfer, Input, and Output sets.

The FSA only operates on reads and writes that happen within ROIs. This design decision enables CARMOT to avoid profiling code outside ROIs, but it also makes the assumption that program state members written in an ROI will be read outside the ROI, so they will be part of the Output set. This assumption is conservative and does not affect the correctness of the computational spoor.

CARMOT performs CSD by profiling a specific run of the target program, hence the information embedded in a computational spoor is bound to that specific execution. We envision CARMOT users will automatically perform CSD on a program multiple times to cover many program inputs, and combine the generated computational spoors. Combining computational spoors can be done through set union. For example, if program state member *p* is classified in the Input and Output sets in the first run, and in the Cloneable and Output sets in the second run, the computational spoor across runs classifies *p* is in the Cloneable, Input, and Output sets. The only exception to this union rule is when *p* is in the Cloneable set for one run and in the

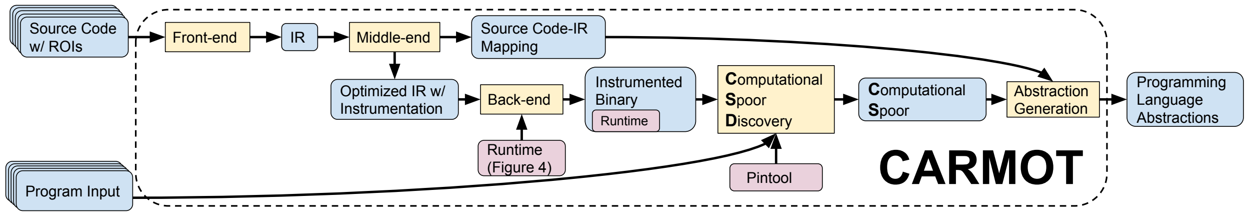


Figure 5.5. CARMOT produces the mapping between source-IR code and runs an instrumented binary to generate the computational spoor. The computational spoor generates the target abstraction information for the programmer at the source code level.

Transfer set for another run. In this case, the conservative answer is to classify *p* in the Transfer set. Currently and only for engineering reasons, CARMOT's users need to manually apply these rules to merge multiple computational spoors related to different runs.

CARMOT is a profiling tool that provides programmers recommendations and support for abstractions at the source code level for a specific program execution. The computed computational spoors and, by extension, the abstractions information generated by CARMOT from a computational spoor are also sound only for that execution. The process of generalizing abstractions to make them sound for all possible program inputs is outside the scope of this work and is currently left to CARMOT users.

5.3.2. CARMOT as a system

CARMOT implements the compilation flow of Figure 5.5, and it includes compiler and runtime optimizations to make CSD feasible even for large codebases. CARMOT's compiler (Section 5.3.3) generates a binary from C/C++ source files including code instrumentation. Complementarily, CARMOT's Pintool (Section 5.3.4) covers code that lack available source files, such as precompiled or closed-source libraries. CARMOT does so by loading its Pintool into memory when a program starts its execution. This Pintool is co-designed with CARMOT's compiler to decrease its overhead. The compiler injects code to notify the Pintool when the execution could jump into code that lacks available source files. The Pintool starts capturing information about memory changes only at that point.

CARMOT's runtime (Section 5.3.5) is embedded within the generated binary as a static library. The runtime processes the reads and writes provided both by the instrumentation generated by CARMOT's compiler and by its Pintool. This generates the computational spoor of each ROI specified by the CARMOT's pragma included in the program's source code (Listing 5.1). The computational spoors are then translated into programming language abstractions information that are chosen by CARMOT users and supported by the tool.

CARMOT includes CSD-specific code analyses and optimizations applied at code generation time to further reduce its overhead. For the same goal, CARMOT's runtime offloads its processing to a computational pipeline distributed between hardware cores.

5.3.3. Compiler

CARMOT's compiler instruments a program to compute the ROIs' computational spoors. The computational spoor of an ROI is computed at the IR level and then translated to the source code level. The benefit of performing CSD at the IR level rather at the binary level is two-fold. First, we can easily implement precise and effective specialized code analyses and optimizations. Second, the amount of instrumentation is considerably reduced compared to binary instrumentation where spilling of variables onto memory already occurred generating extra memory loads and stores. Next we describe the generation of the IR that preserves the mapping between IR and source code and the code analyses and optimizations designed specifically for CSD.

IR Generation. CARMOT's compiler uses its front-end to translate C/C++ source code files down to LLVM's IR. It does so by invoking `clang` without using compiler optimizations and with debugging symbols enabled to guarantee a predictable and reversible mapping between source code statements and the IR code. Unfortunately, instrumenting this unoptimized IR code leads to high overhead, making CSD infeasible for a large codebase. To solve this problem, CARMOT adds the following CSD-specific code analyses and optimizations.

CSD-Specific Code Analyses and Optimizations. The next CSD-specialized code optimizations rely on the FSA of Figure 5.4 to reduce run-time overhead making CSD feasible even for a large codebase. These optimizations are conservative and are applied only when they can guarantee they will not impact the correctness of the computational spoors generated.

Removing redundant instrumentation. The FSA of Figure 5.4 shows that transitions of program state members to a different state (and hence computational spoor set) happen only upon the first read or write (R_f, W_f) of a new dynamic invocation of an ROI. The only exception is a subsequent write (W_n) in the same ROI dynamic invocation when the program state member is in the `I` state. Following this observation, this optimization aims to instrument only the first read and write of a program state member and avoid instrumenting subsequent accesses that are proved to always access the same member.

To this end, a new intra-procedural data-flow analysis is added in CARMOT to identify where a program state member must have been accessed already since the beginning of an ROI. For this data-flow analysis, successors of basic blocks that leave an ROI are not followed during the data-flow value propagation as only instructions within an ROI need to be considered. For the same reason, predecessors outside an ROI are not considered. We do so by considering the entry point of an ROI as the entry point for our analysis. Elements in the GEN, IN, and OUT

sets are the variables and memory locations (i.e., program state members) of the target program. We use symbols to represent them as conventional memory alias analyses do and we rely on such analyses to identify such symbols. Given an instruction i that is either a load or a store, the sets for the data-flow analysis are defined as follows. The GEN set of i is the program state member a that a load is guaranteed to access or a store must write to ($GEN[i] = \{a\}$). The IN set of i is first initialized to be the union of all program state members, and then refined to be $IN[i] = \bigcap_{\forall p \in preds(i)} OUT[p]$, where p are the predecessors of i. The OUT set of i is initially empty, and then refined to be $OUT[i] = IN[i] \cup GEN[i]$.

This data-flow analysis runs until a fixed point is reached for each ROI. Elements in the IN set of an instruction i are the program state members that must have been accessed between the entry of the ROI and i. Hence, CARMOT reduces profiling overhead by avoiding instrumenting instructions i where the member accessed by i belongs to $IN[i]$.

Reducing instrumentation for only read members. The FSA in Figure 5.4 shows that program state members that are always only read will always be classified as Input in the computational spoor. Following this observation, we conclude that program state members that can be verified to be only read at compile time can be instrumented only once and still be correctly classified in the Input set.

Although an ROI is a general code region, we currently enable this optimization only for ROIs that wrap the body of a loop. We determine whether a program state member is only read by verifying that the corresponding load instruction that reads that program state member is loop invariant.

Call graph-based optimizations. This optimization aims to select functions that can be optimized with conventional transformations while preserving the aforementioned IR-to-source-code mapping needed for CSD. This optimization is based on the observation that if a function f cannot be in the callstack when any ROI starts, then f can be optimized with conventional optimizations (such as `-O3`) without breaking the IR-to-source-code mapping. This is because anything allocated in the stack by f will not be part of the computational spoor of any ROI. This holds even if f is invoked within an ROI, as its stack is freed before returning to its caller and therefore such stack variables and objects cannot be involved in any data dependences that cross the boundaries of an ROI. Therefore, only objects that are heap allocated by f need to be tracked, which are preserved by the current set of optimizations included in `-O3` of `clang`.

To perform this optimization, CARMOT identifies the functions that cannot be in the callstack when any ROI starts. To do so, CARMOT computes the complete callgraph of a program (i.e., a callgraph where the lack of an edge (f_i, f_j) means f_i cannot invoke f_j). Unfortunately, the callgraph provided by LLVM is not complete; the lack of an edge (f_i, f_j) could also mean that f_i invokes f_j via an indirect call through a pointer. To generate the complete callgraph of a program, CARMOT computes the program dependence graph (PDG) to automatically discover the possible callees to which a pointer could refer. CARMOT uses the same memory alias analyses used by the previous optimization.

Armed with the complete callgraph, CARMOT identifies the set of functions that can be optimized. For every ROI, CARMOT takes the function f where the ROI belongs to and traverses the edges of the callgraph backwards from f and tags all functions reached (including f). All functions in the program that are not tagged are optimized by CARMOT invoking the `-O3` optimizations of `clang`.

CARMOT uses the call graph to also reduce Pin instrumentation by enabling the Pintool only when it cannot guarantee that a call will not jump to precompiled code.

Reducing callstack computation. CARMOT needs to record the callstack of every allocation of every variable or memory location, which are part of the program state members. In a typical function, many variables and memory locations are allocated (please recall functions that might be executed before an ROI are not optimized and therefore all variables are allocated in the stack at the IR level). In a naive implementation, every time an allocation of a program state member occurs the callstack must be computed and assigned to that program state member. However, allocations made within the same invocation of a function share the same callstack. To avoid recomputing the callstack for each program state member allocation within a function, CARMOT computes the callstack only once at the beginning of the function. The instrumentation that documents allocations can now collectively share the computed callstack instead of producing redundant callstack records that are clones of one another.

5.3.4. Pin instrumentation

When the target program includes code outside the available sources (e.g. precompiled libraries), it is impossible to track all program state information with a purely compiler-based approach. However, the activity must be tracked in order for the computational spoor to be correct and complete. CARMOT uses dynamic binary instrumentation to perform this tracking. Our mechanism is a Pintool that builds upon the Pinatrace memory access tracing tool from Intel [**9**]. The key challenge is to efficiently communicate between the Pintool and the compiler/runtime environment that comprises the rest of CARMOT.

To control the Pintool, the CARMOT compiler inserts calls to our `start_tracking()` and `stop_tracking()` functions before and after black-box code. Pin intercepts calls to `start_tracking()` and invokes our Pintool. The Pintool then tracks memory allocations/deallocations and reads/writes until `stop_tracking()` is called. This is a heavyweight operation, but it only occurs when the target program is executing black-box code. Outside of black-box execution regions, no additional overhead is incurred.

The Pintool must transfer the results of tracking such memory operations to the CARMOT runtime, which is challenging because the runtime cannot access Pin's memory. To this end, we embed a custom interface in the Pintool called `get_tracking(void *dest, size_t n)`, which uses `PIN_SafeCopy()` to copy its current trace to `dest`. The Pintool reports tracking information in batches through `dest` of this interface.

5.3.5. Runtime

CARMOT's runtime, extended from TEXAS, processes the accesses of program state members provided by either compiler-injected instrumentation or the Pintool. We need to process this information at run-time because of the high amount of collected data, which makes a storing-based approach infeasible.

Data structures and requests. The primary structures the runtime builds are the Active State Member Table (ASMT) and the ROIs' computational spoors. The ASMT captures metadata about *active* state members (program variables and memory locations) such as the state member's callstack and their size in bytes. The runtime generates a computational spoor by enacting the FSA (Section 5.3.1) upon program state members for each specified ROI. These data structures are built by the following requests from the instrumentation code:

`create_state_member()`: This request declares the creation of a new state member. This results from variable declarations and stack/heap/.bss allocations. Once created, the state member is considered *active*, and inserted into the ASMT. Instructions like `malloc(4*sizeof(int))` conceptually create 4 state members; however, the runtime aggregates the state members into a single member with length of `4*sizeof(int)`. In this example, state member set membership is determined as offsets into the allocation rather than solely the memory location, so as to give contextually richer information to the user.

`remove_state_member()`: This request declares that an active state member is no longer active. This results in the state member(s) being removed from the ASMT. Note that this does not imply that state member information is deleted, it simply means that the state member will no longer contribute to building ROIs' computational spoors. An example of where this instrumentation would be used is a `free()` instruction. Upon calling `free()` on an allocation, the allocation will no longer be valid; therefore, it no longer can (correctly) contribute to any computational spoor.

`state_member_access()`: This request declares an access (read or write) of a state member. This results in the runtime taking the access and potentially updating the state member's set membership in an ROI-bound computational spoor.

`track_escape()`: This request declares a pointer is stored within a member (e.g., memory location). We say the pointer escaped. This results in tracking the escaped pointers between active state members, which in turn assists in forming an interconnected graph of all state members of a program.

`ROI_begin()/ROI_end()`: This request declares when to start and stop processing

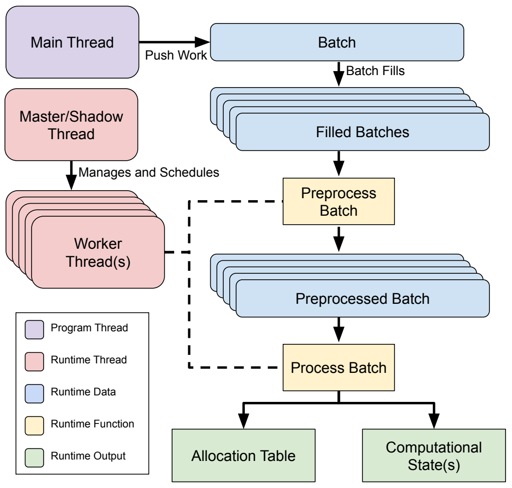


Figure 5.6. The runtime utilizes batching, shadow profiling, and pipeline parallelism to efficiently perform CSD.

`state_member_access()` instrumentation for an ROI's computational spoor. This instrumentation is determined by the user specified *#pragma carmot roi* scoping (Listing 5.1). When within the scope of an ROI, `state_member_access()` instrumentation contributes to the developing set membership of state members in the corresponding ROI-bound computational spoor. For any run time `ROI_begin()` call there must be a matching `ROI_end()` run time call. This pairing constitutes an *invocation* of an ROI.

System and operation. Figure 5.6 shows the components and the processing flow of the runtime. The *Main Thread* runs in main (i.e., the target program). The compiler-injected instrumentation or the Pintool push requests described in Section 5.3.5 into a *batch*. Once a batch fills, it is pushed into an ordered queue of filled batches, and the instrumentation begins filling a new batch. The *Master/Shadow Thread* schedules filled batches for preprocessing by *Worker Threads*. Each preprocessed batch is then added to a second ordered queue by the Master/Shadow Thread for final processing. The processing respects the ordering of the original requests generated by the instrumentation. The results of processed batches are updates to the

ASMT and computational spoors. The batches are processed following a parallel pipeline:

Preprocessing batches. The preprocessing stage processes the instrumentation from Section 5.3.5 to build the ROIs' computational spoors for all state members. It does so by implementing the FSA in Figure 5.4 on active state driven by calls to `state_member_access()`. This is implemented per ROI's computation spoor via unordered maps in which the memory address is the key and the FSA set membership is the value. Once the batch has been preprocessed, it is then queued to the next pipeline stage and the next batch can be preprocessed.

Processing batches. The next stage of the processing flow is the processing stage, which adds contextual information to the ROIs' computational spoors. This stage is responsible for connecting metadata to state members. This includes the callstack, escaped pointers, origin of the state member (heap, stack, global), and uses (reads and writes of program state members). This metadata is attached/associated to program state members throughout the execution of the program.

5.4. Evaluating CARMOT

To show the effectiveness of CARMOT, we evaluate 15 benchmarks from the SPEC CPU 2017, NAS [**24**], and PARSEC (version 3.0) [**34**] benchmark suites. We chose these benchmarks because they already use, or are well suited for, the abstractions that CARMOT currently supports. When evaluating the performance benefits of CARMOT we used the "reference", "class C", and "native" inputs, respectively. When evaluating the overhead of CARMOT we used the "test", "class A", and "simsmall" inputs. The difference in inputs for performance and overhead results reflects how we see CARMOT being used. The smaller, representative inputs follow CARMOT being used to determine the computational spoor of the program at development time. Larger, production level inputs are used for the performance results as this is indicative of the actual speedup that programs developed with CARMOT can attain.

5.4.1. Experimental setup

Our evaluation is done on a dual socket server with two Intel Xeon Silver 4116 CPU running at 2.1GHz. Each processor has 12 cores with 2-way hyper-threading, and 16.5 MB of last level cache. The cores are supported by 125 GB of main memory at 2400 MHz. The OS is Red Hat Enterprise Linux 8.2 (kernel 4.18.0-193.6.3). CARMOT is built on top of LLVM 9.0.0 [**106**], Pin 3.13 [**112**], and NOELLE 9.3 [**116**]. The baseline for both performance and overhead evaluation we show is the sequential version of each benchmark, compiled with "clang -O3 -march=native".

5.4.2. OpenMP use case

OpenMP is widely used by developers today, however its full potential is often missed from a lack of knowledge about the target program. We show that, using its computational spoor, CARMOT is able to automatically generate *#pragma omp parallel for*, *#pragma omp critical*, *#pragma omp ordered*, and *#pragma omp task* annotations. CARMOT can be used as a tool by developers to verify the correctness and improve the performance of existing pragmas for a specific program execution. For example, an existing pragma declaring a variable as private even though it is only read in the abstraction's region. In this case, CARMOT would suggest declaring that variable as shared to remove a useless extra copy. Furthermore, CARMOT can be used to parallelize programs by generating brand new pragmas.

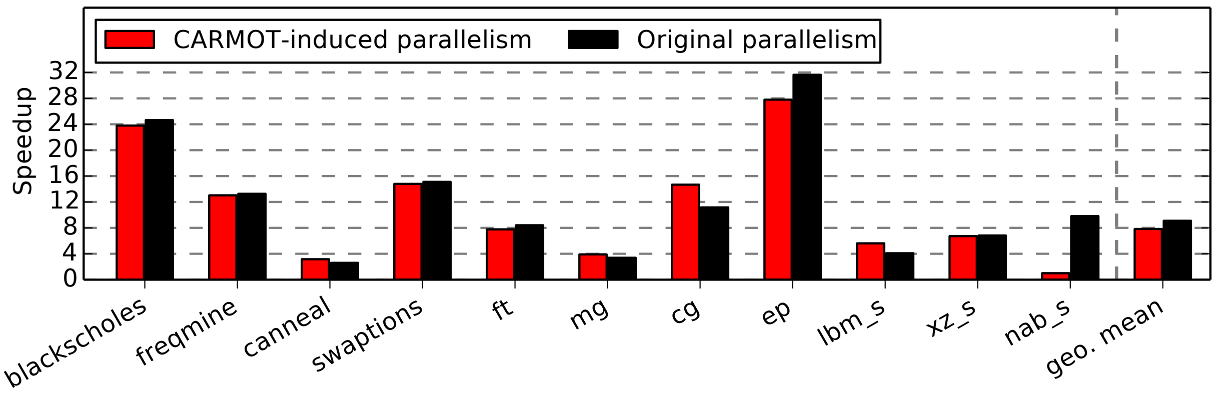


Figure 5.7. CARMOT-generated OpenMP pragmas achieve the same speedup of the original program parallelism manually implemented by a programmer.

Figure 5.7 shows the speedup benefits of automatic CARMOT-generated pragmas (either verified pragmas or brand new ones) versus the original (manually extracted) parallelism (either through omp pragmas or pthread) for each benchmark we consider. This data shows that with CARMOT-generated pragmas, we are able to achieve speedups that are as good or better than pragmas implemented manually by a programmer. For benchmarks like `canneal` and `swaptions`, where the only original source of parallelism comes from pthreads, the new pragmas generated by CARMOT match the performance of the labor-intensive pthreads parallelism. The only exceptions are `ep` and `nab` for which CARMOT was unable to extract all parallelism potential. In both cases the main source of parallelism in these benchmarks comes from general OpenMP *#pragma omp parallel* sections that include synchronization mechanisms such as *#pragma omp barrier* or *#pragma omp master* that are abstractions currently not supported by CARMOT.

