CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation

Brian Suchy
Northwestern University
Evanston, IL, USA
brian@briansuchy.com

Souradip Ghosh
Northwestern University
Evanston, IL, USA

Drew Kersnar
Northwestern University
Evanston, IL, USA

Siyuan Chai
Northwestern University
Evanston, IL, USA

Zhen Huang
Northwestern University
Evanston, IL, USA

Aaron Nelson
Northwestern University
Evanston, IL, USA

Michael Cuevas
Northwestern University
Evanston, IL, USA

Alex Bernat
Northwestern University
Evanston, IL, USA

Gaurav Chaudhary
Northwestern University
Evanston, IL, USA

Nikos Hardavellas
Northwestern University
Evanston, IL, USA
nikos@northwestern.edu

Simone Campanoni
Northwestern University
Evanston, IL, USA
simonec@eecs.northwestern.edu

Peter Dinda
Northwestern University
Evanston, IL, USA
pdinda@northwestern.edu

ABSTRACT

Virtual memory, specifically paging, is undergoing significant innovation due to being challenged by new demands from modern workloads. Recent work has demonstrated an alternative software-only design that can result in simplified hardware requirements, even supporting purely physical addressing. While we have made the case for this Compiler- And Runtime-based Address Translation (CARAT) concept, its evaluation was based on a user-level prototype. We now report on incorporating 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 allows significant new benefits for systems including energy savings, larger L1 caches, and arbitrary granularity memory management.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507771

CCS CONCEPTS

• **Software and its engineering** → **Operating systems, compilers**; **Runtime environments**; • **Blended systems**;

KEYWORDS

virtual memory, memory management, kernel, runtime

ACM Reference Format:
Brian Suchy, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, Simone Campanoni, and Peter Dinda. 2022. CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507771

1 INTRODUCTION

Virtual memory, specifically address translation implemented with paging, is deeply embedded in today's systems at all levels, but particularly within the hardware and the kernel. As we have known since the 1960s [**?**], virtual memory solves numerous problems. This includes providing a simplifying memory abstraction for programmers, protecting the kernel from processes and processes from each other, and extending physical address space via swapping to/from storage. Its most popular form, paging, also provides a natural unit for memory management.

Unfortunately, paging[1] is not without cost. Paging *requires* hardware/software codesign spanning the hardware directly on the access path to main memory and the deepest levels of the kernel. The hardware structures supporting the traditional address translation model (per-core DTLBs, ITLBs, STLBs, separate structures for different page sizes, nested TLBs, quad pagewalkers, walker caches)

[1] And its kissing cousin, segmentation.

together require almost as much area as L1 caches [**?**]. A TLB consumes a significant energy [**? ? ? ?**], and is a thermal hot spot [**?**]. Early studies find TLB power consumption is as high as 15-17% of the chip's power [30**?**]. Industry reports that TLBs consume up to 13% of a core's power [**? ?**], with later studies estimating that TLBs are responsible for 20-38% of L1 cache energy consumption [**? ?**]. Distributed TLBs are no panacea either, with significant overheads to keep them coherent [**? ? ? ? ? ?**].

As workloads have changed and become larger, we see more strain on hardware. The increasingly complex design of TLBs, pagewalk caches, pagewalkers, etc, face increasing concern [**? ? ? ? ? ?**]. This has sparked considerable innovation attempting to extend the paging model. Also note that paging hardware is increasingly becoming coupled to other aspects of the memory system. Modern L1 caches are virtually indexed, allowing a cache lookup to start in parallel with a TLB lookup. This coupling limits the number of sets in the L1 cache, guarding against synonyms (i.e., different virtual addresses mapped to the same physical location), since the cache is indexed by the LSBs common to physical and virtual address [7**? ?**]. Removing this limitation, which CARAT implementations do, might allow L1 caches to provide higher performance without needing to increase associativity.

Modern processors employ these hardware structures despite their overheads because performance *depends* on fast address translation. But, what if we can get by *without* this hardware? It is tantalizing to consider machines which do not need hardware such as TLBs, pagewalk caches, or pagewalkers, or at least can power it down when the kernel desires.[2] Compiler technology has been advancing [**? ? ? ? ? ? ? ?**] and becoming more readily available through vehicles such as LLVM [**?**].

Historically, another approach to solving the problems that paging solves, namely protection and mapping (address binding), has been through purely software-based memory management. This approach has been limited to managed languages, where it serves a different purpose than paging in a general-purpose OS which can run any arbitrary code in a user process. However, we have recently made a case for Compiler- And Runtime-based Address Translation (CARAT), an approach to *general-purpose* software-based memory management based on a compiler/kernel codesign [53].

CARAT can operate without hardware support, instead using compiler transformations along with a runtime system to maintain a precise view of a running program's memory at the granularity of program objects. CARAT allows programs to run with physical addressing. Because the transformations operate at the level of the compiler intermediate representation (the LLVM IR in our prototype), CARAT is applicable to many unmanaged languages with few restrictions. Performance and memory overheads for CARAT's memory object and pointer tracking, memory access protection checks, and memory object migration were shown to be very low for workloads selected from the Mantevo, NAS, PARSEC, and SPEC2017 benchmark suites. We claim that CARAT is feasible and workable *now* due to advances in compiler technology.

A weakness of the case is that the CARAT prototype was implemented/evaluated at user level, basically operating on top of paging within a Linux process. It did not address challenges posed by a kernel-level implementation of CARAT, like building abstractions such as processes on top of CARAT. In this paper, we address this limitation of our previous work.

We develop Compiler- And Runtime-based Address Translation for CollAborative Kernel Environments (CARAT CAKE), a new, kernel-based implementation of the CARAT concept. More specifically, we extend 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 our 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.

We describe the design and implementation of the CARAT CAKE prototype, drawing observations about key challenges. Two of these are creating necessary compiler transformations that will work on a kernel-level codebase, and achieving separate compilation of the kernel and user-level codebases despite their extremely tight coupling at runtime.

We evaluate our prototype along two dimensions. First, we report on the overall effort involved in implementing the system, starting from a known compiler analysis and optimization framework and a kernel without any deeply embedded assumptions of paging. Second, we report on the performance overheads of the CARAT CAKE system, in comparison with a high performance paging-based implementation in Nautilus and Linux. We find that CARAT CAKE is a viable alternative to paging.

We make the following contributions:

- We show how to (and do) extend the previously published CARAT concept into CARAT CAKE, a full-blown alternative to paging within a kernel.
- We describe the design and implementation of the CARAT CAKE prototype, drawing observations about the challenges of this form of memory management within the kernel.
- We create a Linux-compatible process abstraction on top of CARAT CAKE, drawing observations about the challenges of separate compilation and kernel support. We also implement the process abstraction using a paging codebase that leverages modern x64 hardware support features for improving paging performance.
- We evaluate CARAT CAKE's overheads in comparison with paging (and with paging on Linux), as well as the software engineering effort involved. Our evaluation shows that CARAT CAKE is a viable alternative to paging.
- We describe new benefits made possible with CARAT CAKE centered around microarchitecture and software benefits.

[2]Note that every current x86 processor includes the ability to deactivate paging when running in 32 bit mode.

2 SOFTWARE, TESTBED, AND BENCHMARKS

Our prototype is built on LLVM, WLLVM, GLLVM, NOELLE, and the Nautilus kernel. We compare paging and CARAT CAKE within Nautilus (and with Linux) using NAS and PARSEC benchmarks.

2.1 Software

2.1.1 LLVM. LLVM [**?**] is a widely-used compilation framework in academia and industry that enables sophisticated code analyses and transformations. In this work, we use the framework to implement CARAT CAKE at the level of the LLVM intermediate representation (LLVM-IR, static single assignment (SSA) form), within the "middle-end" of LLVM. The middle-end provides the API to develop custom program-wide code analyses and transformations. We leverage, for example, scalar evolution analysis and alias analysis. While Clang, which targets C and C++ (including OpenMP), is our front-end, by working in the middle-end, CARAT CAKE can be used with the numerous other language front-ends to LLVM. We use Clang/LLVM 9.0.0 for all code, both kernel and user.