When designing new development tools, striking a balance between effectiveness and feasibility is paramount. The feasibility of CARMOT as a tool is measured by the computational overhead required to perform CSD. Figure 5.8 shows the computational overhead of CARMOT

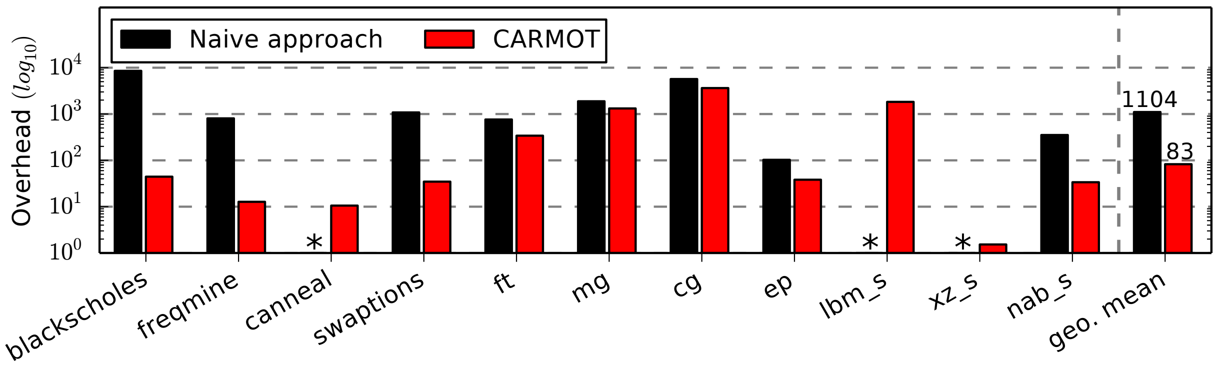


Figure 5.8. The CARMOT overhead to generate OpenMP pragma information is one order of magnitude less than a naive approach without CSD-specific optimizations.

when automatically generating OpenMP pragmas information. We compare the CARMOT overhead with a naive approach that does not employ any CSD-specific optimization, but can still generate a correct computational spoor. CARMOT outperforms the naive approach by lowering the overhead of performing CSD by one order of magnitude. In some cases the execution of the naive approach required an excessive amount of memory and did not complete, we marked the missing data with $*$. To showcase the power of CSD-specific optimizations, Figure 5.9 shows the impact of the optimizations described in Section 5.3.3. For the benchmarks where the naive approach finished successfully, we show in percentage the breakdown of the delta between the black and red bars of Figure 5.8 for every CARMOT optimization. The reduction of Pin instrumentation and the callgraph-based optimization both enabled by the complete callgraph of NOELLE have the highest impact. We observed similar results in the other use cases, for this reason we do not report their overhead reduction breakdown.

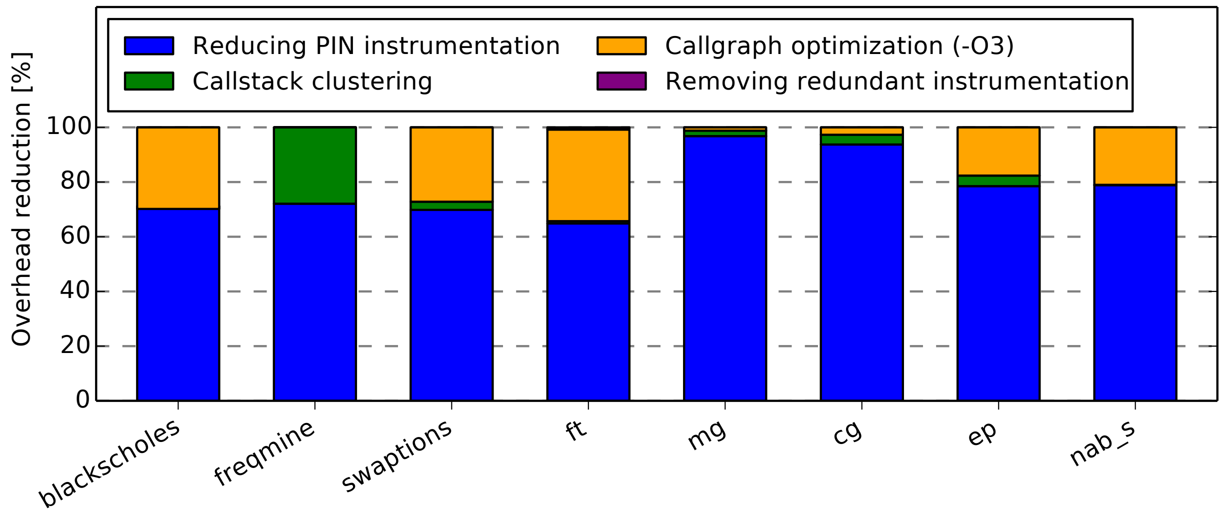
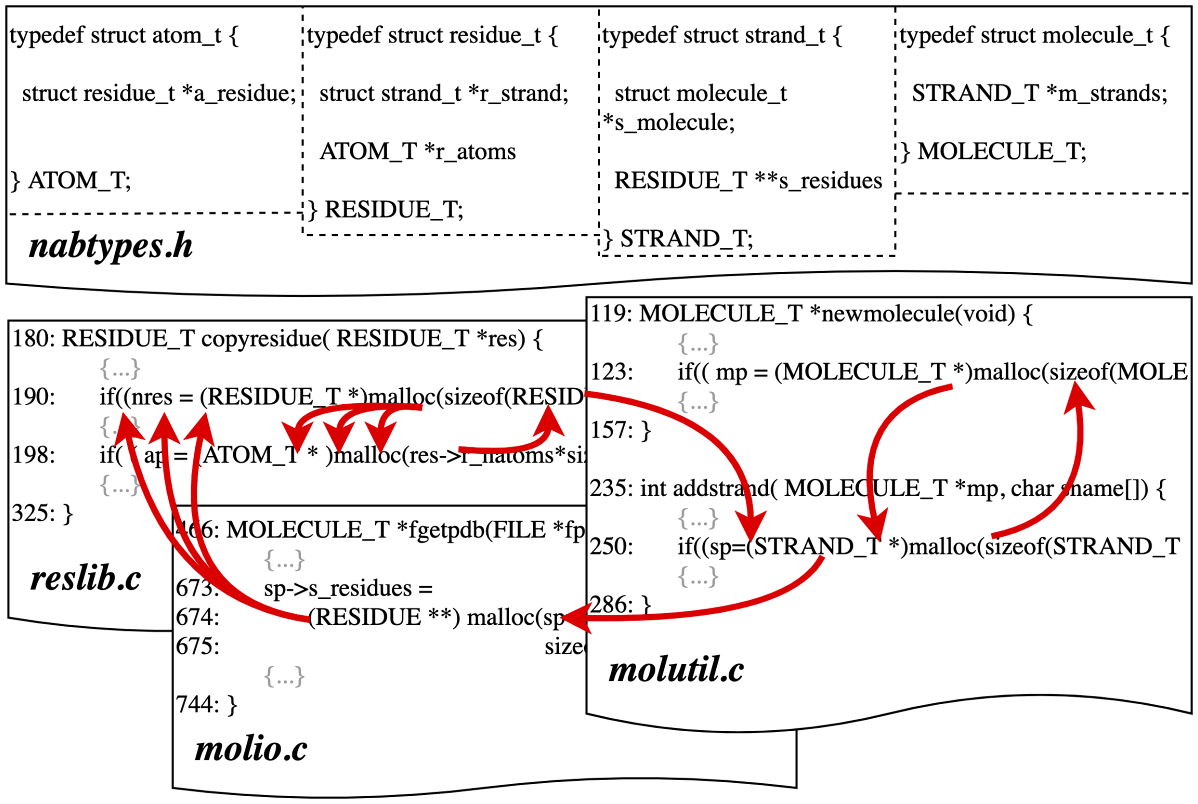


Figure 5.9. Overhead reduction of Figure 5.8 characterized per CARMOT optimization.

5.4.3. Smart pointers use case

Here we show the generality of CARMOT on a use case unrelated to parallelization. Here CARMOT is used to identify reference cycles related to an ROI while gradually porting an application to use C++ smart pointers. This use case shows that CARMOT can also be used to identify reference cycles in application which already use smart pointers. In addition to identifying the allocations involved in the reference cycle, CARMOT can also determines which allocation to port to a *weak_ptr* to break the cycle.

Figure 5.10 shows an example of a reference cycle that CARMOT identified in the `nab` benchmark. This cycle spans across several different files, functions, and data structures and demonstrates the complexity of porting an existing application to use smart pointers correctly. CARMOT identifies the allocations performed in the ROI for the program that we want to port to use smart pointers and can determine which allocation should be declared as *weak_ptr* to break the reference cycle. We measure the benefit that utilizing smart pointers for this reference



```
typedef struct atom_t {
  struct residue_t *a_residue;
} ATOM_T;
```

```
typedef struct residue_t {
  struct strand_t *r_strand;
  ATOM_T *r_atoms
} RESIDUE_T;
```

```
typedef struct strand_t {
  struct molecule_t
*s_molecule;
  RESIDUE_T **s_residues
} STRAND_T;
```

```
typedef struct molecule_t {
  STRAND_T *m_strands;
} MOLECULE_T;
```

```
RESIDUE_T copyresidue( RESIDUE_T *res) {
        {...}
        if((nres = (RESIDUE_T *)malloc(sizeof(RESID
        {...}
        if( ap = (ATOM_T * )malloc(res->r_natoms*si
        {...}
}
```

```
MOLECULE_T *fgetpdb(FILE *fp
        {...}
        sp->s_residues =
          (RESIDUE **) malloc(sp
                              size
        {...}
}
```

```
MOLECULE_T *newmolecule(void) {
        {...}
        if(( mp = (MOLECULE_T *)malloc(sizeof(MOLE
        {...}
}

int addstrand( MOLECULE_T *mp, char sname[]) {
        {...}
        if((sp=(STRAND_T *)malloc(sizeof(STRAND_T
        {...}
}
```

Figure 5.10. CARMOT-identified reference cycle across files, functions, and data structure in the nab benchmark.

cycle would generate for the benchmark. After correcting a naiveness in the original nab code which over allocates memory, we measure the total bytes leaked by the application as 230,537 bytes. The total bytes leaked by the application that would be realized by correctly porting this reference cycle to smart pointer is reduced to 127,633, a reduction of 44.6%.

Figure 5.11 shows the overhead of CARMOT when finding reference cycles. In this use case CARMOT only needs to track allocations of program state members and the reference graph of such allocations. For this reason, CARMOT's overhead is two orders of magnitude smaller

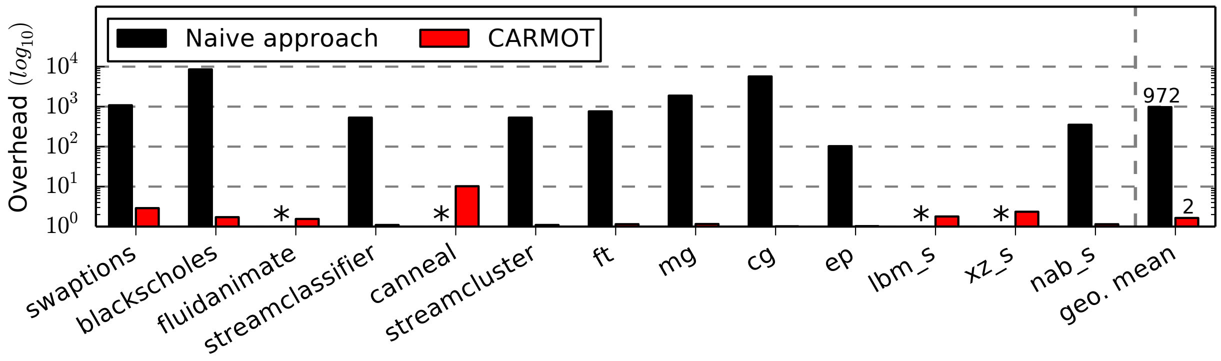


Figure 5.11. The CARMOT overhead for identifying reference cycles is two orders of magnitude less than a naive approach.

than a naive approach that lacks CSD-specific optimizations and the concept of computational spoor and its components.

5.4.4. STATS use case

Here we show that CARMOT can be used to build the Input, Output, and State classes required by the STATS abstraction. CARMOT is able to accurately generate these classes, such that they match those manually implemented by the authors of STATS. Furthermore, CARMOT was able to identify some misclassifications of program state members made by the authors of STATS. While these misclassifications have no impact on correctness, they lead to extra unnecessary copies of variables. In this case, the misclassification does not lead to a noticeable speedup. However, CARMOT's ability to outperform the manual and labor-intensive classification lends to its usefulness as a tool for abstractions that are difficult for developers to use correctly.

Figure 5.12 shows CARMOT's overhead for classifying program state members into STATS's Input, Output, and State classes. We can see that the CARMOT overhead is one order of magnitude lower than a naive approach. This is due to the STATS abstraction not requiring the

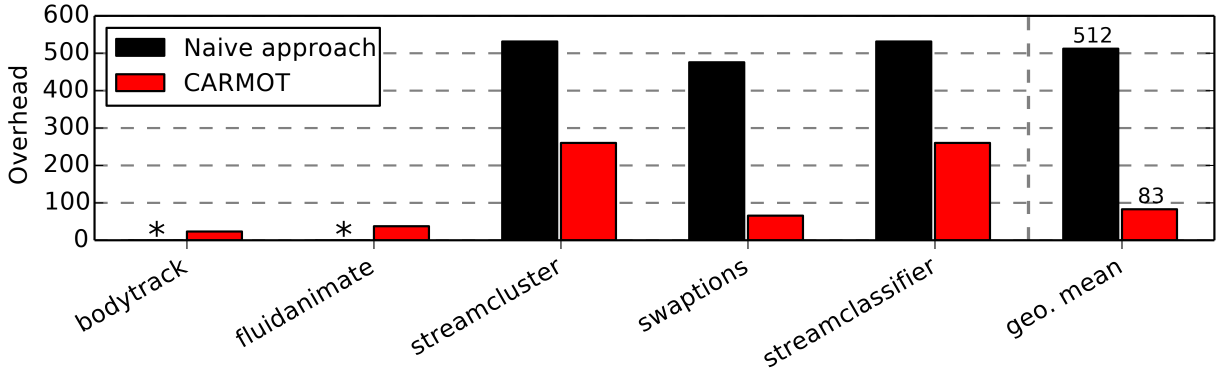


Figure 5.12. The CARMOT overhead to generate the STATS abstraction is one order of magnitude less than a naive approach.

tracking of all **Use-callstacks**, a very costly operation, and the CSD-specific optimizations of CARMOT.

5.4.5. How CARMOT uses TEXAS

The CARMOT project in its entirety began as an extension to the TEXAS runtime. The runtime of CARMOT uses variants of TEXAS's data structures and abstractions to accomplish its goals of building the computational spoor.

The *Active State Member Table, ROIs, and State Members* are variants of the Allocation Map, State, and Allocation abstractions respectively. Additionally, core compiler-based optimizations for CSD are derived from the optimizations made with the protection-based optimizations from TEXAS/CARAT.

CARMOT does add in a significant amount of novelty outside of the TEXAS runtime as well. The concept of a Computational Spoor and its application found in CARMOT are independent ideas from TEXAS. TEXAS provided the ease and opportunity to enable Computational Spoor Discovery in a feasible manner.

5.5. Contributors

This project was accomplished through a large effort beyond just myself for many components of this work.

My contributions primarily involved (but was not limited to): the extension of the TEXAS runtime to accommodate tracking and profiling programs, the creation of the use case of cycle detection in C++ smart pointers, aiding in the creation of the four Computational Spoor sets, the creation of finite state automata (FSA), and compiler optimizations based on the FSA.

Enrico Deiana[3] was primarily involved in creation of the all other CARMOT use cases, aiding in the creation of the four Computational Spoor sets, the overarching story and motivation of CARMOT, the creation and modification of compiler passes/optimizations, creation of a wrapping interface around TEXAS, and evaluation of CARMOT.

Mike Wilkins was primarily involved in creation of the custom binary instrumentation tool based on Pin to extend TEXAS into pre-compiled libraries.

Brian Homerding, Tommy McMichen, and Katarzyna Dunajewski were primarily involved in helping further developing CARMOT's use cases as well as evaluation.

Peter Dinda, Nikos Hardavellas, and Simone Campanoni offered guidance as well as helping craft the overarching story and motivation for CARMOT.

[3]Enrico Deiana will be the first author of the eventual publication of CARMOT. An attribution well deserved and earned.

CHAPTER 6

Manufacturing Locality: Improving Locality by Matching Memory Layout to Program Access Patterns

TEXAS has been used so far to replace the functionality of the hardware[1] or in the observation of programs[2]. What TEXAS has not yet been used for is the transformation, or X'formation, of a program's memory. Memory management for unmanaged languages, like C/C++, requires expertise from the programmer to fully utilize the memory hierarchy for performance. However, even with the assumption of a competent programmer, the access pattern of a dynamic program may not always reflect the static layout that the programmer has provided. The core issue that programmers face is that unmanaged languages afford the programmers the ability to fine-tune the layout and partitions of Allocations with minimal overhead, but suffer from not being able to dynamically make changes to this arrangement during execution. On the other hand, programmers *do* have this ability when using managed languages[3], but the costly overhead of managed languages presents other problems. If there was a low overhead mechanism that could address this disparity, then there would be great potential for performance gains.

To address this, we introduce Manufacturing Locality. Manufacturing Locality extends the TEXAS runtime to enable a program to dynamically repack memory in order to modify the layout of memory to better match its access pattern, improving the spatial locality of a program.

[1]CARAT and CARAT CAKE
[2]CARMOT
[3]Managed languages can match access pattern to layout at a very coarse granularity through mechanisms like stop-and-copy garbage collection.

Running on TEXAS, Manufacturing Locality creates a Connection Map of a program and performs transformations based on defragmenting, layout/access pattern matching, and BFS/DFS search patterns. Manufacturing Locality shows that when pairing TEXAS with programs, performance gains can be had for free on existing computing systems.

6.1. How to Manufacture Locality

Figure 6.1. Tree Example for Manufacturing Locality

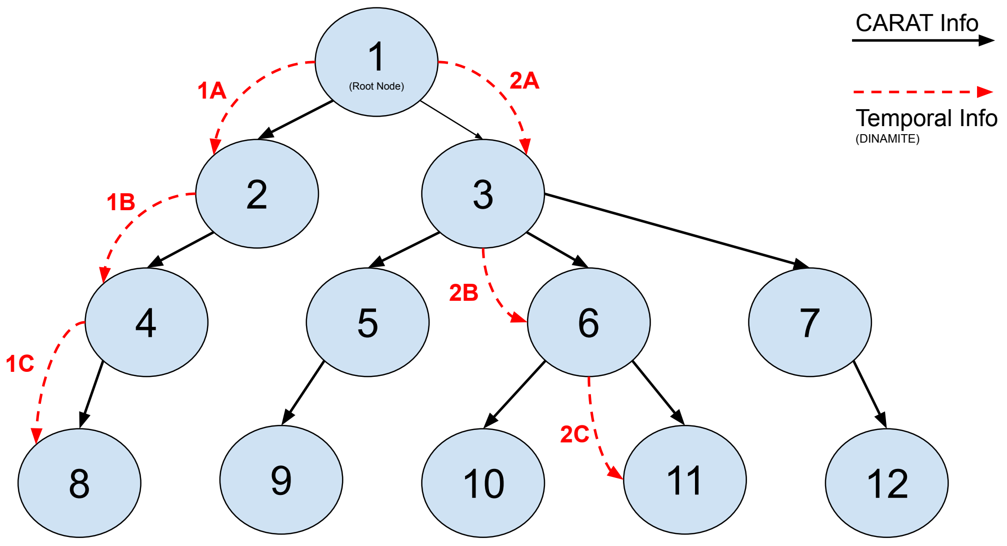


Figure 6.1 shows an example of what information would be available to TEXAS to aid in Manufacturing Locality. The example focuses on observing the connectivity of a tree as well as its temporal access pattern. The black solid lines indicate escapes from one node to another (in this case a child node will have an escaping reference inside of their parent). The red dashed lines indicate the temporal access pattern observed for two separate iterations through the tree.

Figure 6.2. Example of Defragmentation Repacking (6.2.1.1) for Figure 6.1

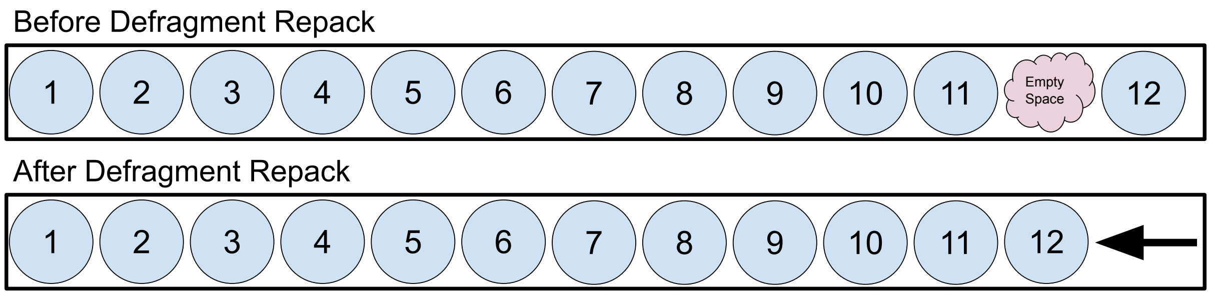
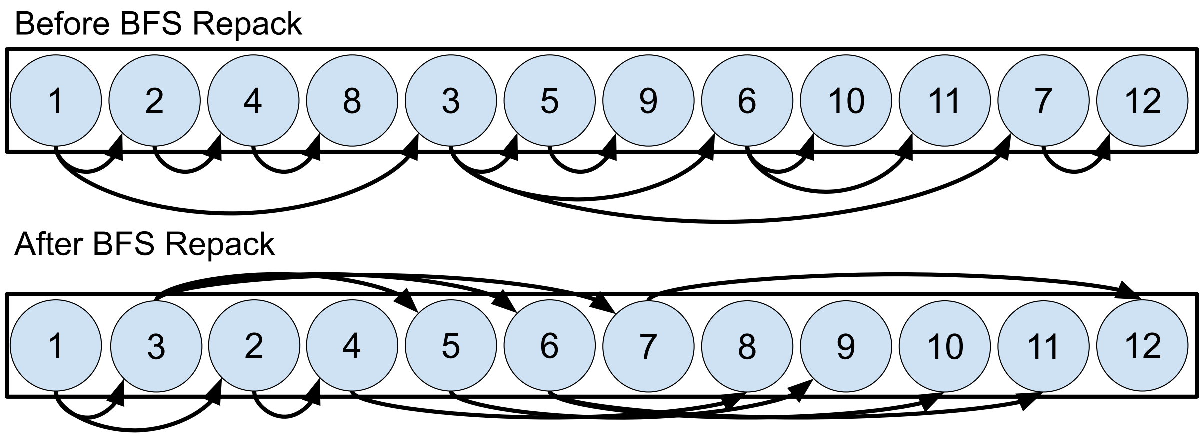


Figure 6.3. Example of BFS Repacking (6.2.1.2) for Figure 6.1



The access pattern $1 \rightarrow 2 \rightarrow 4 \rightarrow 8$ is contained in the edges 1A-1C and the access pattern $1 \rightarrow 3 \rightarrow 6 \rightarrow 11$ is contained in the edges 2A-2C.