2.1.2 WLLVM/GLLVM. Whole-Program LLVM [50] extends LLVM compilation to aggregate all the LLVM bitcode in a project that uses separate compilation. This aggregation gives us two key capabilities. Firstly, the CARAT CAKE compiler transformations must target entire applications and the entire kernel, and this interacts badly with specialized build models. Nautilus, in particular, builds itself using an incremental linker, where transformations are more difficult to apply, rebuild, and link together. WLLVM allows the CARAT CAKE transformations to run on the entire kernel at once without having to consider more complex rebuilding and linking steps. Secondly, a whole-kernel bitcode source allows the CARAT CAKE transformations to consider the entire kernel as one module, enabling a whole-kernel transformation and optimization. The same technique is used for user-space compilation. An alternative tool, GLLVM [28], can be used as a replacement for WLLVM. Each is version 1.3.0.[3]

2.1.3 NOELLE. CARAT CAKE is heavily based on compiler analysis and transformation. In our work, we build on a powerful new analysis framework, NOELLE [**?**], which provides numerous high-level abstractions on top of LLVM that allow for aggressive analyses and transformations. Of the many abstractions that NOELLE provides, CARAT CAKE extensively utilizes NOELLE's data flow engine, NOELLE's invariants and induction variables, and finally NOELLE's dependence analysis to generate the Program Dependence Graph (PDG) of the program being compiled. Because the overhead of CARAT CAKE is inversely related to the accuracy of the PDG, we configured NOELLE to generate the most accurate PDG. NOELLE does so by combining many alias analyses, including those provided by SCAF [**?**] and SVF [**?**]. All code in the kernel and at user level is analyzed using NOELLE. As compiler analysis improves, so does the argument for CARAT.

2.1.4 Nautilus Kernel Framework. Nautilus [24] is a publicly available open-source kernel codebase that currently runs directly on x64 NUMA hardware, including Xeon Phi. It comprises over 331K lines of code as measured by `sloccount`. Nautilus was designed with the goal of supporting hybrid run-times (HRTs). An HRT is a mashup of an extremely lightweight OS kernel framework, such as Nautilus, and a parallel run-time system [22, 23]. Nautilus can help a parallel run-time ported to an HRT achieve very high performance by providing streamlined kernel primitives such as synchronization and threading facilities. It provides the minimal set of features needed to support a *tailored* parallel run-time environment, avoiding general purpose features that inhibit scalability.

Nautilus has a range of features that help make the execution of an HRT faster and more predictable. Identity-mapped paging with the largest possible page size is used. All addresses are mapped at boot, and there is no swapping or page movement of any kind. As a consequence, TLB misses are extremely rare, and, indeed, if the TLB entries can cover the physical address space of the machine, do not occur at all after startup. There are no page faults. All memory management, including for NUMA, is explicit and allocations are done with buddy system allocators [31] that are selected based on the target zone. For threads that are bound to specific CPUs, essential thread (e.g., context, stack) and scheduler state is guaranteed to always be in the most desirable zone. The core set of I/O drivers developed for Nautilus have interrupt handler logic with deterministic path lengths. Finally, interrupts are fully steerable, and thus can largely be avoided on most hardware threads. Application benchmark speedups from 20–40% over user-level execution on Linux have been demonstrated, while benchmarks show that primitives such as thread management and event signaling are orders of magnitude faster [24, 25, 36, 49]. In essence, Nautilus provides a barebones memory management substrate that we build upon.

In order to support this work, we added an address space, or ASpace, abstraction to Nautilus. By default, threads and interrupts execute within the "base" ASpace, the original identity-mapped model established at boot, which effectively is the physical address space of the machine. A thread with a different associated ASpace has that ASpace switched in/out on a context switch. An ASpace is conceptually a memory map of regions, similar to a Linux `mm_struct`, but designed without the assumption of paging. This allows radically different implementations to be plugged in, such as paging and CARAT CAKE, as we describe later. Allocators can also be plugged in above the system-level buddy allocators. All memory in the system is managed in a unified way.