To manufacture locality, we can take the information given by TEXAS to begin assigning weights to the edges. For TEXAS, we can assign a weight based on how Allocations are connected via Escapes. We can also assign a weight equal to the number of times an Allocation is accessed temporally after another Allocation. Now, when we perform a topological sort and relocation of a graph, whenever we reach a point where there is more than one edge to take, we can use various heuristics to determine the order of repacking.

Figure 6.4. Example of DFS Repacking (6.2.1.2) for Figure 6.1

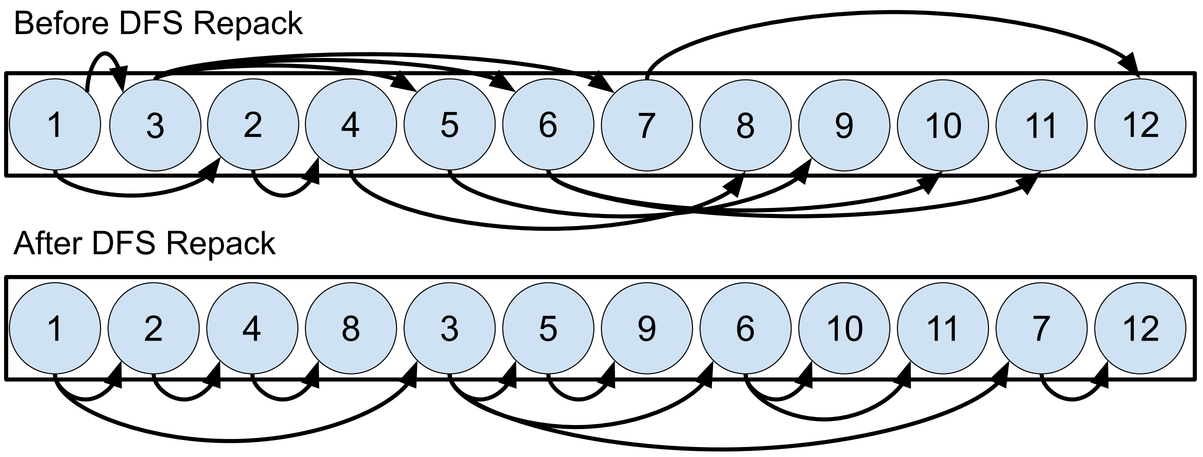
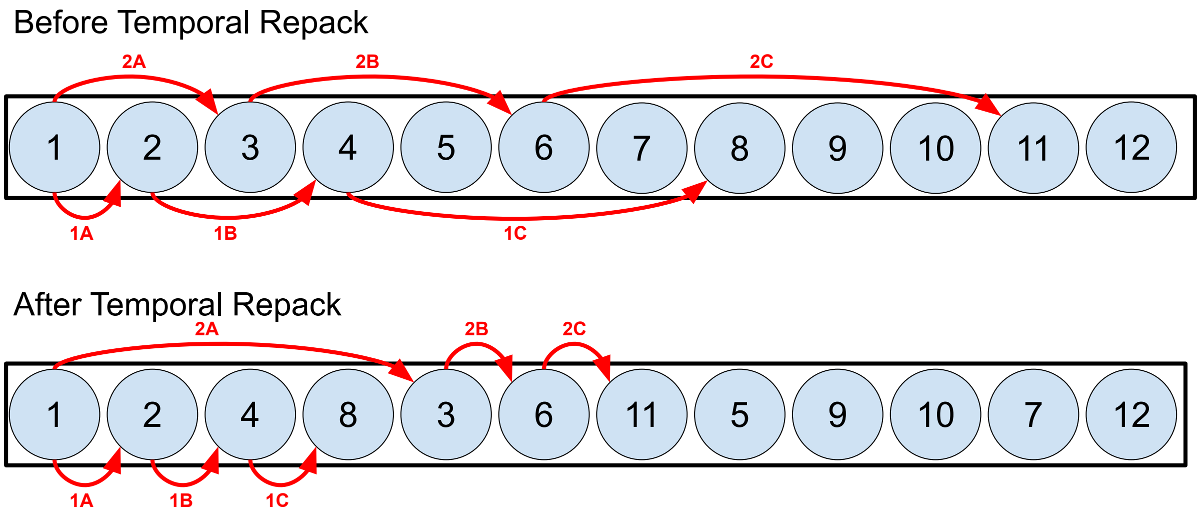


Figure 6.5. Example of Temporal Repacking (6.2.1.3) for Figure 6.1



Not shown in Figure 6.1 (due to simplification) are additional ways to view temporal access patterns. Instead of considering only the edges between two nodes, one could envision considering the paths from one node to another such as the path from the root node '1' to the leaf node '8' ($1 \rightarrow 2 \rightarrow 4 \rightarrow 8$). There may be some instances where considering this path as a whole may be more beneficial for a program instead of just considering individual edges, such as if the entire path fits within a single cache line or a page. A step beyond paths is to try and extract n-grams from the graph, where n is equal to the number of nodes that fit in either a cache line or a page. This can be used later to help determine how to repack the data.

There are various obstacles to implementing the idea of repacking memory. First, there is the question of how many allocations we should actually move. Obviously, if an allocation only ever used once (or very sparingly) in a program, there is no point in wasting cycles on moving the data to be more cache friendly. On the other hand, if a data structure (like the tree) is constantly being accessed, then it is worth it to move the data. In order to make use of Manufacturing Locality, we need to find common structures in code that provide opportunities to transform memory to improve its locality. A natural control structure that allows this is a loop. A loop typically operates on a data structure(s) for many iterations. By using TEXAS's compiler to transform the loops into TEXAS loops, we can instrument selected iterations with instructions to dynamically build Figure 6.1. The more iterations peeled/split, the more informative rich version will become.

Figure 6.6 shows how TEXAS's compilers transform relevant loops into TEXAS Loops and then instruments them to accommodate Manufacturing Locality. After being converted to a TEXAS Loop, the Manufacturing Locality compiler pass instruments the TEXAS Loop with Use Tracking. Once the TEXAS Loop has run for enough iterations to get a satisfactory partial graph of Figure 6.1, then the 'Repack Memory' block can be started which will use the Figure 6.1 graph to perform a heuristic-based repacking of memory.

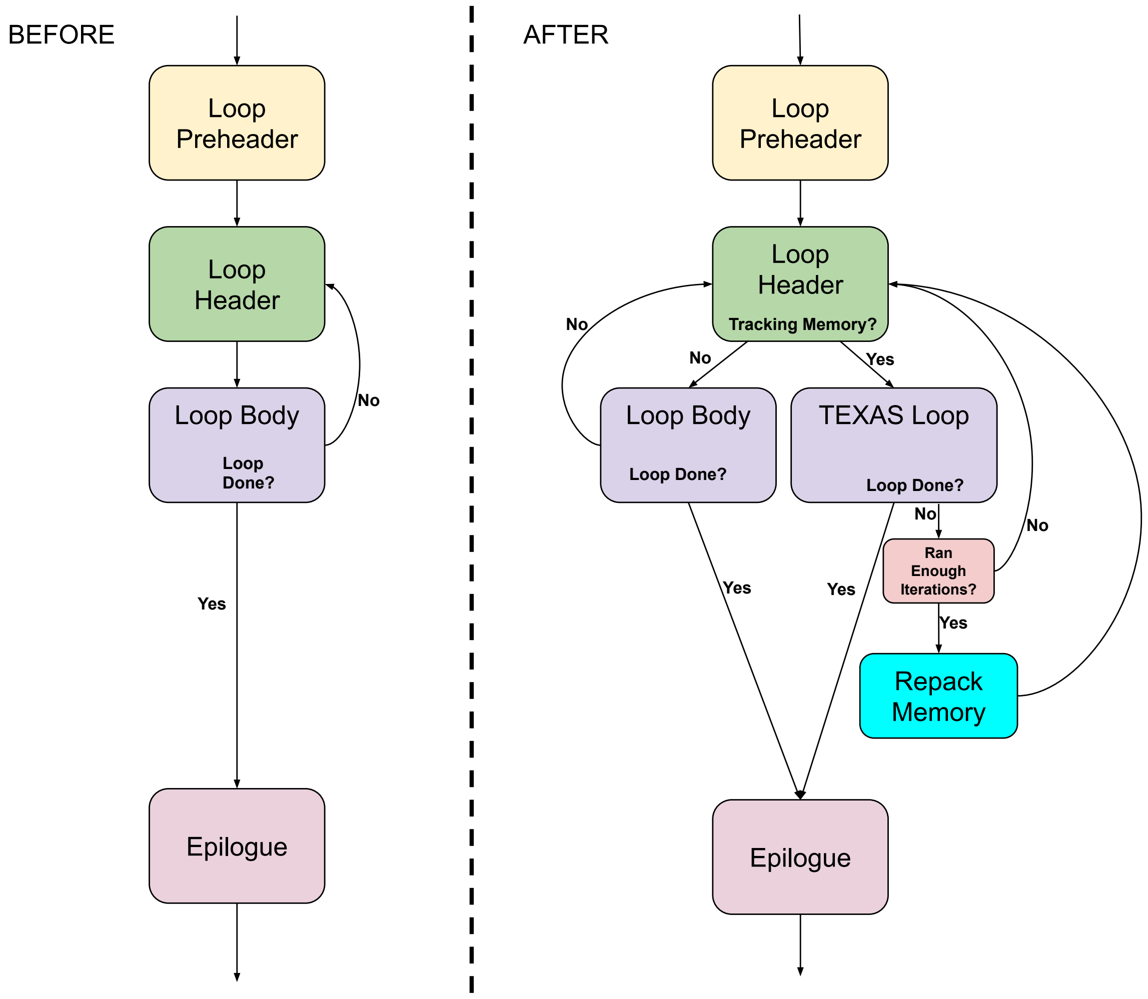


Figure 6.6. Example of transforming a loop into a TEXAS Loop.

6.1.1. Manufacturing Locality without TEXAS Loops

TEXAS can also attempt to manufacture locality without requiring the TEXAS Loop compiler transformation. Instead of using Use Tracking to glean access pattern information, TEXAS can make transformations based only on the layout and connectivity of Allocations[4] given in the

[4]Connectivity of Allocations can be determined by seeing which Allocations contain Escapes to other Allocations.

Connection Map. Using the Connection Map, different repackings can be performed heuristically. These methods are discussed in Section 6.2.1.

6.2. Adapting TEXAS

To accommodate Manufacturing Locality with TEXAS, a few modifications were made to the base form of TEXAS were made. The changes primarily involve being able to contextualize a graph of all the Allocations in the program, followed by support for performing transformations on the Allocation graph; therefore, moving the Allocations around.

6.2.1. X'forming with the Allocation Graph

With TEXAS's Connection Map, we can now manufacture locality among the Allocations in running programs by making transformations to the Connection Map's Allocations with various static and dynamic heuristics. These heuristics vary in the intensity of computation and improve locality amongst the Allocations in the graph in various ways. Visual examples of heuristics that perform repacks on the tree in Figure 6.1 are shown in Figures 6.2-6.5.

6.2.1.1. Defragmentation. The first heuristic that can be implemented is a simple defragmentation of Allocations. This heuristic simply takes all the Allocations in the graph, creates a new SuperMalloc, and then places all the Allocations into this new SuperMalloc packed together. The intention of this repacking method is two-fold. The first intention is to simply perform a defragmentation, which supports large memory use.

The second intention is focused on Manufacturing Locality through defragmentation by repacking active Allocations together, slightly improving the spatial locality amongst them. The order of repacking the Allocations for this heuristic is simply iterating through the Allocations in the Connection Map with no guiding heuristic in mind.

An example of this repacking method can be seen in Figure 6.2.

6.2.1.2. BFS/DFS. The second heuristic that can be implemented on the Allocations in the graph is a Breadth-First/Depth-First repack. This heuristic takes all the Allocations in the graph and then repacks them by traversing the graph in a BFS/DFS manner, packing Allocations that share traversed edges in the graph. Also note that the selection of the starting node(s) in the graph can be determined in multiple ways:

- Randomly. A random Allocation can be selected as a starting node.
- Incoming/Outgoing Edge(s). The starting Allocation can be chosen based on how many incoming/outgoing edges it has in the graph. Choosing an Allocation with no incoming edges suggests that it could be a root node in a data structure.
- Virtual Address. The starting Allocation can be chosen based on having the smallest or largest virtual address. A smaller virtual address may indicate an earlier allocated Allocation.

Another note about this repacking is that it also performs defragmentation. An example of these repacking methods can be seen in Figures 6.3-6.4.

6.2.1.3. Temporal. The third heuristic that can be implemented on the Allocations in the graph is a temporal repacking. This repacking selects the most accessed Allocation (*current*) in the graph and selects which Allocation to repack next to it based on which Allocation (*next*) is most accessed after the *current* Allocation. Once the *next* Allocation is selected, it becomes the *current* Allocation and iterates. This process in Figure 6.1 would start with Node 1 being *current* and Node 2 or 3 being *next* depending on the weights of the red edges (1A and 2A).

This temporal repacking will pack Allocations together based on the order of the access pattern given to it.

An example of this repacking method can be seen in Figure 6.5.

6.2.1.4. Hottest Allocation. An alternative approach to utilizing the weighted temporal red edges in the graph can be to simply repack from hottest to coldest Allocation without regard to position on the graph. This repacking heuristic will use the same access pattern but instead apply weights to the nodes. When repacking, the order of repacking will go from the node with the highest weight to the node with the lowest weight. This repacking will pack the hottest Allocations together, regardless of proximity within the access pattern.

6.2.2. Compilation Pipeline

To ensure the ability to manufacture locality in programs using one of the heuristics from section 6.2.1, there are alterations made to the compilation pipeline of typical programs to enable instrumentation/injection of TEXAS code.

6.2.2.1. Normalization. Upon obtaining the single bitcode file, a preliminary transformation is performed on it to *normalize* the code for further analysis and transformation. This normalization is performed by NOELLE [**116**], an open source compilation enhancing tool, and transforms the given bitcode into a form more amenable to optimizations and analysis. An example of one of these normalizations is the conversion of loops in a program into LCSSA form, which looks like the “Before” side of Figure 6.6.

6.2.2.2. Temporal Tracking. In addition to instrumenting the code for tracking Allocations and Escapes, a new pass is inserted to track temporal use of memory within loops. This pass performs the transformation on the loops shown in Figure 6.1. In the cloned loop body, every load and store instruction is instrumented to build a temporal memory access trace that, in turn, can be used by the runtime to perform the temporal transform on the graph described in Section 6.2.1.3.

6.2.2.3. Libraries. Libraries present a unique problem for TEXAS to address. The first problem is that libraries are typically pre-compiled; therefore, they are unable to be instrumented with TEXAS. The first solution is to recompile the required libraries with TEXAS's compilation pipeline and link the now instrumented library with the program. The second solution, although accompanied with more overhead, is to use CARMOT's PIN tool (§ 5.3.4) to dynamically instrument the library's execution.

The second problem that occurs is during linking. Given an instrumented library, if the library is linked with the instrumented program as well as the TEXAS runtime, then any calls to the library from the TEXAS runtime may result in an infinite loop as TEXAS begins to mistakenly track itself. The first solution to this is to create a separate library for the TEXAS runtime. The second solution has TEXAS create an internal boolean toggle that disables tracking of TEXAS's Allocations and Uses specifically. Both solutions have negligible overhead.

6.3. Manufacturing Locality on Data Structures

As an initial investigation, I focus on improving the layout of Allocations inside common mapping data structures. The reason for this focus is that mapping data structures are commonly found across all types of applications. If locality can be manufactured for these underlying data structures, then it indicates that there is potential for Manufacturing Locality in the many applications that use them.

6.3.1. Data Structure Tester (DST)

To test if mapping data structures are amenable to Manufacturing Locality, the Data Structure Tester, or DST, was created. The DST is a program that abstracts mapping data structures into a single general abstraction. The abstraction implements the very basic functionality a user would expect from a mapping data structure: insertion, deletion, update, and lookups. Using this very basic abstraction, various data structures can be slid underneath and tested for manufacturing locality. Using the mapping data structure, the DST performs traces of allocated node insertions followed by lookups on the data structures. Figure 6.7 shows an overview of the DST.

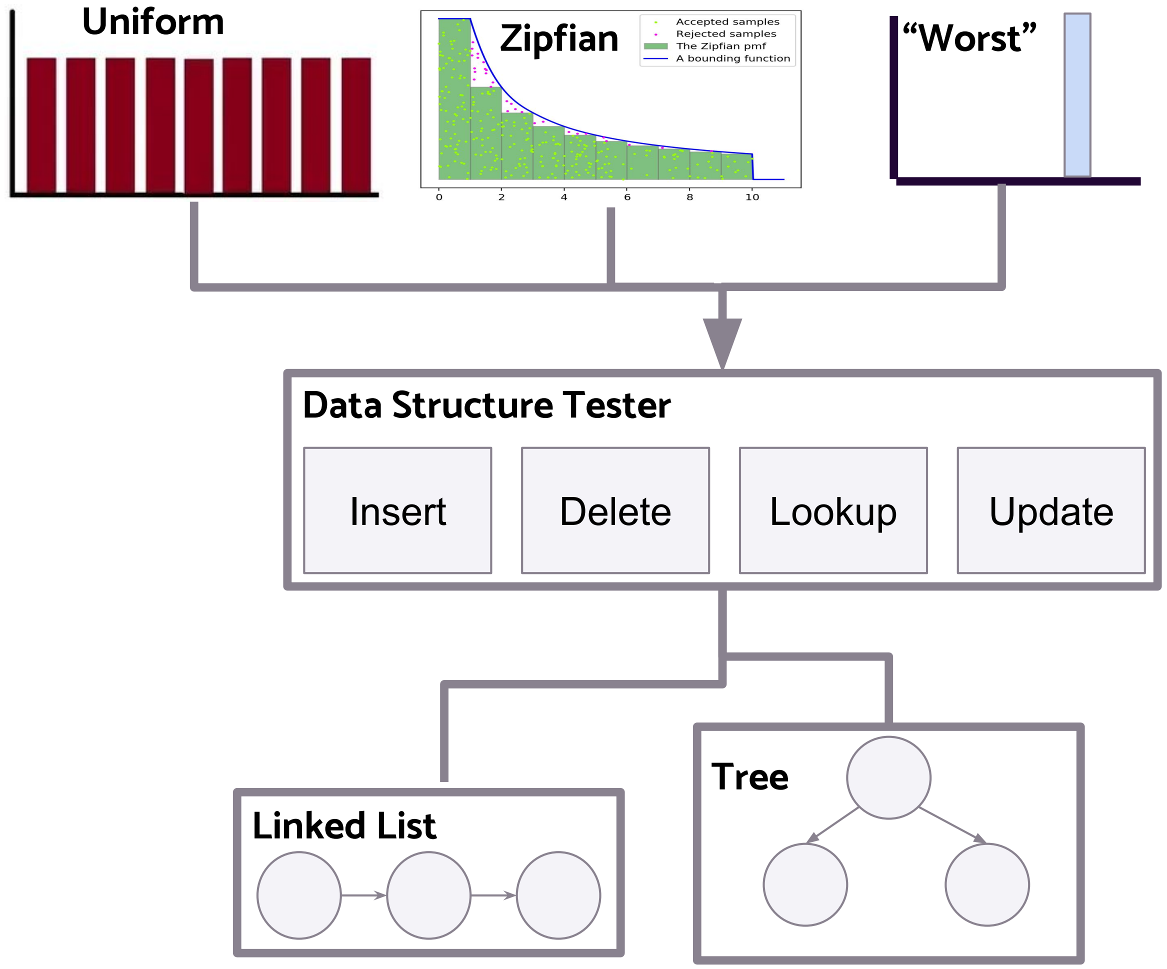


Figure 6.7. Overview of the Data Structure Tester

6.3.1.1. Traces. Given the DST, we test out Manufacturing Locality with multiple trace distributions. These traces offer different views of workloads that can be expected for these data structures. The distributions chosen for the DST are a uniform, Zipfian, and "Worst" distribution.

The uniform distribution searches through the data structure with keys selected as according to a uniform probability distribution. This allows us to determine how well Manufacturing Locality can perform when the observed temporal trace does not provide richer information than being given the Connection Map produced by Escapes. The uniform distribution can be considered the worst distribution for Manufacturing Locality's Temporal heuristic to optimize as there are no discernible trends observed.

The Zipfian distribution searches through the DST using keys that follow a power-law popularity distribution. This type of distribution is a common workload that is observed in many applications and represents a workload that would be expected to be performed on the DST in actual applications. The Zipfian distribution is the expected behavior of reality and offers expected optimization potential for Manufacturing Locality's Temporal heuristic.

The Worst distribution searches through the DST using the single key that would result in the most work for a linked list, namely always searching for the last node in the list. Consequently, this distribution offers the Temporal repacking heuristic of Manufacturing Locality the best improvement, as the traveled path will be deterministic and perfectly regular. The Worst distribution can be considered the best distribution for Manufacturing Locality's Temporal heuristic to optimize as there is a single highly regular trend to observe.

6.3.1.2. Node Layout. The DST also controls the initial layout nodes in the data structure have amongst themselves, and lookup/access distribution. By varying the layout and lookup distribution, we can test the effectiveness of attempting to manufacture locality for different layouts and workloads.

For simplicity, the DST performs operations in two phases. The first phase inserts nodes into the DST's selected data structure. The second phase performs lookups into the data structure according to the given distribution.

To accomplish this, all the nodes that will be inserted into the DST are allocated by a factory allocator before insertion into the DST. Initially, the nodes are staged to be inserted in an order in which the current inserted node is spatially local to the previously inserted node; however, this is the point where mutations to the insertion order can be made. Before insertion begins, locations of pre-allocated nodes can be shuffled to disrupt the layout, and therefore the spatial locality, of the nodes being inserted.

For testing out the DST, we have two initial layouts for the nodes. The first performs no shuffling of nodes, allowing the layout to relatively match the upcoming access pattern[5]. The second performs a randomized shuffle of the nodes being inserted. This shuffling removes the matching of layout to access pattern and offers TEXAS more opportunity to re-Manufacture the locality that was lost.

6.3.1.3. Data Structures and Repacking Heuristic.

Data Structures The data structures that Manufacturing Locality will attempt to optimize are linked lists and binary search trees. These data structures were selected for their search/layout complexities. For a linked list, the searches that are performed are O(n), where n is the number of nodes in the linked list. This structure, outside of profiling a linked list, will also give us insight into Manufacturing Locality's potential when it can optimize a very long list of temporally accessed nodes.

[5]For linked lists, spatial locality will 100% match temporal locality for the DST.

A binary tree was selected, as trees are also very common data structures used in applications. Although it does not contain the complexities found in red-black trees or splay trees, it does represent what gains might be expected for more complicated tree data structures.

Repacking Heuristic The Temporal repacking heuristic will be used for the data produced in Section 6.3.2. The Temporal repacking creates the largest run time overhead out of all the repack heuristics in TEXAS. The performance gain that Temporal repacking produces for linked lists is equivalent to BFS/DFS repacking. The performance gain that Temporal repacking produces for binary search trees is equivalent to DFS repacking. If the programmer can statically inform the runtime that the data structure will be optimized by a BFS/DFS repacking, then performance can be improved through using these repacking heuristics. The use of the Temporal heuristic is completely agnostic to the programmer and offers us the chance to see the upper bound of Manufacturing Locality's produced overhead. It also informs us of the performance gains of BFS, DFS, and Temporal repacking heuristics.

6.3.1.4. Testbed. All numbers we report are collected on a dual-socket system with 2.3 GHz Intel Xeon E5-2695 v3 processors and 256 GB of DRAM spread across 2 NUMA zones. It runs RHEL Server release 7.7 with a 3.10.0 Linux kernel.

6.3.2. Results

Using the DST and its control knobs, we were able to make a few observations about Manufacturing Localities potential in improving locality.

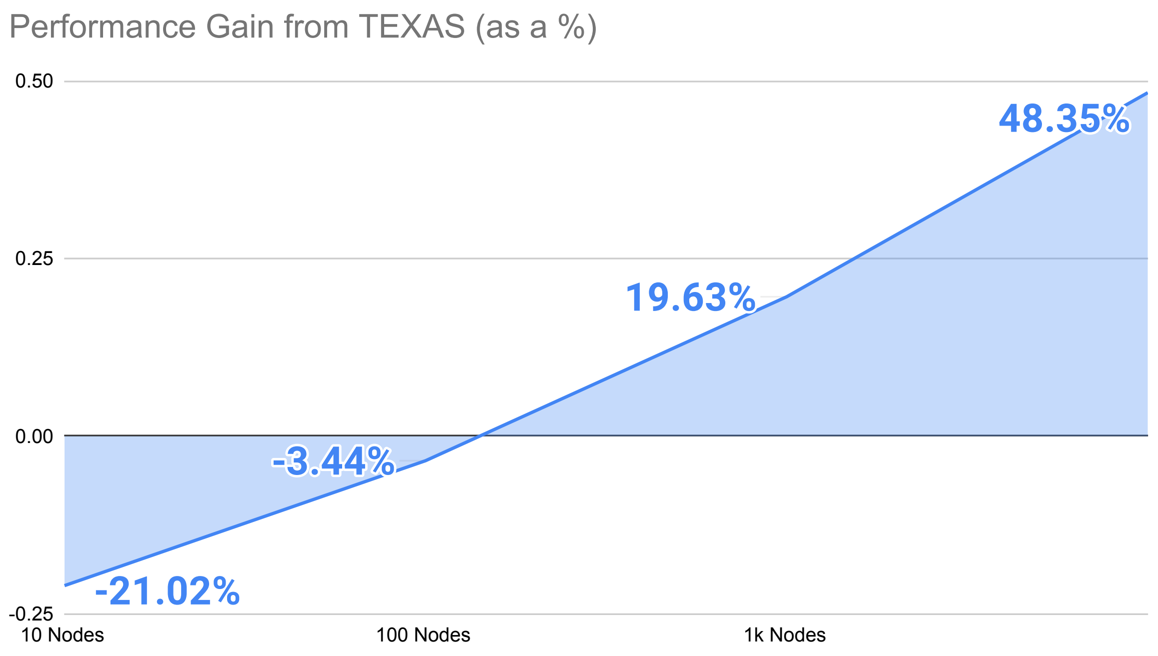


Figure 6.8. Impact of increasing the size of a data structure being optimized by Manufacturing Locality

6.3.2.1. Effect of increasing data structure size. Figure 6.8 shows the impact of increasing the size of the data structure that Manufacturing Locality is trying to optimize. The first trend observed is that increasing the amount of nodes being operated on allows Manufacturing Locality to have a more positive pronounced effect on performance. This observation suggests that when Manufacturing Locality is optimizing a linked list, having more nodes in the linked list will lead to better performance.

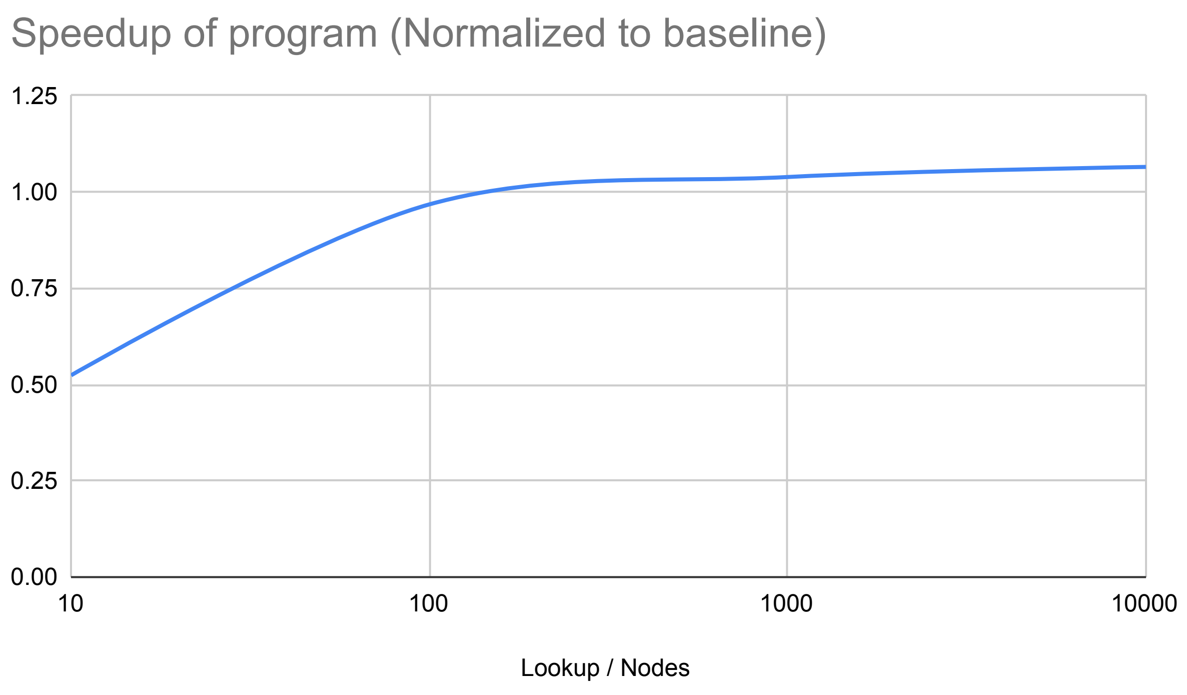


Figure 6.9. Impact of increasing the number of lookups per node of a data structure being optimized by Manufacturing Locality

6.3.2.2. Effect of increasing lookups/node. Figure 6.8 shows the impact of increasing the number of lookups per node of the data structure that Manufacturing Locality is trying to optimize. The second trend observed is that increasing the ratio of lookups to the amount of nodes reduces the overhead that TEXAS introduces, especially at low amounts of nodes. This effect is seen more prominently with the 10 and 100 node lines, in which the programs typically experience an overhead rather than a performance boost when operating with a low lookup/node ratio.

This observation suggests that with a program that only uses its data minimally, there is likely little benefit (and more likely detriment) to using Manufacturing Locality.

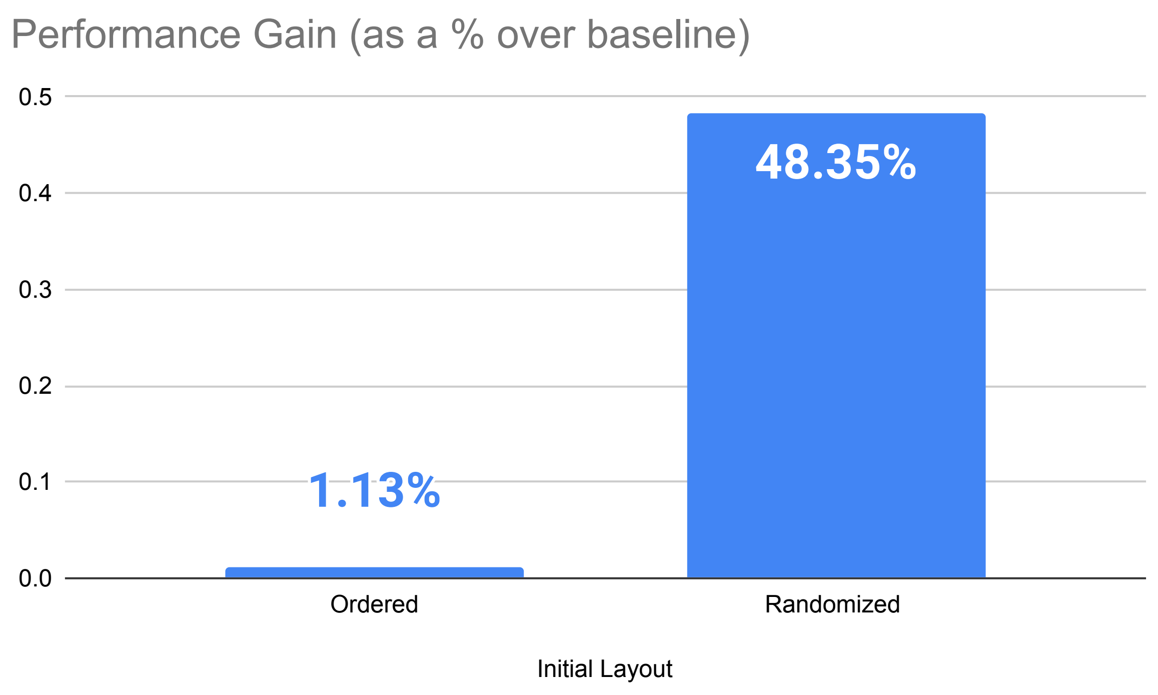


Figure 6.10. Impact of initial layout of a data structure being optimized by Manufacturing Locality

6.3.2.3. Effect of initial layout to access pattern matching. Figure 6.10 shows the impact of the initial matching of the data structure's layout to the access pattern that Manufacturing Locality is trying to optimize. The third trend observed is that Manufacturing Locality can make a significant performance improvement when operating on data which does not have the layout match the access pattern. This effect is seen when comparing the equivalent data points between the upper and lower plots for each figure. For example, Figure 6.12 shows a performance gain for the green 1k and 10k node lines and a slight performance gain for the 100 node line at 10k lookups/node data point. This observation suggests that there is the potential for Manufacturing Locality to improve the performance of linked lists when the layout of the nodes do not match the access pattern of the nodes.

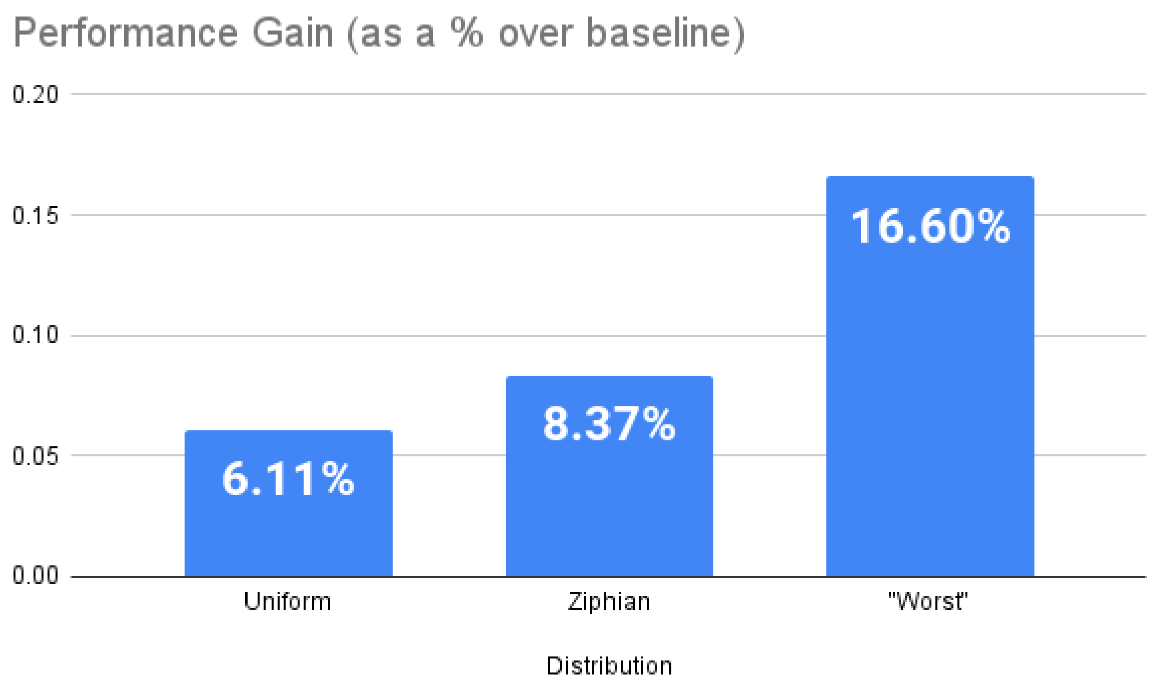


Figure 6.11. Impact of access pattern of a data structure being optimized by Manufacturing Locality

6.3.2.4. Effect of traces. Figure 6.11 shows the impact of the initial matching of the data structure's layout to the access pattern that Manufacturing Locality is trying to optimize. The fourth trend observed is that Manufacturing Locality can make varied performance improvement depending on the distribution of the access pattern being used. In the figure, we show the performance gains of a binary search tree when manufacturing locality around different access patterns. As we can see, when the access pattern distribution displays more skew, we have greater potential for performance gains. I also want to note that even with a perfectly uniform access pattern, Manufacturing Locality still was able to offer performance gains, even if they were not to the scale of Zipfian or “Worst”.

6.3.3. Takeaways from the DST

From running experiments on the DST, we can determine that Manufacturing Locality has definite potential for outright improving the performance of existing applications that use data structures like linked lists and trees. Furthermore, it suggests that there might also be the potential to improve applications that use graphs in general.

6.3.4. Raw Results

The following graphs are the results of running the DST and varying the control knobs. These results were used to form the observations discussed in Section 6.3.2.

6.3.4.1. How to read figures. The following figures show how the performance of the data structures being used by the DST improve or suffer based around number of nodes as well as lookups performed.

The vertical (y) axis on figures 6.12-6.17 show a normalized performance of a TEXAS instrumented version of the code that performs Manufacturing Locality on the DST over a baseline that shows an uninstrumented version that simply runs without TEXAS. If the data point/line on the figure goes above 1.00, then Manufacturing Locality actually improves the performance of the program by the percentage above 1.00 it is. For example, figure 6.12’s bottom figure has a green line that has a y-value of approximately 1.5. This 1.5 means that the performance of the DST when running with Manufacturing Locality performs 50% faster than the baseline[6].

The lines on the figures differ in the number of nodes inserted into the DST's data structure. The blue line shows results for a DST data structure containing only 10 nodes, and continues up to the green line showing results for a DST data structure with 10k nodes inserted.

The horizontal (x) axis on figures 6.12-6.17 show a change in the amount of lookups (of the given distribution) performed on the DST's data structure, or lookup/node ratio. The reasoning for this ratio rather than absolute numbers is highlighted by the green lines of 10k nodes in the DST data structures being tested. Data structures tend to have more lookups than insertions into them. Having only 10 lookups into a 10k nodes data structure does not provide useful information. Instead, we have the lookups scale with the size of the data structure itself in ratios of 10x, 100x, 1000x, and 10,000x more lookups than nodes in the data structure. Reading one of the lines from low to high x values (left to right) will tell an observer how performance changes as the DST's data structure experience a heavier balance of lookups/nodes.

6.3.4.2. Linked List. Figures 6.12-6.14 show the performance implications of using Manufacturing Locality on a linked list.

[6]The results also account for the overhead that TEXAS introduces.