2.2 Testbed and Benchmarks

Testing and performance measurements for this paper are done on a Colfax Ninja Xeon Phi server, which is based on a Supermicro K1SPE motherboard that includes a 1.3 GHz Intel Xeon Phi 7210 (64 cores, 256 hardware threads) mated to 16 GB of MCDRAM and 96 GB of DRAM. Both Nautilus and Linux are booted directly on this platform. The Linux kernel involved is version 5.8.0. To evaluate our work, we use class B benchmarks from the NAS 3.0 Application Benchmark Suite [5, 29] as ported to C+OpenMP [46]. We also include two benchmarks, Streamcluster and Blackscholes, from the Parsec 3.0 [8] benchmark suite.

[3] Note that separate compilation can extend to libraries—each library can include the bitcode along with the object code. In fact, in modern MacOS, applications and libraries are already supplied in bitcode form so they can be customized for the particular target machine at install time.

Table 1: Terminology for CARAT.

Term	Meaning
Allocation	Any memory allocation that a program makes. This includes heap allocations (`mallocs`), global variables, the stack, etc.
Free	A deallocation of an Allocation
Escape	Any reference to an Allocation stored outside of the initial Allocation pointer.
Contained Escape	An Escape that that is stored within an Allocation. This represents a subset of the Escapes within a program.
Guard	A protection check before a memory access that ensures proper access permissions.
Memory Region	A contiguous block of memory addresses.
Allocation Tracking	Injected code that will keep track of every Allocation (and Free) of a program.
Escape Tracking	Injected code that will keep track of every Escape of a program.
Guard Injection	Injected instruction that performs a Guard against a memory reference within the program.

3 CARAT CONCEPTS AND PRIOR RESULTS

Suchy *et al.* [53] argue that a modern OS running programs written in unmanaged languages can achieve performant memory management and protection without hardware support. A naive approach to purely software-based memory management and protection is, of course, destined to be horrifically slow. A major claim is that compiler analysis technology has advanced to the point where high performance in an approach like CARAT is possible. This is supported by extensive measurements in the CARAT paper that capture run-time overheads both for normal operation and when objects are moved.

3.1 Concepts

Goal and Security Model: The goal of CARAT is to provide the same capabilities as the traditional paging model. In terms of protection this means that a user process may only access physical addresses chosen by the kernel, and then only read/write/execute data at those addresses as the kernel sees fit. Execution can only enter the kernel via well-defined entry points associated with exceptions, interrupts, and system calls. When executing within the kernel, any address can be accessed (monolithic kernel model). The kernel may move data from one physical address to another at any time, analogously to how a paging-based kernel can change a page table at any time, albeit now at an arbitrary granularity.

Trusted Computing Base (TCB): The CARAT approach is based on a trust relationship between the kernel and the compiler toolchain that builds the kernel and the user programs. This appears to increase the size of the software TCB, but, of course, the compiler toolchain is already trusted to build the kernel. We assert that the true expansion is in the analysis and transformations added to the compiler toolchain specifically to support CARAT, and that this is offset by a decrease in the size and complexity of the hardware TCB.

Terminology: Table 1 summarizes the terminology used in CARAT. These and the following explanation apply equally to CARAT and the work described in this paper. CARAT combines compiler-injected and optimized code with a runtime system to implement memory management and protection. More specifically, the compiler transforms the intermediate representation (IR) of all code,[4] injecting runtime calls that match the memory management features provided by paging.

Similar to a garbage collector in a managed language, Allocations and Escapes are tracked. Unlike such a garbage collector, CARAT tracking also includes statics, stacks, etc.—*all* allocations produced from the unmanaged language and visible at the IR level are tracked. The compiler injects code that works at the granularity of these Allocations (instead of page granularity) by adding calls to the runtime at the site of each Allocation, Free, and Escape.