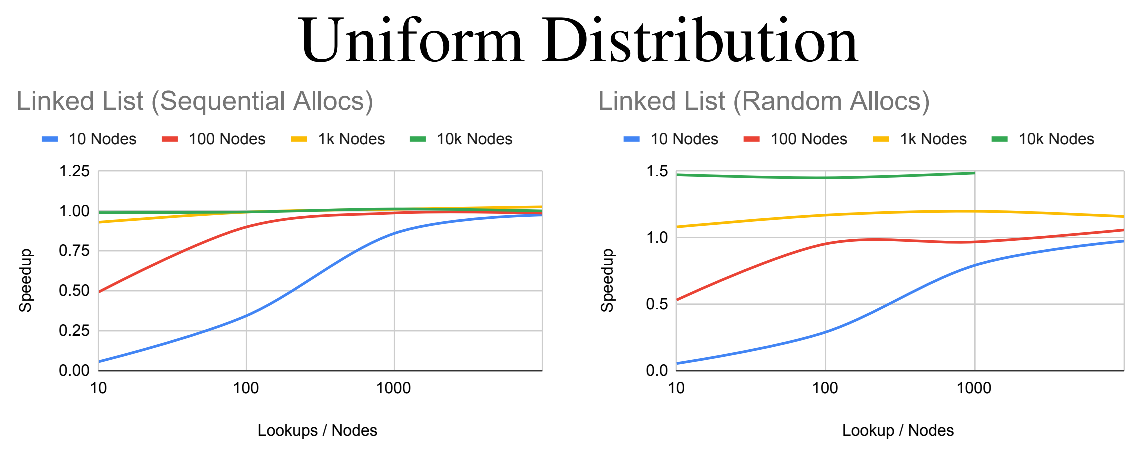


Figure 6.12. Linked list performance improvement with uniform distribution of accesses

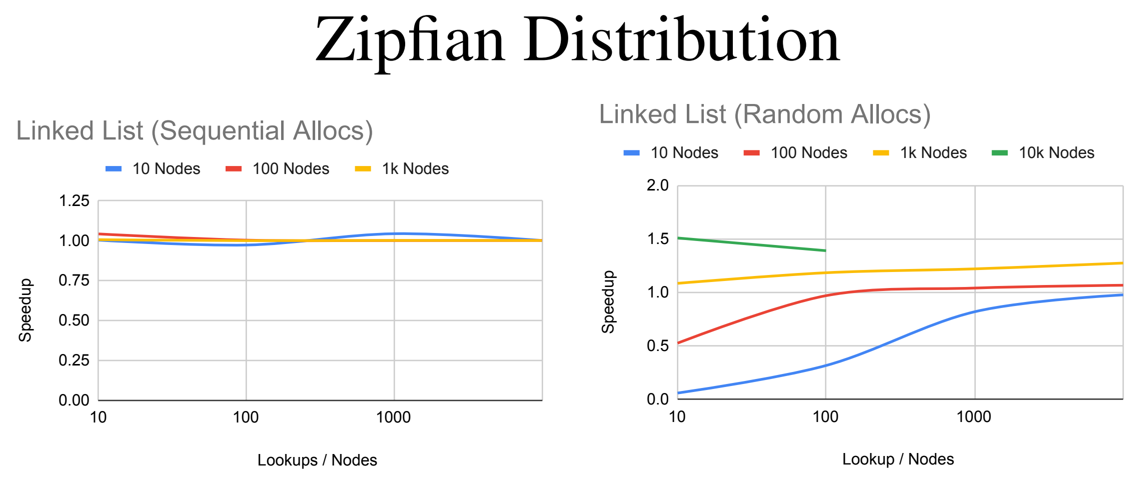


Figure 6.13. Linked list performance improvement with Zipfian distribution of accesses

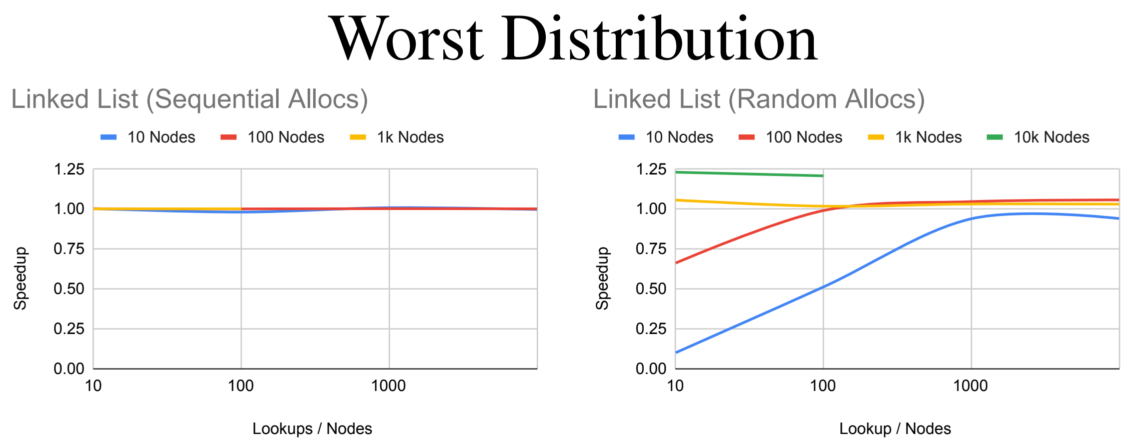


Figure 6.14. Linked list performance improvement with worst distribution of accesses

6.3.4.3. Binary Tree. Figures 6.15-6.17 show the performance implications of using Manufacturing Locality on a Binary Tree.

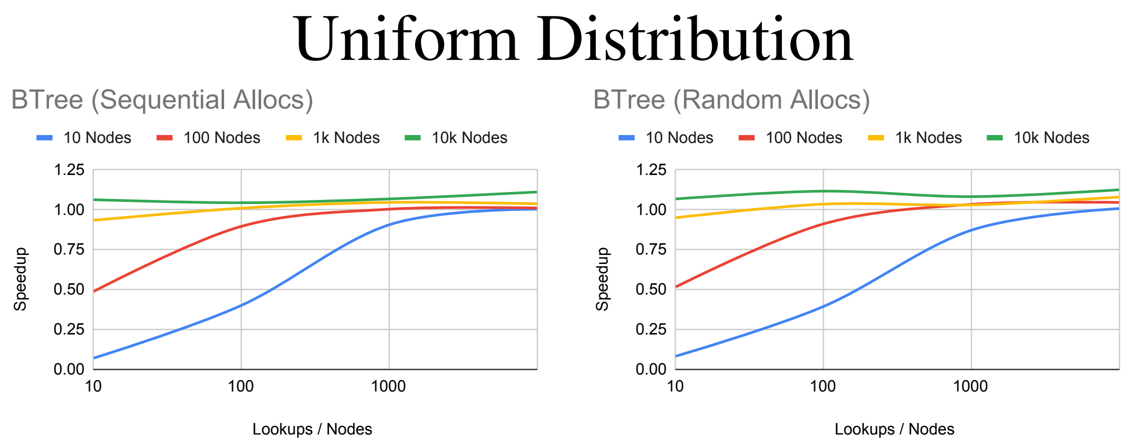


Figure 6.15. Binary tree performance improvement with uniform distribution of accesses

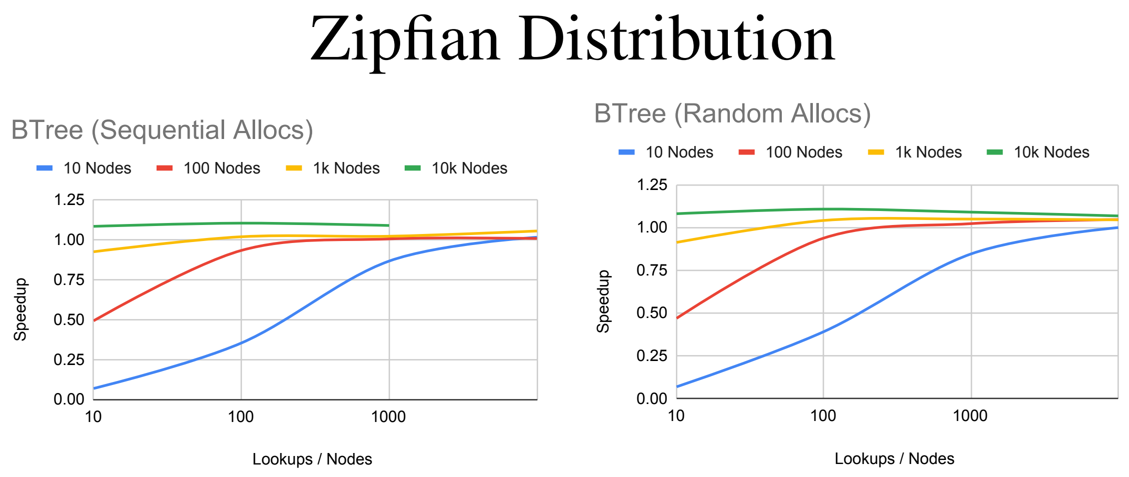


Figure 6.16. Binary tree performance improvement with Zipfian distribution of accesses

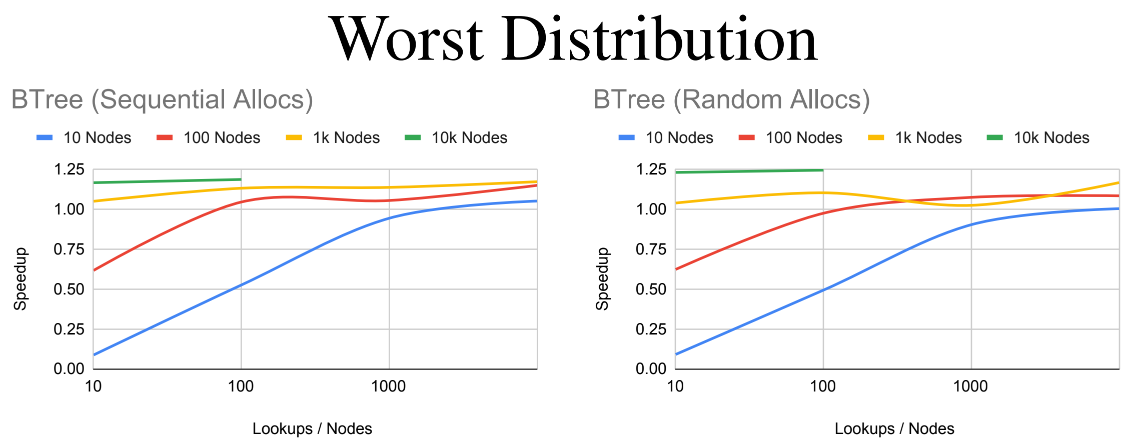


Figure 6.17. Binary tree performance improvement with worst distribution of accesses

6.4. Contributors

This project was achieved through a great effort beyond myself for many components of this work.

My contributions primarily involved (but was not limited to): the extension of TEXAS to accommodate Manufacturing Locality, the creation of the DST for evaluation, and assisting in the TEXAS Loop compiler pass.

Tommy McMichen primarily was involved in creation of the TEXAS Loop compiler pass.

Griffin Dube and Nick Wanninger were responsible for significant ease-of-use modifications to the TEXAS repository, as well as taking up the project[7] upon my departure.

Peter Dinda, Nikos Hardavellas, and Simone Campanoni offered guidance as well as helping to craft the overarching story and motivation for Manufacturing Locality.

[7] Tommy McMichen will also be taking up the project.

CHAPTER 7

Collaborative Uses of TEXAS

TEXAS is not only limited to the use cases highlighted in earlier chapters, but also has active lines of research that involve TEXAS in one form or another. Although these lines of work are not the focus of this thesis work, they are noteworthy for being distinctly related to the goals of the thesis as well as being in part based around the work of this thesis.

7.1. TrackFM

One area that TEXAS and its ideas can be applied is in remote/far memory management. Current software based far-memory solutions, have been able to improve memory utilization in cloud data centers, by enabling applications with high memory demands to be met by utilizing memory from a remote server with fast network operations.

Applications that reference memory with non-uniform access patterns benefit from deep, hierarchical memories that match locality of reference to a memory tier with appropriate performance characteristics. For example, Lagar-Cavilla et al. found that applications infrequently access an average of 32% of pages across jobs in Google's warehouse-scale system [**104**], presenting an opportunity for a cheaper, slower tier of memory between DRAM and disk. One example of such a *far memory* tier is *remote memory*, alternatively referred to as *disaggregated memory* [**13**]. In this setup, DRAM on an external server connected over a high-performance interconnect is used as remote swap space. Remote memory systems accomodate memory-constrained applications by scaling workloads across standard machines rather than overprovisioning individual, expensive, large-memory machines, reducing ownership costs [**163**] and mitigating application crashes due to unmet memory demands.

Both software and hardware-based approaches exist for implementing remote memory. Software only approaches rely on a modifying OS kernel [**176, 76, 17**] to provide transparency of access[1] via the paging subsystem. FastSwap is a notable example that uses a modified Linux kernel swap subsystem to leverage unused memory from a remote server [**17**]. The primary benefit of paging-based approaches implemented in the kernel is that application code need not be modified. However, due to page fault overheads in the kernel [**146**], applications incur a significant performance penalty relative to a setup with only local memory [**146**].

While kernel-based, transparent approaches place minimal burden on the developer, their performance ultimately depends on page faults generated by the hardware. This means that access transparency comes at the cost of I/O amplification determined by the architected page size of the system. Specialized hardware can improve this situation by reducing the granularity of memory faults, for example at the granularity of cache lines [**44**], but only research prototypes or simulated systems [**131**] currently exist.

Application integrated far memory (AIFM) presents an alternative approach, where developers use modified (or custom) libraries that include data structures designed to leverage remote memory entirely in user space [**146**]. AIFM builds on the Shenango runtime system [**134**] for high-performance tasking and networking. This approach sacrifices overall transparency, but

[1]The developer is agnostic to the presence of remote memory.

still insulates application developers from dealing with remote memory while achieving significantly improved performance over kernel-based approaches. In the best case, developers need only make minimal changes to their code to leverage the remote versions of data structures (e.g., a remote hashmap). However, if the AIFM libraries do not provide appropriate data structures, developers are left to design their own.

Thus, there is a tension between transparency and performance in state-of-the-art remote memory frameworks. TEXAS shows that modern compilers are well-situated to resolve this tension. The benefits of both transparency and performance can be gained through TEXAS and has resulted in TRAnsparent Compiler-assisted Kernel-agnostic Far Memory, or TrackFM. TrackFM extends the concepts of TEXAS and implements a user transparent far memory management system. TrackFM achieves full transparency using knowledge gained from state-of-the-art compiler middle-end analyses and transforms, and achieves high performance using AIFM as a backend.

7.1.1. Modifications to TEXAS

To enable for remote/far memory, components of TEXAS were extracted to create a customized TrackFM-specific runtime. The first component extracted from TEXAS is a modified Guard Injection pass from the CARAT work (§ 2). In TrackFM, every memory access needs to Guarded to ensure that the access exists on the local machine and not on remote memory. To accomplish this, the Guard Injection pass was modified to enforce this modified Guard for existence rather than protection.

The other component extracted from TEXAS is a modified Allocation (and Allocation tracking). TrackFM at its core operates at a standardized granularity similar to pages, but free from the limitation of 4KB, 2MB, or 1GB. “Allocations” in TrackFM also operate like pages in the sense that multiple individual Allocations can exist within a TrackFM “Allocation” (fully utilizing the arbitrary granularity memory management TEXAS provides).

7.1.2. Contributors

Brian Tauro is the main contributor to this project, but the project involves a large effort beyond just Brian Tauro.

Brian Tauro is involved in every facet of this project as the primary contributor.

Myself and Kevin McAfee are involved in assisting Brian Tauro as need for conceptual design, optimization of the runtime, evaluation, and crafting the overarching story and motivation for TrackFM.

Kyle Hale, Peter Dinda, and Simone Campanoni offered guidance as well as helping craft the overarching story and motivation for TrackFM.

7.2. WARDen

Another area that TEXAS can be of use is in selectively disabling cache coherence in programs written in unmanaged languages. There is work in this area focused around selectively disabling cache coherence for programs written in high level programming languages that enable this disabling to happen without issues due to the construction of the language itself. The construction of these high level managed languages can result in programs that automatically can express the WARD property, a variant of the disentanglement property.

7.2.1. Disentanglement

To avoid potentially-disastrous race conditions, fork-join parallel programs use memory in a highly disciplined manner. There are a number of approaches to this end, including race-detection [**67, 71, 47, 119, 143, 144, 164, 28, 69**], type and effect systems [**39, 90**], programming techniques for ensuring some degree of determinism [**101, 102, 37**], as well as achieving determinism-by-default with purely functional programming [**78, 22, 38, 36, 138, 109, 80, 157, 70, 142, 77, 170, 21**].

Recent work, as of 2020, identifies one such discipline of fork-join programs called *disentanglement*, which (informally) is the property that concurrent threads remain oblivious to each other's allocations [**142, 77, 170, 21**]. Disentanglement has broad applicability; it emerges naturally in fork-join parallel programs [**142**] and is more general than pure functional programming. Disentangled programs may contain arbitrary communication between concurrent threads as long as all shared data is allocated by common ancestors.

A clearer definition would be that a fork-join parallel program is **disentangled** if each thread only holds pointers (in its stack or registers) to data in either its own heap or the heap of an ancestor.

7.2.2. WARD

From disentanglement, the WARD property can also arise. We define the WARD property for a memory location M. M displays the WARD property when two conditions hold for all hardware threads. For any two hardware threads i and j:

(1) There are no RAW dependencies between i and j at M.

(2) WAW dependencies between i and j at M may be resolved in any order.

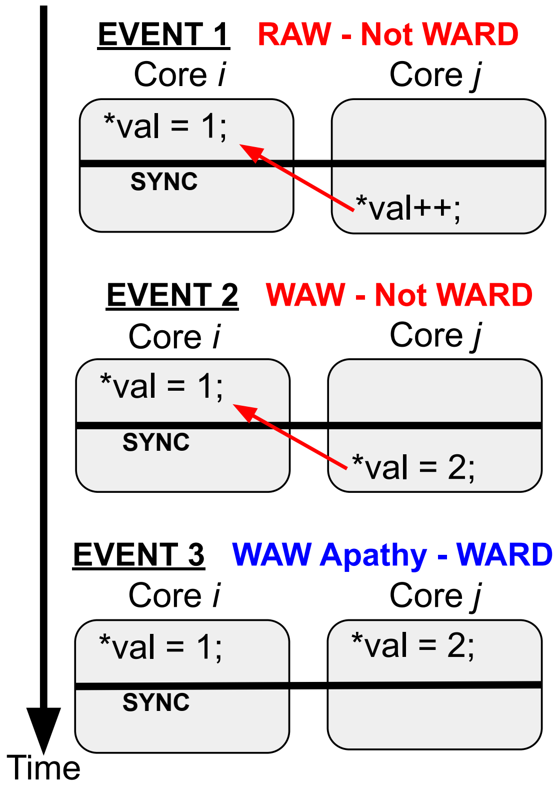


Figure 7.1. Examples of non-WARD (Events 1, 2) and WARD regions (Event 3). Either value is accepted in Event 3 (§7.2.3).

If these conditions are true for all possible combinations of i and j, then M has the WARD property.

Because the WARD property may exist for a specific set of memory locations and/or for a limited amount of time, we refer to the WARD property in terms of regions. A WARD region r is constrained in memory space and time such that:

$$r = \big(\{M\}, (t_s, t_e)\big) \tag{7.1}$$

During the time interval from t_s to t_e, the set of addresses $\{M\}$ have the WARD property.

7.2.3. General WARD Example

WARD can be understood by observing the example in Figure 7.1. Note that the ”sync” line in the figure refers to natural synchronization points such as barriers and fork/join points from the fork/join model. We see that each event includes two cores (analogous to the hardware threads in the WARD definition), which both operate on the same variable.

In Event 1, hardware thread *i* writes to the shared variable *val*. After synchronization, hardware thread *j* reads and subsequently writes to *val*. When this situation occurs, a RAW dependency exists at *val*. Therefore, the WARD property does not exist by condition 1 of the definition.

In Event 2, *i* writes to *val*, then *j* writes to *val*. After synchronization, hardware thread *j* writes a new value to *val*. When this situation occurs, a WAW dependency exists at *val*. The program requires hardware thread *j*’s final value to persist via the memory fence. The WAW is not apathetic, so the WARD property does not exist by condition 2.

In Event 3, *i* and *j* again both write to *val*. We observe that are no RAW dependencies between *i* and *j* at *val* because *val* is never read during the event. On the other hand, there is a WAW dependency. However, the program does not provide explicit ordering, so it is safe to resolve the WAW dependency in either order and maintain correctness. Event 3 meets both conditions of the WARD definition, and thus *val* holds the WARD property for the duration of Event 3.

7.2.4. Connection to TEXAS/CARMOT

The work surrounding detecting the WARD property is currently limited to high level managed languages due to needing to force the WARD property by language construction *or* by having a comprehensive view of memory that a managed language provides. Consequently, TEXAS is positioned to add the functionality needed to enable dynamic detection of WARD regions in C/C++ (unmanaged languages). The extension to TEXAS needed is similar to the extensions performed for CARMOT, mainly use tracking of memory. In fact, Allocations in CARMOT that do not belong to the *Transfer* state in the FSA can have the Allocations memory be classified as WARD.

CARMOT in its current state can only give a conservative classification of Allocations displaying the WARD property. The reason for this conservative classification is due to the existence of WARD-Allocations **within** the Transfer state. Although not belonging to the Transfer state can guarantee WARD, being within the Transfer state is a classification that CARMOT applies for a given execution of a program for its entirety.

WARD is a property that can be valid for subsets of execution in a program; whereas, once an Allocation is classified as Transfer state, the Allocation stays in the Transfer state. Instead, TEXAS/CARMOT could analyze Allocations for smaller subsets of execution and assign the WARD property to Allocations for those smaller windows instead of the whole program's execution.

CARMOT can enable programmers to now automatically identify WARD regions/Allocations, but this again is different than the freely given WARD classification provided by HLPLs. If TEXAS/CARMOT wanted to perform the equivalent automatic detection, then the CARMOT/TEXAS runtime classification would need to become a dynamic online analysis. This analysis can (and does) work; however, there is active work in lowering the overhead of this detection so that it is feasible for applications. For the NAS benchmark suite [**132**], CARMOT's

WARD region detection is currently able to detect that roughly 90% of all variables can be labelled as WARD.

7.2.5. Contributors

Mike Wilkins and Sam Westrick are the main contributors to this project, but the project involves a large effort beyond just them.

Mike Wilkins is involved as the primary contributor to this work and focuses on the simulator, coherence protocol creation, evaluation, motivation.

Sam Westrick is also a significant contributor to this work and focuses on the MPL language, benchmarks, and proofs.

Vijay Kandiah, Alex Bernat, Myself, and Enrico Deiana are involved in this work as support for various parts in every facet of the work.

Nikos Hardavellas, Peter Dinda, Umut Acar, and Simone Campanoni offered guidance as well as helping craft the overarching story and motivation for WARDen.

CHAPTER 8

Related Work

All of the projects worked on throughout this thesis work are related in many different ways to ongoing and historical research thrusts. For this chapter, each of the projects individually breaks down the related work for each project by section.

8.1. Related Work: CARAT CAKE

Extending TLB and virtual memory. There is a long chain of work on improving and extending the existing paging model, each of which attempts to address some measured deficiency [**30, 31, 32, 55, 72, 82, 96, 135, 153, 174, 25, 15**]. In contrast, there is much less work on software-based memory management as an alternative.

Historical software-based memory management. Software-based memory management has a significant historical market impact [**137, 19**], and remains in use today in many forms. One form, automatic handle-based memory management with protection, dates back to the Burroughs B5000 [**43, 108**], which restricts programmers to specific high-level languages. A more recent incarnation is the IBM 801 [**140**], which combined physical addressing, a heavily restricted high-level language (PL.8), and a trusted compiler as the basis for protection ([**140**, pp. 240]).