The compiler also introduces software protection checks, Guards, that replace hardware-level checks against page table entries. Conceptually, each memory access at the IR level has a Guard introduced before it, which calls into the runtime to determine if the memory address being accessed by the program is allowed. Protection of the stack from control-flow-based reads and writes is achieved by introducing Guards before calls. The runtime compares the address and the access mode with the protections of the memory region to which the address belongs. With appropriate CARAT-specific compiler optimizations, it is possible to safely avoid most of these direct protection checks. This is central to good performance.[5]

When needing to move Allocations, for example to compact or defragment physical memory, CARAT uses tracking information maintained by the runtime to patch all relevant Escapes. Conceptually, this works like a stop-and-copy or generational garbage collector, but again is happening in the context of an unmanaged language, and garbage is not freed.

3.2 User-Level Prototype and Limitations

The original CARAT paper [53] developed a prototype of this system within the context of LLVM. Our runtime system is a user-level library in C++. To measure the overheads of tracking and data movement the CARAT prototype uses signals: linking a transformed Linux user program with the library allows it to respond to an external signal that prompts it to move an Allocation or change the protections of a Region. A special mode aggregates Allocation-level movements to emulate page-sized data movements for comparison with Linux.

For guarding each memory access (load, store and call instruction at the LLVM IR level), which seems an infeasible task, the compiler toolchain includes specially-developed custom data flow analysis, loop invariant analysis, and induction variable analysis to elide redundant guard injections, which dramatically reduce the number of dynamically-encountered guards while still maintaining protection. These custom analyses statically guarantee a memory access is safe (removing the guard completely), relocate the guard to reduce redundant checking of a memory address, or conservatively place the guard before the access when there is no other option.

The CARAT prototype considers several guard runtime options. The simplest is an inlined single region bounds check. A variant of this assumes hardware-accelerated bounds checking (Intel MPX).

[4] Assembly code must be transformed by hand and attested to. Undefined behavior in the source code has been resolved at this point—LLVM IR has no undefined behavior.
[5] Protection for instruction fetches is based on static control-flow integrity checking, position independence, specialized linking, and load-time checks. Self-modifying code is not supported.

The paper also provides the overheads for several data structures supporting an arbitrary number of regions.

Performance of User-level Prototype: Measurements of the prototype showed that tracking introduces a 2% performance overhead, while the protection overhead was 5.9% when leveraging MPX, and 35.8% using software methods. The total CARAT overhead (slowdown of applications) was ~9% (using MPX). Note this slowdown would be matched with a hardware speedup on a system with no paging. Unfortunately, paging cannot be deactivated on x64. When data is being moved, the overhead is, of course, higher. However, even at double the maximum rate of any kind of page-level operation measured when running the benchmarks on Linux, the total CARAT overhead was only 171%. Even though the paging semantics were being emulated, the user-level CARAT prototype's overheads were quite small.

Limitations of User-level Prototype: The most obvious limitation is that the previous CARAT prototype operates entirely at user level on Linux, which suffices to make the case for the CARAT concept, but lacks in *true* implementations of protection and memory management as needed to facilitate kernel-level abstractions such as processes. Other limitations include limited or no support for heap expansion, defragmentation, shared memory, swapping, device mapping, demand paging, and copy-on-write. However, the paths to achieving these were presented, though not tested. We determined it is necessary to design and implement a full CARAT system in order to take the evaluation of the CARAT concept to the next level.

3.3 Benefits of CARAT-Based Systems

CARAT-based systems provide many immediately visible benefits when employed. The following benefits are obtained:

No Dependence on Address Translation Hardware: The largest change for a CARAT-based system is the newfound freedom from the TLB, and paging hardware in general. This lack of a virtual memory system causes the following effects:

- No more TLB misses (as there is no longer a TLB)
- ~15% energy savings on chip [30**? ? ? ? ? ? ? ? ?**].
- Area savings, assuming you remove the paging hardware, that are on par with the size of an L1 cache [**?**].
- Removal of synonyms/homonyms from cache design. This would allow larger L1 caches. We estimate that on x86/64, L1 caches could increase from 64KB to 256KB while maintaining the same energy and timing requirements.