Compiler approaches. The use of the compiler to implement software protections is not a new concept. EffectiveSan [**63**] uses software checking to to sanitize object types and bounds for C/C++ with low overhead. This work relates to CARAT's guards, but not tracking. CARAT can also be thought of as extending Software Fault Isolation (SFI) [**168, 74, 45, 150**] to all user programs regardless of trust. Many of the innovations in SFI are orthogonal to the concepts of CARAT, and some of the optimizations in CARAT could potentially be deployed in SFI.

Proof carrying code. Another alternative to achieving protection is proof-carrying-code (PCC) [**125, 126, 127**] which has been demonstrated to allow safe kernel extensibility. If code can carry a verifiable proof with it that it is safe with respect to some security policy, then it is possible to eliminate all guards. The optimizations that CARAT is performing for guard amortization is somewhat akin to a compiler trying to generate proofs about the safety of the code.

Sandboxing. Sandboxing, more specifically eBPF [**10**], is also a related area of work. The idea of this work is to set up an environment to run code where it is isolated from other processes and can be observed. eBPF does this via a compilation/loading process similar to the CARAT process. One difference is that eBPF does not run on arbitrary code, but runs on pseudo-C code that is aware of eBPF. CARAT-like systems might be able to complement/enable sandboxing by allowing native low-level code to execute with CARAT guards enforcing the protection/sandboxing.

Singularity. Another related work we are aware of is the Software Isolated Process (SIP) [**85**] of the Singularity OS [**86**]. A SIP is an opaque, self-contained process that communicates through monitored channels and is written in a modified version of C#, called Sing#. The protection and mapping of a SIP rely on guarantees rooted in this managed language, and implemented via its compiler and runtime environment. In contrast, our focus is on unmanaged languages. CARAT CAKE *does* leverage the idea of sealing a process through controlled channels, but the controlled channel is implemented via the compiler and kernel. The SIP concept was in part motivated by a study of address translation performance that showed software-based isolation could have a much lower overhead than hardware-based isolation [**14**], which we believe remains true. Zagieboylo et al [**178**] have more recently revisited the costs of software-based memory management compared to paging.

CHERI. Another line of work to note is the CHERI capability model [**171, 51, 169, 50, 117, 129, 120, 57, 173, 68, 115**]. CHERI's research thrust in MIPS/RISCV implements a fine-grained protection system for general purpose computing. This line of work differs from the CARAT line of work in its goals and execution. CHERI's goal is to enable the enforcement of language memory models and fault isolation through hardware and does so through the modification of MMU (paging) hardware, ISA, and making minor additions to the microarchitecture. CARAT-based systems seek to completely remove the MMU hardware and enforce the goals of CHERI, as well as other problems CHERI is not concerned with, through software. The CARAT line of work is not necessarily at odds with this line of work, but are potentially orthogonal lines of research working in a related area.

O.S. Approaches. LibOSes [**162**], and unikernels [**114**] in general, are also related to CARAT CAKE. Nautilus's original goal was to support individual parallel applications and their runtimes *as* kernels, which can be thought of as an extreme incarnation of a LibOS. Nautilus has been gradually extended to add functionality without losing sight of that model. CARAT CAKE is in line with this model as well.

Also closely related are embedded operating systems such as Tock [**107**], Theseus [**41**], and RedLeaf [**124**], which also leverage the properties of specific languages (Rust) to build protection without hardware support. Tock does provide some support for memory protection for general languages (C), but this relies on simplified memory protection hardware. In contrast to such work, CARAT CAKE's goal is to support arbitrary languages and code through the IR and concomitant transforms of a modern compiler. A viable future line of work exists concerning the utilization of the base concepts of Rust and other memory-safe languages. In fact, languages like Rust could be leveraged by CARAT CAKE when they are used as an application or kernel implementation language.

Virtual Ghost. Virtual Ghost [**56**] protects an application from a hostile kernel (the inverse of CARAT CAKE's protection goal) via compiler analysis and Intel MPX, but paging is required. Additionally, Intel MPX limits the number of regions that can be protected.

Twizzler. Twizzler [**35**] manages non-volatile memory (NVM) at almost arbitrary granularity for applications through automated pointer swizzling. Applications must be ported to Twizzler. In contrast, CARAT CAKE avoids porting and focuses on providing an alternative to paging, particularly for kernel-level abstractions such as processes.

Hardware Codesign. Kadayif et al. [**92, 93**] developed compiler/TLB co-designs that make it possible to skip the normal TLB lookup path when the virtual address is within a set of shortcuts installed by the compiler-generated code. Considerable power and energy could be saved. Like CARAT, this work heavily leverages the compiler to reduce requirements for the hardware. However, it still requires a TLB to function. In contrast, CARAT's goal is to achieve purely physical addressing, avoiding virtual memory support in hardware altogether.

DINAMITE. DINAMITE [**122**] implements a form of CARAT's allocation tracking to support bug-finding, but it is limited to dynamic memory allocations (malloc). CARAT tracks *all* Allocations and Escapes, and uses this information for memory management.

8.2. Related Work: Manufacturing Locality

The idea of manipulation of memory placement in order to optimize for locality and/or coherence finds common ground with different areas of research including (but not limited to) TLB optimization, garbage collection, compiler optimization, and inspector-executors.

DINAMITE. There is prior work whose concept encourages us to pick a more temporally local order. This project is called DINAMITE [**122**]. DINAMITE introduces the concept of tracking the temporal access pattern of the entire execution of a program in terms of dynamic allocations. It does so by instrumenting tracking much like the escape tracking but for every load and store instruction as opposed to just pointer loads and stores. DINAMITE's purpose differs from TEXAS's as it is an attempt to improve profiling of applications so that they may be statically improved as well as only dealing with dynamic allocations. This means that overhead is not important to the project and dynamically improving the program is not done. Implementing some of the ideas of DINAMITE could allow TEXAS to produce an understanding of the temporal access pattern of a program before execution.

Intel MKL. Another tool that attempts optimization on memory is the Intel MKL [**87**]. Intel's MKL is a library that implements optimized versions of math functions, typically for HPC purposes. The MKL library, in tandem with their icc compiler, has an inspector-executor model for sparse basic linear algebra routines that will manipulate memory of sparse matrices before performing computation on them. This work is close to what manufacturing locality is trying to achieve with a few distinct differences. The first distinction is that this library is limited only to its own functions and is not intended to be a general tool like manufacturing locality. In fact, there are many different research papers [**148**] that exist that also deploy inspector-executors in this fashion, but Intel's MKL tool is among the most popular tools. Additionally, MKL, or other tools, will not dynamically run the inspector-executor and instead must use "hints" that are manually set before running the program by either the programmer or via static analysis.

Static layout optimization. In the past there were many static approaches to optimizing memory layouts of programs before they run that are currently used today in compilers [**95, 52, 94, 147, 49**]. These approaches for the most part were effective in improving locality for the programs that they would target; however, the programs that they would target were simple to analyze statically and were very simple and regular, like nested for loops iterating through a matrix. The optimizations I am pursuing are not of this variety. Instead, I am focusing on data structures that are not simply arrays, but instead are arbitrary graphs like DAGs, trees, and other more complex node-based structures. Additionally, I am going to be able to shard, or fragment, the matrices and potentially benefit by optimizing the shard localities.

Garbage collection. Another line of research related to what I am trying to do within this project is garbage collection methods to optimizing locality [**46, 84, 154, 48**]. These method use the garbage collection process of a program to dynamically relocate memory in order optimize the locality of the surviving entries. This approach is similar in the sense that it is dynamic instead of static, but the time granularity, knowledge, and applicability are lacking. In terms of granularity, the garbage collection methods are limited to operating only at times where garbage collection takes place. This results in the improvements not being able to be applied specifically to "hot" portions of the code.

Instead, the applications will in general improve the performance over time. The method of optimizing for locality for many of these methods involves looking at allocations that are simply within a general time range of one another. For example, it could use a metric of if both allocations survived a generation in a generational garbage collector as the metric to spatially locate the entries together. However, the most important difference between these approaches is that they operate within managed runtime languages, like Java. This in itself completely separates my work, because my work focuses on being applicable at the bitcode level. This implies that it can be applied effectively agnostic to the language being used. With this said, the difference between the locality optimization I propose is almost orthogonal (and possibly complimentary) to the garbage collection method which implies a potential opportunity.

8.3. Related Work: CARMOT

While CARMOT is the only tool capable of computing a complete and correct computational spoor, there are other tools that enable programmers to better understand a program's behavior and how to improve it. We compare CARMOT to these tools with respect to their ability to track a program's state, build aspects of the computational spoor, and report back information to the user at the source code level. We categorize these tools in four sets.

Memory correctness. These tools [**7, 1, 5, 6, 8, 3, 65, 2, 128, 4, 9, 152**] detect memory correctness problems of a program such as memory leaks, double frees, wild pointer writes, buffer over/underflow, and use-after-free. Some of these tools report some source-code level information such as the callstack of the error site; however, none of them track any aspect of a computational spoor. The most notable tools that perform some tracking of a program's state and that could theoretically be extended to accomplish a few of the needed tasks to perform CSD are: AddressSanitizer [**152**], Valgrind [**128**], and the Pintool Pinatrace [**9**]. However, none of these tools tracks program variables or are able to distinguish between different stack locations or globals. AddressSanitizer is also unable to track the behavior of a program in precompiled code. AddressSanitizer and Valgrind can detect memory leaks due to reference cycles that should have been garbage collected, but they cannot identify the actual cycles in the source code responsible for the leak. For all these reasons they are not able to compute a correct and complete computational spoor.

Memory performance. These tools [**123, 136, 103, 118**] report memory bottlenecks that limit program's performance, but they do not detect any aspect of the computational spoor.

Parallelism discovery. These tools analyze the memory utilization of a program to identify potential parallelization opportunities using static and/or dynamic analysis and profiling techniques. They can be divided in two subsets: critical path analysis [**73, 79, 145, 16, 100, 83**] and dependence testing [**105, 179, 98, 172, 167, 97, 20, 141, 110, 130, 40, 121**]. Critical path analysis tools identify the critical path of a program and concurrent operations that, potentially, can be executed in parallel. However, they suffer from false positive (i.e., they can identify a program's region as parallel when in fact it is not). Dependence testing tools are more conservative in identifying parallelism, which they do by testing whether a dependence exists or not. If a dependence between two parts of a program does not manifest, then both program's parts can be executed in parallel. We consider these tools and their objective to be orthogonal to CARMOT. In fact, once potential parallel regions of a program are discovered, CARMOT can be used on those regions to understand exactly how they can be parallelized using the supported parallelism-related abstractions, verify the presence of actual parallelism (in case of false positives), and improve it if possible.

Reference counting garbage collection aid. Only two approaches are able to aid programmers in finding cyclic reference counting at the source code level: Xcode [**11**] and Distefano *et al.* [**62**]. Xcode only works for swift and objective-c, but does not currently handle C++ smart pointers. Distefano *et al.* only identifies the line number in the source code that can potentially create a reference cycle, but it does not identify the actual references. CARMOT instead reports the reference cycle at the source code level for C++ programs. Moreover, CARMOT can report how to break cycles.

CHAPTER 9

Conclusion

The ability of software to take its place in the foundations of memory management is becoming more and more apparent as the issues with current memory management become more and more intrusive. Along all components of the hardware/software stack, software offers flexibility, accessibility, and generality in creating new or alternative solutions to problems that are becoming prevalent with the current system in place. These qualities are going to become invaluable over the next era of computing as the workloads and machines also undergo their respective transformations in order to not prohibit progress in performance and efficiency.

TEXAS, and more generality revisiting software-based memory management, shows that there is a path forward instead of relying on our current system from the 1960s. Going forward, TEXAS can serve as a proof of the potential of re-exploring this previous, traditionally infeasible, research direction with a renewed life and vigor.

9.1. Contributions of the Thesis Work

- The primary contribution of this work is the argument that revisiting software-based memory management should be at the forefront of research into solving modern day problems we are experiencing in memory management.
- The Tool to Examine and X'form Allocation State (TEXAS). A tool for acquiring, profiling, and transforming the memory state of a program. This tool has demonstrated its flexibility, as well as usability, for a variety of tasks and enables its users to explore

software-based memory management in a multitude of ways that are alternative to the current hardware and software solution of paging.

- CARAT CAKE, a full system implementation of the CARAT memory management system from the kernel up that can entirely replace paging in computing systems. It also offers a Linux Compatible Process Abstraction that allows modern day benchmarks/programs to run within a CARAT CAKE process. Additionally, this work has led to a peer reviewed publication at ASPLOS 2022 [**160**]. CARAT CAKE also is available as an artifact for researchers to further explore CARAT CAKE's potential, as well as serving as definite proof of the CARAT concept's feasibility.
- CARMOT[1], a TEXAS system that assists programmers in utilizing modern day abstractions to enhance their code with minimal/no modification to the source code. CARMOT automates detection of cycles in C++ smart pointers, generates OpenMP pragmas, generates annotation for STATS [**59**], and introduces the novel new concept of Computational Spoors. Additionally, Computational Spoors have been shown to be relevant for dynamic detection of WARD 7.2 regions for programs written in an unmanaged language.
- Manufacturing Locality, an extension to TEXAS that manufactures spatial locality between Allocations in order to better match the layout to the access pattern of a program. This extension provides various static and dynamic methods to repack memory, dependent on how intrusive the user of Manufacturing Locality wishes to be. In turn, this increased locality leads to improved performance as well as more efficient memory use for a program while remaining transparent to the programmer. On top of introducing

[1]Large contributions of this work are also attributed to Enrico Deiana.

the concept, a contribution of this work is that it provides evidence for the concept through the results of the DST.

- Framework for TrackFM[2], in which the concepts and idea of TEXAS are utilized to enable a far memory management system. Others have also extended the idea of TEXAS's software-based memory management to swapping, specifically with remote memory. This system is transparent to the user and performs better than current transparent solutions, such as FastSwap [**17**]. Additionally, the performance of TrackFM is competitive with current non-transparent solutions, such as AIFM [**146**].

9.2. Observations

Throughout the completion of these projects, I made a few observations about the implications of adopting software-based memory management with respects to different domains of computing not explored in this thesis.

9.2.1. The Role of Hardware

One observation I arrived at is that memory manage is being severely limited if we use the current system of hardware-software codesign. There is plenty of hardware, such as the TLB, which is very limited in the tasks which they perform[3] and consume area/power on the chip in a manner that is unjustifiable when compared to software-only solutions, as highlighted by CARAT CAKE. Although hardware may not currently be synergistic with systems like TEXAS, there is still a potentially bright future for hardware in software-based memory management systems.

[2]The main contributor to this work is Brian Tauro.

[3]It should be noted that they perform their tasks quite well.

The TLB could offer an opportunity for software-based memory management to enhance its abilities. At its core, a TLB is a sub-cycle cache within the micro-architectural pipeline that also performs a minor function, mainly translation and bit checking. If the TLB was slightly generalized into a sub-cycle cache with a small set of software-programmable logic gates, then there could be massive new opportunities for the TLB that would greatly justify its existence in the future. For example, in CARAT CAKE, the cache could be used as the first step in the runtime hierarchy for Guarding memory accesses.

A step beyond the TLB is the potential for integrating FPGAs within software-based memory management systems. FPGAs provide the potential for software systems to offload their tasks onto a piece of hardware specifically designed for accomplishing that task specifically. Because the FPGA's hardware[4] on an FPGA is mutable just like software, the FPGA offers macro-scale operations in software-based memory management to be offloaded. This ability is not limited to just TEXAS/CARAT like systems, but is also applicable to the current memory system. Examples of some interesting use cases today are offloading operations like kernel same-page merging, creation of transparent huge pages, garbage collection, and other macro scale operations.

9.2.2. Parallel Workloads

Another observation I made throughout the completion of the thesis projects are the implications associated with handling parallel workloads. TEXAS, in its current state, is running with

[4]The lookup tables, interconnects, and logic blocks are what is actually mutable

programs that are strictly running with a single thread. Although TEXAS and its ideas do support running multithreaded programs and workloads, there are new implications that must be considered.

For the TEXAS runtime, many aspects of the runtime can remain unchanged. The key difference between a single threaded program running with TEXAS versus a multithreaded program running with TEXAS is the introduction of localized data structures and synchronization. TEXAS can have separate data structures, like Allocation Maps, on a per-thread basis. Upon a synchronization event occurring, the local data structures can also be merged with one another to form a single global state of a program. Overall, tracking and protection from TEXAS's runtime would remain unchanged for overhead, but a new cost of synchronizing the data structures would be introduced to the runtime.

An unexplored problem for parallelization of programs is the lack of proper alias analysis for code analysis and transformation. Alias analysis for parallelization is currently in a similar infancy, akin to how compilers were just 10-15 years ago. Because of this, many optimizations that TEXAS uses that rely on current Alias Analysis would no longer work, or be valid, for multithreaded programs. If TEXAS and software-based memory management were to take a more influential role, then one of the priorities of research would need to be in the development of Alias Analysis for parallel workloads.

9.2.3. Verification and Security

The changing of the guard from hardware-software codesign to a software only design also fundamentally changes the trusted computing base, or TCB. By changing the TCB to be less hardware focused and expanding it into more software, I see opportunities for software-based memory management to make significant impacts. Although there must be caution observed whenever expanding the TCB to new areas[5], software-only solutions like TEXAS can be used to confront current problems facing the TCB. For example, Address Space Layout Randomization, or ASLR, is a method that the TCB currently uses to counter attacks like buffer overflow attacks. A system like CARAT CAKE could instead offer alternative solutions to ASLR because, in combination with its Guards, it has the ability to move any part of memory, including itself. Using things like Use Tracking and Guards perhaps enables a TEXAS-like system to open up research on new side channels or other aspects of security that are not currently being investigated.

9.2.4. Accelerators

Another observation that I made during my thesis work is software's potentially significant role in unifying memory management across different computational devices. For example, GPUs have significant problems with address translation and paging, which in turn has led to an explosion in current research into remedies for these problems. Additionally, accelerators tend to focus on workloads that process and produce data rather than perform structural manipulations on data. Because of their problems paired with their workloads, if GPUs/accelerators operated under a CARAT CAKE system, then there would be many new opportunities and solutions to their problems.

Firstly, by operating on physical addresses with no pages, GPUs could remove most of the complexity that has been causing this massive increase in research. But a TEXAS-like system could also enable us to rethink how memory interacts with different accelerators. Using a GPU

[5]CARAT CAKE expands the TCB to include many more aspects of the compiler.

again, the way that a GPU processes data compared to a CPU is quite different. A GPU relies on the programmer optimizing the processing of data for sets of threads, or warps, that have their accesses interleaved with one another with the main goal of providing very high throughput. On the other hand, a CPU relies on the programmer optimizing the processing for sets of threads to have their accesses completely isolated from one another in order to accomplish the main goal of minimizing latency of processing. These separate goals come into conflict when we choose to force their memory management and layout to be uniform with one another. TEXAS and software-based memory management could allow us to break this limitation. Because memory can be managed as arbitrary granularity and allow the programmer to have control on the layout of memory, new opportunities can arise for optimizing the management and layout depending on what accelerator or CPU is operating on the data.

9.2.5. Adopting Software-based Memory Management in the near-Future

The potential of Software-based Memory Management is definitely something that should be pursued in the future, but the path, or paths, to replacing the current system is not yet defined. In order for the new management scheme to take over, there must be incentives that can offer improvements today to push forward Software-based Memory Management. some potential areas that could offer promise today are:

CARAT CAKE's Future Work CARAT CAKE provides a system that has little to no overhead today to run a CARAT system on x86 architectures. The future work of CARAT CAKE for databases, JITs, Garbage Collection, and Virtualization offer more immediate applications of Software-based Memory Management on existing machines. These types of future work can offer new benefits to these applications/goals without needing to modify the hardware or current system[6]. These thrusts are further discussed in Section 9.3.1.

Handle-based Memory Management Another more immediately obtainable goal would be to utilize TEXAS to implement an automatic handle-based memory management system on top of virtual memory. This would be accomplished by using a TEXAS runtime that handles memory management in its entirety, instead of working with system allocators. Additionally, the CARAT protection pass would be modified to instead automatically perform the handle acquisition and release transparently to the programmer.

By implementing this handle-based system, with TEXAS underpinning its operation, the need for compiler instrumentation of Allocation and Escape Tracking will no longer be needed. Allocation Tracking will instead be performed by the runtime when granting memory to an allocation request. Escape Tracking gains the greatest benefit as there is now only a single Escape that can exist per Allocation, the handle itself. Although seemingly similar to a re-implementation of paging[7], the key difference lies with the complete freedom enabled by not being limited to what the hardware can provide to the system.

[6]Modifying the system would amplify the benefits of CARAT CAKE's future work.

[7]Paging could be thought of as the handles being the virtual address and the Allocation being the physical page.