Software Benefits: Employing a CARAT-based system leads to various software benefits as well. The first benefit is that memory can now be managed at an arbitrary granularity. For any given program, its execution can have its memory managed by allocations, pages, `mmap` regions, arenas, or any granularity the user would want to specify. The second benefit is that improvements to the CARAT software, particularly, the compiler, runtime, or kernel, would increase the performance of existing programs on existing hardware. A third benefit to hardware testing, verification, and bug fixing. Using a CARAT-based system, the hardware of address translation would ideally no longer exist; therefore, the testing and verification of this hardware could be avoided. Additionally, bugs in protection/management (e.g., Spectre) could potentially be fixed by software updates instead of microcode patching or major abstraction changes (KAISER/KPTI). Finally, it is important to note that LLVM is an open-source project with a *much* larger set of eyes on it than closed-source processor teams have. This perhaps increases the likelihood of detecting and patching bugs in these vital TCB components.

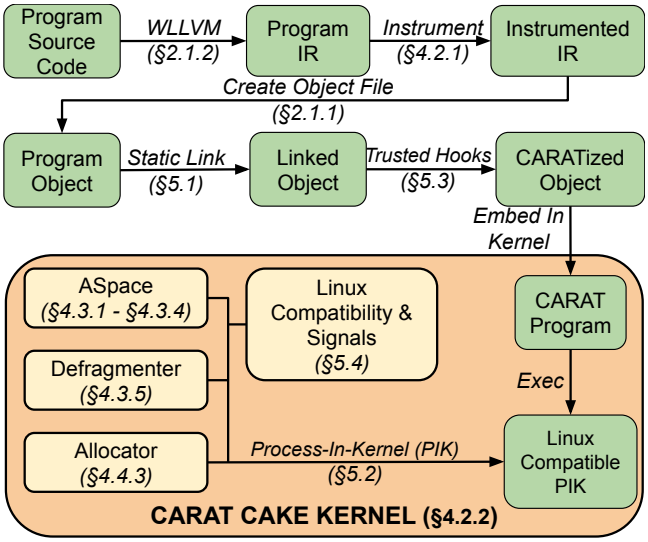

Figure 1: CARAT CAKE System Diagram.

4 DESIGN AND IMPLEMENTATION: BAKING A CARAT CAKE

The CARAT user-level prototype (§3) 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 driverâĂŹs 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 CAKEâĂŹs mechanisms instead of paging.

4.1 System and Design Choices

Figure 1 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 §3, 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

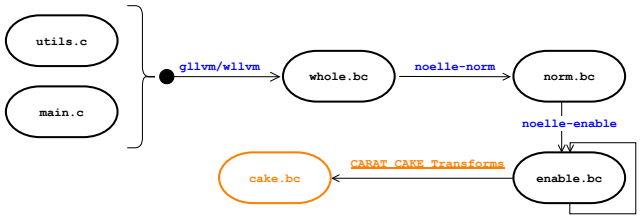

Figure 2: Example build procedure of a codebase into a CARATized program using NOELLE.

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 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 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 §5.

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 §5. 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 (§5).

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 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,

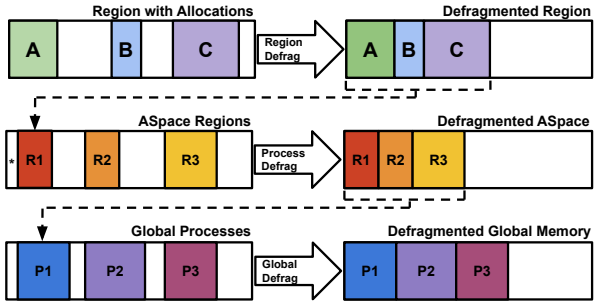

Figure 3: CARAT CAKE defragments hierarchically.

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.

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 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 [52], 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 (§5). 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. Recall from §2 that Nautilus uses buddy system allocation. 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.

5 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 [36]. 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.

5.1 Code Generation and Linking

As we described in §4, 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.[6] 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.

5.2 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.

5.3 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, 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.

5.4 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.

6 EVALUATION

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

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 compares the measured performance of our benchmarks on our target platform (both described in §2.2) using Linux as a baseline for normalization. As can be seen, CARAT CAKE and paging in Nautilus are comparable to Linux.[7] 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.