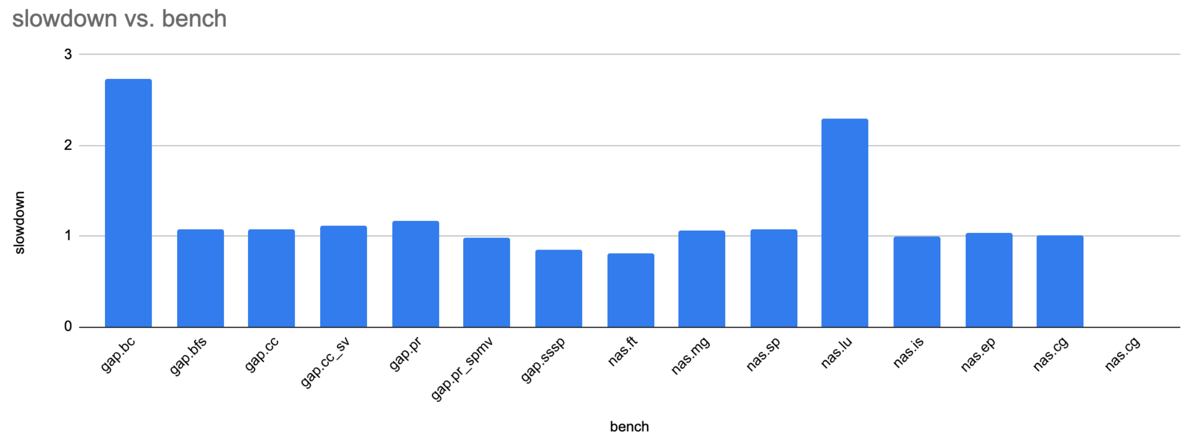


Figure 9.1. Initial investigation into overheads of handle-based memory management via TEXAS.

Initial investigations into implementing this system are shown in Figure 9.1. The initial investigation indicates that the overhead of running such a system would be on par with the overheads that the original CARAT2 work.

9.3. Future Work

The opportunities for TEXAS are not yet fully explored. Each of the following subsections relays the current next steps for each of the three primary thesis projects.

9.3.1. CARAT CAKE

CARAT CAKE is the first prototype full system implementation of the CARAT concept. We claim that CARAT is a feasible, general purpose alternative to paging that could enable a range of benefits. There are untested aspects of CARAT CAKE with regard to a general purpose goal.

Other Benchmarks and Workloads Although CARAT CAKE is not evaluated on an *extremely* diverse set of workloads, there is no particular reason why a CARAT system could not feasibly and efficiently run them. This is in part supported by the results from the prior original investigation [**159**], which shows a much wider range of benchmarks from NAS, PARSEC, and Mantevo. We also note that for overlapping benchmarks, the overheads presented in the original investigation 2 have been reduced due to efforts in compiler optimization. This supports a claim the original work makes, that compiler optimization enhancement will continue to lower the overheads of CARAT-like systems.

One important workload not explored by either paper is a server application such as a web server or database engine. We have not done so because extending LCP to support such applications would be significant engineering effort. However, we would expect CARAT to perform quite effectively, if not synergistically, with these types of applications, particularly a database server. A high performance database engine typically attempts to allocate/`mmap` a single large chunk of memory from the kernel and use it as a scratchpad during execution.

The concept of incorporating a kernel-mode database is already being explored [**75**]. In essence, it is already trying to eschew the kernel and other actors when it comes to memory management. CARAT CAKE is positioned to easily allow a database engine to have this bypass. Tracking overhead would likely be negligible, especially if the memory involved was a single or small number of regions. Similarly, protection overheads would be low given most operations would simply touch this scratchpad. Should the compiler understand this, guarding the database engine would be heavily optimized for these accesses, allowing the database free reign over its data, while maintaining protection.

Just-in-time Compilation (JITs) CARAT CAKE currently relies on LLVM compiler transformations applied statically to implement instrumentation for tracking and protection. However, the fundamental ideas of CARAT are arguably compatible with more dynamic compilation approaches, such as JITs. Instrumenting protection and tracking could be implemented dynamically rather than statically. There may even be a convergence underway that will simplify this through the dynamic use of LLVM. For example, the JavaScript JIT WebKit[**53**] has historically used LLVM optimization in its JIT compilation infrastructure, which strongly suggests CARAT's tracking and protection could be implemented within it.

Garbage Collection CARAT is conceptually similar to a garbage collector, and its model of tracking all allocations and references to them (escapes) has direct analogs to garbage collector primitives. We don't garbage collect because our goal is to provide an alternative to paging for as wide a range of programs as possible, in particular those written in unmanaged languages. A future line of work in CARAT is to integrate a CARAT system with existing garbage collectors, leveraging the information garbage collectors provide and vice versa. We anticipate a CARAT CAKE defragmenter could be integrated with language-specific garbage collectors, as we only need GC movements to be visible to CARAT CAKE. This would allow the first step in defragmentation to be a call to the process's garbage collector, if one exists.

Virtualized Environments While virtualization is not a focus of this work, CARAT CAKE is certainly compatible with it. Indeed, a CARAT CAKE-based guest could present a special opportunity since the GPA$\rightarrow$HPA mapping could then simply be an unchanging identity map. Note that you could view the resulting GVA$\rightarrow$GPA$\rightarrow$HPA mapping either as nested paging without any "inner" page tables, or the best possible case of shadow paging.

In a system without any hardware paging support, including for virtualization, if a trust relationship (e.g., via attestation) could be established for the compiler toolchain used to build the guest applications and the toolchain used to build the guest kernel, and we could a determine that protections had been added to both the guest kernel and its applications, the host kernel/VMM could treat them essentially as processes.

CARAT-based systems would be largely orthogonal to namespace-based containers, since in the end all that would be needed is a slightly fancier process abstraction.

Swapping, Remote Memory, and Handles We also want to have a notion of a memory object not being currently present in memory. This can be used in swapping, to support machines with insufficient physical memory, and currently in lazy evaluation (e.g., demand paging), in PGAS parallel computing models/languages, and increasingly to support remote memory in cloud environments.

For x64 systems, our original investigation [**159**] proposed the use of non-canonical physical addresses to signify an absent object. When accessing a non-canonical address, an x64 system can generate a general protection fault (not a page fault), or use compiler instrumentation to dynamically detect non-canonical addresses. Furthermore, when the object is not present, the pointers to it can be patched to not just be non-canonical, but also to have unused address bits overloaded as a mapping key to the object's current location. Note that for a scenario in which swapping is common, the overhead is likely to be dominated by the swapping costs, not CARAT-based costs.

Even if hardware support similar to x64 non-canonical address checks or Intel MPX checks is unavailable, a CARAT-like system could potentially employ handle-based memory management to have the same effect. In essence, our compiler analysis for protection checks already optimizes for where handle acquires should go. The analysis could also introduce handle releases for both correctness and to optimize the duration between the acquire and release. Then, a handle acquire call to the runtime could detect an absent object and fetch it. Partial work towards this goal has started with the TrackFM research thrust.

Pointer Obfuscation One of the limiting factors for the CARAT concept revolves around the complexity of pointers. More specifically, programs that are still compilable but are invalid with respect to its language standard. One key idea of CARAT is the ability to keep track of Escapes to Allocations. In the common case, this is a simple task that keeps track of locations in which the potential escapes may exist. When an Allocation is moved, these potential escapes must be "patched", and the CARAT runtime determines at this point if the escape truly does alias the Allocation, and modifies it accordingly.

What if the escape is encoded? In this case, the pointer would not be detected as aliasing its respective allocation unless the escape is decoded before the aliasing check is performed. An example of unintended pointer obfuscation is an XOR linked list[8]. In order to accommodate such obfuscations, the CARAT runtime would need to either be able to avoid/disable pointer encoding or be able to decode the pointer at runtime.

For example, simple LLVM IR arithmetic/bitwise encoding (like an XOR linked list), the compiler can track the pointer arithmetic in the IR via the use-def chain or by querying the dependence analysis. This could, in many cases, allow the compiler to handle decoding the pointer (reducing the amount of times an already rare situation like this would require special attention).

[8]An infamous bane of garbage collectors worldwide.

Self-modifying code Self-modifying code presents a significant issue to CARAT systems. Because CARAT CAKE relies on injection of Guards to a program, if a programmer is able to modify this injection, then vulnerabilities can appear. CARAT's current solution is to disallow self-modifying code, but there are a couple ideas that would allow CARAT to ease this restriction.

The first solution, although breaking transparency, would be to require proof carrying code, or PCC, when a programmer wishes to use self-modifying code. PCC provides a proof along with the code that allows the system to verify a certain behavior or trait of the code before executing. For CARAT, this trait could be ensuring the Guards are not modified. The second, more complex, solution would be for the compiler itself to verify the code during compilation. Although this solution restores the transparency, it creates a new demand for compiler analysis that as of now is not currently robust. Both of these solutions could be used together in a hierarchical manner as well. For example, when the compiler is unable to statically verify the code, then it can break transparency and demand a proof from the programmer in order to use self-modifying code.

Fully turning off the TLB The final piece of future work for CARAT CAKE is to run on a system that can actually completely disable the TLB and operate with only physical memory. That effort is currently underway with research into running Nautilus on a RISC-V architecture. Once Nautilus is able to run on RISC-V, there are RISC-V boards as well as simulators that enable CARAT CAKE to run on them with an entirely disabled TLB. This will finally actualize the promised energy savings that CARAT CAKE promises. Additionally, running on a simulator allows CARAT CAKE to modify the cache hierarchy to be physically addressed and tagged as well as increasing the size of L1 cache equivalent to the space that the TLB consumes to test performance gains promised by CARAT CAKE.

9.3.2. CARMOT

Runtime Improvement The first future work for CARMOT involves general improvement of the runtime in order to reduce memory consumption and lower the performance overhead. There are multiple ideas being explored to accomplish these goals including: a revamped memory management scheme, shadow memory that embeds the FSA, and assigning Computation Spoor states to ranges of addresses, rather than individual addresses.

More use cases CARMOT is also actively exploring additional use cases that can make use of the Computational Spoor. Some current use cases being developed include: C++ perfect forwarding, dynamic WARD region detection, OpenMP Target, and detection of memory races.

9.3.3. Manufacturing Locality

Benchmarks and applications The first piece of future work for Manufacturing Locality is to add more data structures for the DST as well as their evaluations. Currently, there is added support for linked lists, binary trees. Additionally, the addition of red-black trees, splay trees, hash maps, and AVL trees are in active development.

The second piece of future work for Manufacturing Locality is to attempt optimizing actual benchmarks and applications. Currently, Manufacturing Locality has been shown to work on data structures that underpin important applications, but actually attempting to optimize the applications is the true challenge. Currently, Manufacturing Locality has been shown to be able to instrument benchmarks from Parsec [**33**], NAS [**132**], GAP [**27**], the MCF benchmark from SPEC2017 [**54**], and GROMACS [**165**]. In order to test these benchmarks/applications, Manufacturing Locality still needs to locate the hot loops of the programs and instrument them for Manufacturing Locality. Additionally, Manufacturing Locality's runtime is still in active development and is not completely stable for these more complex instrumented benchmarks.

Profile-based loop selection and heuristic selection Another significant future work to be done is the automation of the transformation of programs for Manufacturing Locality. Currently, Manufacturing Locality operates on loops designated by the user. In the future, profile guided selection based on loop hotness can be used to make the selection of loops completely transparent to the programmer. Additionally, once a loop is selected, the repacking to heuristic repacking is currently also designated by the programmer. In the future, the compiler can use profiling and analysis to determine what type of repacking will benefit the loop about to execute the most. If the compiler can determine that the loop is going to traverse a graph in a BFS manner, then the BFS repacking will be the best available heuristic to repack with; however, if the repacking heuristic cannot be decided, then the temporal repacking heuristic can be used to observe arbitrary access patterns.

9.4. Final Thoughts

The current model for memory management is a system that dates back to the 1960s, a time when the landscape of computing was vastly different than it is today. We collectively are short-changing ourselves if we make this observation and then do not decide to act upon it. Although research has expanded into extending the current memory management model to address its issues. If we do not take a more radical approach, the the bottlenecks of modern memory management systems will become more apparent and prevalent as workloads and machines continue to change and scale. We have the opportunity to make this radical change via Software-based Memory Management upon every facet of modern computing from embedded systems up through exascale computers, and by doing so can offer us a direction to continue to *improve* computing instead of fixing the problems of the past.

References

[1] dmalloc. https://dmalloc.com/. Accessed: 2020-03-16.

[2] Duma. https://www.linuxlinks.com/duma/. Accessed: 2020-03-16.

[3] Electric Fence. https://linux.die.net/man/3/efence. Accessed: 2020-03-16.

[4] Intel Inspector. https://software.intel.com/en-us/inspector. Accessed: 2020-03-16.

[5] jemalloc. http://jemalloc.net/. Accessed: 2020-03-16.

[6] MemWatch. https://www.linkdata.se/sourcecode/memwatch/. Accessed: 2020-03-16.

[7] Mpatrol. http://mpatrol.sourceforge.net/. Accessed: 2020-03-16.

[8] Mtrace. http://man7.org/linux/man-pages/man3/mtrace.3.html. Accessed: 2020-03-16.

[9] Pinatrace. https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/. Accessed: 2020-03-16.

[10] What is ebpf? an introduction and deep dive into the ebpf technology.

[11] Xcode. `https://developer.apple.com/xcode/`. Accessed: 2020-04-16.

[12] The data deluge. *The Economist* (February 2010).

[13] AGUILERA, M. K., AMIT, N., CALCIU, I., DEGUILLARD, X., GANDHI, J., SUBRAHMANYAM, P., SURESH, L., TATI, K., VENKATASUBRAMANIAN, R., AND WEI, M. Remote memory in the age of fast networks. In *Proceedings of the Symposium on Cloud Computing* (Sept. 2017), SoCC '17, pp. 121–127.

[14] AIKEN, M., FÄHNDRICH, M., HAWBLITZEL, C., HUNT, G., AND LARUS, J. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness (MSPC)* (2006), p. 1–10.

[15] ALAM, H., ZHANG, T., EREZ, M., AND ETSION, Y. Do-it-yourself virtual memory translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, pp. 457–468.

[16] ALLEN, F., BURKE, M., CYTRON, R., FERRANTE, J., AND HSIEH, W. A framework for determining useful parallelism. In *Proceedings of the 2nd international conference on Supercomputing* (1988), pp. 207–215.

[17] AMARO, E., BRANNER-AUGMON, C., LUO, Z., OUSTERHOUT, A., AGUILERA, M. K., PANDA, A., RATNASAMY, S., AND SHENKER, S. Can far memory improve job throughput? In *Proceedings of the 15th European Conference on Computer Systems* (Apr. 2020), EuroSys '20.

[18] AMIT, N. Optimizing the TLB shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 27–39.

[19] APPLE CORPORATION. *Inside Macintosh*. Addison-Wesley, 1985.

[20] ARABNEJAD, H., BISPO, J., BARBOSA, J. G., AND CARDOSO, J. M. An OpenMP based parallelization compiler for C applications. *Proceedings - 16th IEEE International Symposium on Parallel and Distributed Processing with Applications, 17th IEEE International Conference on Ubiquitous Computing and Communications, 8th IEEE International Conference on Big Data and Cloud Computing, 11t* (2019).

[21] ARORA, J., WESTRICK, S., AND ACAR, U. A. Provably space efficient parallel functional programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)"* (2021).

[22] ARVIND, NIKHIL, R. S., AND PINGALI, K. K. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst. 11*, 4 (Oct. 1989), 598–632.

[23] AWAD, A., BASU, A., BLAGODUROV, S., SOLIHIN, Y., AND LOH, G. H. Avoiding TLB shootdowns through self-invalidating TLB entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Sep. 2017), pp. 273–287.

[24] BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, L., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER,

R. S., ET AL. The nas parallel benchmarks. *The International Journal of Supercomputing Applications 5*, 3 (1991), 63–73.

[25] BASU, A., GANDHI, J., CHANG, J., HILL, M. D., AND SWIFT, M. M. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, ACM, pp. 237–248.

[26] BASU, A., HILL, M. D., AND SWIFT, M. M. Reducing memory reference energy with opportunistic virtual caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (2012), ISCA '12, pp. 297–308.

[27] BEAMER, S., ASANOVIC, K., AND PATTERSON, D. A. The GAP benchmark suite. *CoRR abs/1508.03619* (2015).

[28] BENDER, M. A., FINEMAN, J. T., GILBERT, S., AND LEISERSON, C. E. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures* (2004), pp. 133–144.

[29] BHATTACHARJEE, A. Preserving virtual memory by mitigating the address translation wall. *IEEE Micro 37*, 5 (Sep. 2017), 6–10.

[30] BHATTACHARJEE, A. Breaking the address translation wall by accelerating memory replays. *IEEE Micro 38*, 3 (May 2018), 69–78.

[31] BHATTACHARJEE, A., LUSTIG, D., AND MARTONOSI, M. Shared last-level TLBs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture* (Feb 2011), pp. 62–63.

[32] BHATTACHARJEE, A., AND MARTONOSI, M. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *18th International Conference on Parallel Architectures and Compilation Techniques* (September 2009), PACT'09, pp. 29–40.

[33] BIENIA, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[34] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (October 2008).

[35] BITTMAN, D., ALVARO, P., MEHRA, P., LONG, D. D. E., AND MILLER, E. L. Twizzler: a data-centric OS for non-volatile memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 65–80.

[36] BLELLOCH, G. E. Programming parallel algorithms. *Commun. ACM 39*, 3 (1996), 85–97.

[37] BLELLOCH, G. E., FINEMAN, J. T., GIBBONS, P. B., AND SHUN, J. Internally deterministic parallel algorithms can be fast. In *PPoPP '12* (2012), pp. 181–192.

[38] BLELLOCH, G. E., HARDWICK, J. C., SIPELSTEIN, J., ZAGHA, M., AND CHATTERJEE, S. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput. 21*, 1 (1994), 4–14.

[39] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (2009), OOPSLA '09, pp. 97–116.

[40] BONDHUGULA, U., RAMANUJAM, J., AND SADAYAPPAN, P. PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System. *PLDI 2008 - 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008).

[41] BOOS, K., LIYANAGE, N., IJAZ, R., AND ZHONG, L. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2020), pp. 1–19.

[42] BRYANT, R. E., KATZ, R. H., AND LAZOWSKA, E. D. Big-data computing: Creating revolutionary breakthroughs in commerce, science, and society. *White Paper, Computing Community Consortium Committee, Computing Research Association* (December 2008).

[43] BURROUGHS CORPORATION. The descriptor–a definition of the b5000 information processing system. Tech. Rep. BULLETIN 5000-20002-P, Burroughs Corporation, Detroit, MI. USA, February 1961.

[44] CALCIU, I., IMRAN, M. T., PUDDU, I., KASHYAP, S., MARUF, H. A., MUTLU, O., AND KOLLI, A. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS '21, Association for Computing Machinery, pp. 79–92.

[45] CASTRO, M., COSTA, M., MARTIN, J.-P., PEINADO, M., AKRITIDIS, P., DONNELLY, A., BARHAM, P., AND BLACK, R. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), pp. 45–58.

[46] CHEN, W.-K., BHANSALI, S., CHILIMBI, T., GAO, X., AND CHUANG, W. Profile-guided proactive garbage collection for locality optimization. *ACM SIGPLAN Notices 41*, 6 (2006), 332–340.

[47] CHENG, G.-I., FENG, M., LEISERSON, C. E., RANDALL, K. H., AND STARK, A. F. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures* (1998), SPAA '98.

[48] CHILIMBI, T. M., AND LARUS, J. R. Using generational garbage collection to implement cache-conscious data placement. *ACM SIGPLAN Notices 34*, 3 (1998), 37–48.

[49] CHILIMBI, T. M., AND LARUS, J. R. Data structure partitioning with garbage collection to optimize cache utilization, Nov. 20 2001. US Patent 6,321,240.

[50] Chisnall, D., Davis, B., Gudka, K., Brazdil, D., Joannou, A., Woodruff, J., Markettos, A. T., Maste, J. E., Norton, R., Son, S., et al. Cheri jni: Sinking the java security model into the c. *ACM SIGARCH Computer Architecture News 45*, 1 (2017), 569–583.

[51] Chisnall, D., Rothwell, C., Watson, R. N., Woodruff, J., Vadera, M., Moore, S. W., Roe, M., Davis, B., and Neumann, P. G. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, Association for Computing Machinery, p. 117–130.

[52] Clauss, P., and Meister, B. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *ACM SIGARCH computer architecture news 28*, 1 (2000), 11–19.

[53] Community, K. O. S. The webkit open source project, 1998.

[54] Corporation, S. P. E. Spec cpu2017, 2020.

[55] Cox, G., and Bhattacharjee, A. Efficient address translation for architectures with multiple page sizes. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[56] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. Virtual ghost: Protecting applications from hostile operating systems. *SIGARCH Comput. Archit. News 42*, 1 (Feb. 2014), 81–96.

[57] DAVIS, B., WATSON, R. N., RICHARDSON, A., NEUMANN, P. G., MOORE, S. W., BALDWIN, J., CHISNALL, D., CLARKE, J., FILARDO, N. W., GUDKA, K., ET AL. Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), pp. 379–393.

[58] DEIANA, E., AND CAMPANONI, S. Workload characterization of nondeterministic programs parallelized by stats. pp. 190–201.

[59] DEIANA, E., ST-AMOUR, V., DINDA, P., HARDAVELLAS, N., AND CAMPANONI, S. Unconventional parallelization of nondeterministic applications. *ACM SIGPLAN Notices 53* (03 2018), 432–447.

[60] DEIANA, E. A., ST-AMOUR, V., DINDA, P. A., HARDAVELLAS, N., AND CAMPANONI, S. Unconventional parallelization of nondeterministic applications. In *ASPLOS* (2018).

[61] DINDA, P., AND GULIANI, A. Dark shadows: User-level guest/host linux process shadowing. In *Proceedings of the 5th IEEE International Conference on Cloud Engineering* (April 2017).

[62] DISTEFANO, D. S., CALCAGNO, C., AND CHURCHILL, D. Detecting and remedying memory leaks caused by object reference cycles, May 21 2019. US Patent 10,296,314.

[63] DUCK, G. J., AND YAP, R. H. C. Effectivesan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)* (June 2018).

[64] EKMAN, M., STENSTRÖM, P., AND DAHLGREN, F. TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design* (2002), ISLPED '02, pp. 243–246.

[65] EVANS, D., AND LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE Software 19*, 1 (Jan 2002), 42–51.

[66] FAN, D., TANG, Z., HUANG, H., AND GAO, G. R. An energy efficient TLB design methodology. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design* (2005), ISLPED '05, pp. 351–356.

[67] FENG, M., AND LEISERSON, C. E. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems 32*, 3 (1999), 301–326.

[68] FILARDO, N. W., GUTSTEIN, B. F., WOODRUFF, J., AINSWORTH, S., PAUL-TRIFU, L., DAVIS, B., XIA, H., NAPIERALA, E. T., RICHARDSON, A., BALDWIN, J., ET AL. Cornucopia: Temporal safety for cheri heaps. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 608–625.

[69] FINEMAN, J. T. Provably good race detection that runs in parallel. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2005.

[70] FLUET, M., RAINEY, M., REPPY, J., AND SHAW, A. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming 20*, 5-6 (2011), 1–40.

[71] FRIGO, M., HALPERN, P., LEISERSON, C. E., AND LEWIN-BERLIN, S. Reducers and other Cilk++ hyperobjects. In *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures* (2009), pp. 79–90.

[72] GANDHI, J., KARAKOSTAS, V., AYAR, F., CRISTAL, A., HILL, M. D., MCKINLEY, K. S., NEMIROVSKY, M., SWIFT, M. M., AND ÜNSAL, O. S. Range translations for fast virtual memory. *IEEE Micro 36*, 3 (May 2016), 118–126.

[73] GARCIA, S., JEON, D., LOUIE, C. M., AND TAYLOR, M. B. Kremlin: rethinking and rebooting gprof for the multicore age. *ACM SIGPLAN Notices 46*, 6 (2011), 458–469.

[74] Native client. `https://developer.chrome.com/native-client`.

[75] GORINE, A., AND KRIVOLAPOV, A. A kernel mode database system for high performance applications. Tech. rep., McObject.

[76] GU, J., LEE, Y., ZHANG, Y., CHOWDHURY, M., AND SHIN, K. G. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, Mar. 2017), NSDI '17, USENIX Association, pp. 649–667.

[77] GUATTO, A., WESTRICK, S., RAGHUNATHAN, R., ACAR, U. A., AND FLUET, M. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018* (2018), pp. 81–93.

[78] HALSTEAD, JR., R. H. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (1984), LFP '84, ACM, pp. 9–17.

[79] HAMMACHER, C., STREIT, K., HACK, S., AND ZELLER, A. Profiling java programs for parallelism. In *2009 ICSE Workshop on Multicore Software Engineering* (2009), IEEE, pp. 49–55.

[80] HAMMOND, K. Why parallel functional programming matters: Panel statement. In *Reliable Software Technologies - Ada-Europe 2011 - 16th Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, UK, June 20-24, 2011. Proceedings* (2011), pp. 201–205.

[81] HARDAVELLAS, N., FERDMAN, M., FALSAFI, B., AND AILAMAKI, A. Toward dark silicon in servers. *IEEE Micro 31*, 4 (July-August 2011), 6–15.

[82] HARIA, S., HILL, M. D., AND SWIFT, M. M. Devirtualizing memory in heterogeneous systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, ACM, pp. 637–650.

[83] HE, Y., LEISERSON, C. E., AND LEISERSON, W. M. The cilkview scalability analyzer. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures* (2010), pp. 145–156.

[84] HUANG, X., BLACKBURN, S. M., MCKINLEY, K. S., MOSS, J. E. B., WANG, Z., AND CHENG, P. The garbage collection advantage: improving program locality. *ACM SIGPLAN Notices 39*, 10 (2004), 69–80.

[85] HUNT, G., AIKEN, M., FÄHNDRICH, M., HAWBLITZEL, C., HODSON, O., LARUS, J., LEVI, S., STEENSGAARD, B., TARDITI, D., AND WOBBER, T. Sealing OS processes to improve dependability and safety. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys)* (2007), pp. 341–354.

[86] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the software stack. *SIGOPS Operating Systems Review 41*, 2 (Apr. 2007), 37–49.

[87] INTEL, I. R. math kernel library.

[88] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2019.

[89] JOHNSON, R., PEARSON, D., AND PINGALI, K. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (1994).

[90] JR., R. L. B., HEUMANN, S., HONARMAND, N., ADVE, S. V., ADVE, V. S., WELC, A., AND SHPEISMAN, T. Safe nondeterminism in a deterministic-by-default parallel

language. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 535–548.

[91] JUAN, T., LANG, T., AND NAVARRO, J. J. Reducing TLB power requirements. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design* (1997), ISLPED '97, pp. 196–201.

[92] KADAYIF, I., NATH, P., KANDEMIR, M., AND SIVASUBRAMANIAM, A. Compiler-directed physical address generation for reducing dTLB power. In *Proceedings of the IEEE International Symposium on the Performance Analysis of Systems and Software (ISPASS)* (March 2004), pp. 161–168.

[93] KADAYIF, I., SIVASUBRAMANIAM, A., KANDEMIR, M., KANDIRAJU, G., AND CHEN, G. Generating physical addresses directly for saving instruction TLB energy. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (November 2002), pp. 185–196.

[94] KANDEMIR, M., CHOUDHARY, A., RAMANUJAM, J., SHENOY, N., AND BANERJEE, P. Enhancing spatial locality via data layout optimizations. In *European Conference on Parallel Processing* (1998), Springer, pp. 422–434.

[95] KANDEMIR, M., CHOUDHARY, A., SHENOY, N., BANERJEE, P., AND RAMANUJAM, J. A hyperplane based approach for optimizing spatial locality in loop nests. In *Proceedings of the 12th international conference on Supercomputing* (1998), pp. 69–76.

[96] KARAKOSTAS, V., GANDHI, J., CRISTAL, A., HILL, M., MCKINLEY, K., NEMIROVSKY, M., SWIFT, M., AND UNSAL, O. Energy-efficient address translation. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), pp. 631–643.

[97] KENNEDY, K., MCKINLEY, K. S., AND TSENG, C.-W. Interactive parallel programming using the parascope editor. *IEEE Transactions on Parallel & Distributed Systems*, 3 (1991), 329–341.

[98] KIM, M., KIM, H., AND LUK, C.-K. Sd3: A scalable approach to dynamic data-dependence profiling. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (2010), IEEE, pp. 535–546.

[99] KOCHER, P., HORN, J., FOGH, A., , GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)* (2019).

[100] KULKARNI, M., BURTSCHER, M., INKULU, R., PINGALI, K., AND CASÇAVAL, C. How much parallelism is there in irregular applications? *ACM sigplan notices 44*, 4 (2009), 3–14.

[101] KUPER, L., AND NEWTON, R. R. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing* (2013), ACM, pp. 71–84.

[102] KUPER, L., TODD, A., TOBIN-HOCHSTADT, S., AND NEWTON, R. R. Taming the parallel effect zoo: Extensible deterministic parallelism with lvish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), PLDI '14, pp. 2–14.

[103] LACHAIZE, R., LEPERS, B., AND QUÉMA, V. Memprof: A memory profiler for numa multicore systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (USA, 2012), USENIX ATC'12, USENIX Association, p. 5.

[104] LAGAR-CAVILLA, A., AHN, J., SOUHLAL, S., AGARWAL, N., BURNY, R., BUTT, S., CHANG, J., CHAUGULE, A., DENG, N., SHAHID, J., THELEN, G., YURTSEVER, K. A., ZHAO, Y., AND RANGANATHAN, P. Software-defined far memory in warehouse-scale computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, Apr. 2019), ASPLOS '19, Association for Computing Machinery, pp. 317–330.

[105] LARUS, J. R. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems 4*, 7 (1993), 812–826.

[106] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), IEEE, pp. 75–86.

[107] LEVY, A., CAMPBELL, B., GHENA, B., GIFFIN, D. B., PANNUTO, P., DUTTA, P., AND LEVIS, P. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017), p. 234–251.

[108] LEVY, H. *Capability-Based Computer Systems*. Digital Press, 1984.

[109] LI, P., MARLOW, S., PEYTON JONES, S. L., AND TOLMACH, A. P. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007* (2007), pp. 107–118.

[110] LI, Z., ATRE, R., HUDA, Z., JANNESARI, A., AND WOLF, F. Unveiling parallelization opportunities in sequential programs. *Journal of Systems and Software* (2016).

[111] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)* (2018).

[112] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices 40*, 6 (2005), 190–200.

[113] MA, J., WANG, W., NELSON, A., CUEVAS, M., HOMERDING, B., LIU, C., HUANG, Z., CAMPANONI, S., HALE, K., AND DINDA, P. Paths to openmp in the kernel. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC) (Supercomputing)* (November 2021).

[114] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. 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 2013)* (Mar. 2013), pp. 461–472.

[115] MARKETTOS, A. T., BALDWIN, J., BUKIN, R., NEUMANN, P. G., MOORE, S. W., AND WATSON, R. N. Position paper: Defending direct memory access with cheri capabilities.

[116] MATNI, A., DEIANA, E. A., SU, Y., GROSS, L., GHOSH, S., APOSTOLAKIS, S., XU, Z., TAN, Z., CHATURVEDI, I., AUGUST, D. I., AND CAMPANONI, S. NOELLE Offers Empowering LLvm Extensions. In *International Symposium on Code Generation and Optimization, 2022. CGO 2022.* (2022).

[117] MAZZINGHI, A., SOHAN, R., AND WATSON, R. N. Pointer provenance in a capability architecture. In *10th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2018)* (2018).

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

[119] MELLOR-CRUMMEY, J. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91* (1991), pp. 24–33.

[120] MEMARIAN, K., GOMES, V. B., DAVIS, B., KELL, S., RICHARDSON, A., WATSON, R. N., AND SEWELL, P. Exploring c semantics and pointer provenance. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–32.

[121] MISAILOVIC, S., KIM, D., AND RINARD, M. Parallelizing sequential programs with statistical accuracy tests. *ACM Trans. Embed. Comput. Syst.* (2013).

[122] MIUCIN, S., BRADY, C., AND FEDOROVA, A. End-to-end memory behavior profiling with dinamite. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, ACM, pp. 1042–1046.

[123] MIUCIN, S., BRADY, C., AND FEDOROVA, A. End-to-end memory behavior profiling with dinamite. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, Association for Computing Machinery, p. 1042–1046.

[124] NARAYANAN, V., HUANG, T., DETWEILER, D., APPEL, D., LI, Z., ZELLWEGER, G., AND BURTSEV, A. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2020), pp. 21–39.

[125] NECULA, G. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 1997)* (January 1997).

[126] Necula, G., and Lee, P. Proof-carrying code. Tech. Rep. CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, 1996.

[127] Necula, G., and Lee, P. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI 1996)* (October 1996).

[128] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices* (2007).

[129] Neumann, P. G. Fundamental trustworthiness principles. *New Solutions for Cybersecurity* (2018).

[130] Norouzi, M., Wolf, F., and Jannesari, A. Automatic construct selection and variable classification in OpenMP. *Proceedings of the International Conference on Supercomputing* (2019).

[131] Novakovic, S., Daglis, A., Bugnion, E., Falsafi, B., and Grot, B. Scale-out numa. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, Feb. 2014), ASPLOS '14, Association for Computing Machinery, pp. 3—-18.

[132] Omni OpenMP Compiler Group, University of Versailles Saint Quentin en Yvlines. Nas parallel benchmarks 3.0—unofficial openmp c version. `https://github.com/benchmark-subsetting/NPB3.0-omp-C`, 2014.

[133] OSKIN, M., AND LOH, G. H. A software-managed approach to die-stacked DRAM. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)* (2015), PACT '15, pp. 188–200.

[134] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, Feb. 2019), NSDI '19, USENIX Association, pp. 361––377.

[135] PARASAR, M., BHATTACHARJEE, A., AND KRISHNA, T. SEESAW: Using superpages to improve VIPT caches. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (2018), pp. 193–206.

[136] PESTEREV, A., ZELDOVICH, N., AND MORRIS, R. T. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, Association for Computing Machinery, p. 335–348.

[137] PETZOLD, C. *Programming Windows*. Microsoft Press, 1988.

[138] PEYTON JONES, S. L., LESHCHINSKIY, R., KELLER, G., AND CHAKRAVARTY, M. M. T. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS* (2008), pp. 383–414.

[139] PUTTASWAMY, K., AND LOH, G. H. Thermal analysis of a 3D die-stacked high-performance microprocessor. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI* (2006), GLSVLSI '06, pp. 19–24.

[140] RADIN, G. The 801 minicomputer. *IBM Journal of Research and Development 27*, 3 (May 1983), 237–246. Originally published at ASPLOS I and republished in ACM SIGARCH Computer Architecture News, Volume 10, Number 2, March 1982.

[141] RAGHESH, A. A Framework for Automatic OpenMP Code Generation.

[142] RAGHUNATHAN, R., MULLER, S. K., ACAR, U. A., AND BLELLOCH, G. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2016), ICFP 2016, ACM, pp. 392–406.

[143] RAMAN, R., ZHAO, J., SARKAR, V., VECHEV, M., AND YAHAV, E. Efficient data race detection for async-finish parallelism. In *Runtime Verification*, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, Eds., vol. 6418 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 368–383.

[144] RAMAN, R., ZHAO, J., SARKAR, V., VECHEV, M., AND YAHAV, E. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2012), PLDI '12, pp. 531–542.

[145] ROUNTEV, A., VAN VALKENBURGH, K., YAN, D., AND SADAYAPPAN, P. Understanding parallelism-inhibiting dependences in sequential java programs. In *2010 IEEE International Conference on Software Maintenance* (2010), IEEE, pp. 1–9.

[146] RUAN, Z., SCHWARZKOPF, M., AGUILERA, M. K., AND BELAY, A. AIFM: High-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation* (Nov. 2020), OSDI '20, USENIX Association, pp. 315–332.

[147] RUBIN, S., BODÍK, R., AND CHILIMBI, T. An efficient profile-analysis framework for data-layout optimizations. *ACM SIGPLAN Notices 37*, 1 (2002), 140–153.

[148] SALZ, J. H., MIRCHANDANEY, R., AND CROWLEY, K. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput. 40*, 5 (May 1991), 603–612.

[149] SCHÄLING, B. *The boost C++ libraries*. Boris Schäling, 2011.

[150] SEHR, D., MUTH, R., BIFFLE, C. L., KHIMENKO, V., PASKO, E., YEE, B., SCHIMPF, K., AND CHEN, B. Adapting software fault isolation to contemporary cpu architectures.

[151] SEMICONDUCTOR INDUSTRY ASSOCIATION AND SEMICONDUCTOR RESEARCH CORPORATION. Rebooting the IT revolution: A call to action. *NSF Workshop Report* (September 2015).

[152] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: A fast address sanity checker. *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012* (2012).

[153] SHIN, S., COX, G., OSKIN, M., LOH, G. H., SOLIHIN, Y., BHATTACHARJEE, A., AND BASU, A. Scheduling page table walks for irregular GPU applications. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (2018), ISCA '18, pp. 180–192.

[154] SHUF, Y., GUPTA, M., FRANKE, H., APPEL, A., AND SINGH, J. P. Creating and preserving locality of java applications at allocation and garbage collection times. *ACM SIGPLAN Notices 37*, 11 (2002), 13–25.

[155] SLEATOR, D. D., AND TARJAN, R. E. Self-adjusting binary search trees. *Journal of the ACM 32*, 3 (July 1985), 652–686.

[156] SODANI, A. Race to exascale: Opportunities and challenges. In *Keynote at the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (June 2011), MICRO 44.

[157] SPOONHOWER, D. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, May 2009.

[158] STOLLER, S. D., CARBIN, M., ADVE, S. V., AGRAWAL, K., BLELLOCH, G. E., DAN, STANZIONE, YELICK, K. A., AND ZAHARIA, M. A. Future directions for parallel and distributed computing: Spx 2019 workshop report.

[159] SUCHY, B., CAMPANONI, S., HARDAVELLAS, N., AND DINDA, P. Carat: A case for virtual memory through compiler- and runtime-based address translation. In *Proceedings*

of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (June 2020), p. 329–345.

[160] SUCHY, B., GHOSH, S., NELSON, A., HUANG, Z., KERSNAR, D., CHAI, S., CUEVAS, M., BERNAT, A., CHAUDHARY, G., HARDAVELLAS, N., ET AL. Carat cake: Replacing paging via compiler/kernel cooperation. ASPLOS.

[161] SUCHY, B., HARDEVELLAS, N., CAMPANONI, S., AND DINDA, P. Carat: A case for virtual memory through compiler- and runtime-based address translation. In *Conference on Programming Language Design and Implementation (PLDI)* (2020).

[162] TAZAKI, H. An introduction of library operating system for linux (libos).

[163] TIRMAZI, M., BARKER, A., DENG, N., HAQUE, M. E., QIN, Z. G., HAND, S., HARCHOL-BALTER, M., AND WILKES, J. Borg: the next generation. In *Proceedings of the 15th European Conference on Computer Systems* (New York, NY, USA, Apr. 2020), EuroSys '20, Association for Computing Machinery.

[164] UTTERBACK, R., AGRAWAL, K., FINEMAN, J. T., AND LEE, I. A. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016* (2016), pp. 83–94.

[165] VAN DER SPOEL, D., LINDAHL, E., HESS, B., GROENHOF, G., MARK, A. E., AND BERENDSEN, H. J. Gromacs: fast, flexible, and free. *Journal of computational chemistry 26*, 16 (2005), 1701–1718.

[166] VILLAVIEJA, C., KARAKOSTAS, V., VILANOVA, L., ETSION, Y., RAMIREZ, A., MENDELSON, A., NAVARRO, N., CRISTAL, A., AND UNSAL, O. S. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques* (Oct 2011), pp. 340–349.

[167] VON PRAUN, C., BORDAWEKAR, R., AND CASCAVAL, C. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (2008), pp. 185–196.

[168] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP 1993)* (December 1993).

[169] WATSON, R. N., NORTON, R. M., WOODRUFF, J., MOORE, S. W., NEUMANN, P. G., ANDERSON, J., CHISNALL, D., DAVIS, B., LAURIE, B., ROE, M., ET AL. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro 36*, 5 (2016), 38–49.

[170] WESTRICK, S., YADAV, R., FLUET, M., AND ACAR, U. A. Disentanglement in nested-parallel programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)"* (2020).

[171] WOODRUFF, J., WATSON, R. N. M., CHISNALL, D., MOORE, S. W., ANDERSON, J., DAVIS, B., LAURIE, B., NEUMANN, P. G., NORTON, R., AND ROE, M. The cheri

capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (2014), pp. 457–468.

[172] WU, P., KEJARIWAL, A., AND CAŞCAVAL, C. Compiler-driven dependence profiling to guide program parallelization. In *International Workshop on Languages and Compilers for Parallel Computing* (2008), Springer, pp. 232–248.

[173] XIA, H., WOODRUFF, J., AINSWORTH, S., FILARDO, N. W., ROE, M., RICHARDSON, A., RUGG, P., NEUMANN, P. G., MOORE, S. W., WATSON, R. N., ET AL. Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (2019), pp. 545–557.

[174] YAN, Z., LUSTIG, D., NELLANS, D., AND BHATTACHARJEE, A. Translation ranger: Operating system support for contiguity-aware TLBs. In *Proceedings of the ACM/IEEE 46th International Symposium on Computer Architecture* (2019), ISCA '19, pp. 698–710.

[175] YAN, Z., VESELÝ, J., COX, G., AND BHATTACHARJEE, A. Hardware translation coherence for virtualized systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ISCA '17, pp. 430–443.

[176] YOON, W., OH, J., OK, J., MOON, S., AND KWON, Y. DiLOS: Adding performance to paging-based memory disaggregation. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2021), ApSys '21, Association for Computing Machinery, pp. 70–78.

[177] YURYEV, V. Apple's m1 ultra comes with a 32mb tlb bottleneck.

[178] ZAGIEBOYLO, D., SUH, G. E., AND MYERS, A. C. The cost of software-based memory management without virtual memory. *CoRR abs/2009.06789* (2020).

[179] ZHANG, X., NAVABI, A., AND JAGANNATHAN, S. Alchemist: A transparent dependence distance profiling infrastructure. In *2009 International Symposium on Code Generation and Optimization* (2009), IEEE, pp. 47–58.