[6] 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.

[7] 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.

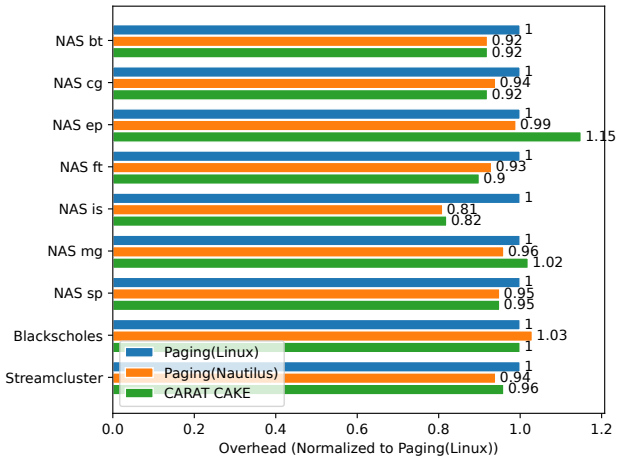

Figure 4: CARAT CAKE has comparable run time overheads.

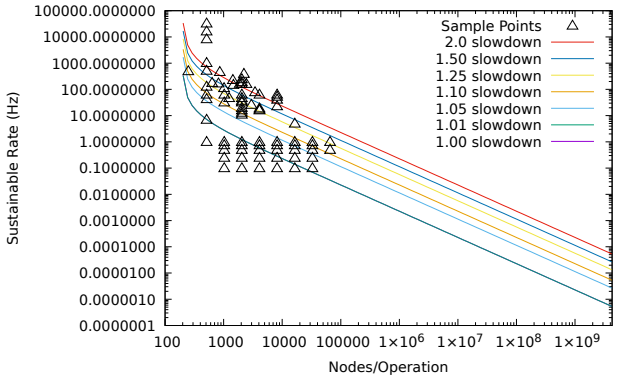

Figure 5: Possible *rate* and *nodes* (size) combinations given various constraints on application slowdown (NAS IS).

Table 2: Many programs display high pointer sparsity (𝔘).

Benchmark	Num. Allocations	Max Escapes	Pointer Sparsity (𝔘)
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

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.

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 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 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, quite high migration levels can be sustained. Also quite large migrations can be sustained at lower rates. For the range of measured page migration rates described in §3, the performance impact is trivial.

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*, 𝔘, which we define as the ratio of the amount of data moved to the number of pointers that are updated. As 𝔘 increases, we approach the `memcpy()` limit.

In the pepper results (Figure 5), we deliberately consider a *low* sparsity move (𝔘 = 8 for a 64-bit pointer linked list being moved.) Table 2 puts pepper in context by showing 𝔘 for the other code we consider, including the kernel itself. As can be seen, many benchmarks display very high 𝔘. If we were to move one of these instead of a linked list, the curves shown in Figure 5 would shift upward.

Engineering Effort is Likely Comparable to Paging: Converting a kernel that already extensively and intrinsically uses

Table 3: Breakdown of implementation sizes.

Component	Lines of code (Engineering Effort) Paging	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

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. Table 3 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 [32, 35]. 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.

7 GENERALITY AND FUTURE WORK

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. We now discuss unimplemented and untested aspects of CARAT CAKE with regard to this general purpose goal, some of which we are currently working on.

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 our prior paper [53], 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 paper have actually gone down in this paper due to efforts in compiler optimization. This supports a claim the original paper 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 [21]. 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[15] 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→HPA mapping could then simply be an unchanging identity map. Note that you could view the resulting GVA→GPA→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 often want to have a notion of a memory object not being currently present in memory. This is used traditionally 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 previous paper [53] proposed the use of non-canonical physical addresses to signify an absent object. When accessing a non-canonical address, an x64 system will generate a general protection fault (not a page fault). 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. We are currently working on just such a system for remote memory.

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 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).

For more complex obfuscation (such as encoding via indirect function calls), programmer transparency may need to be broken. The programmer would need to provide a trusted decoding function to the CARAT runtime to be used when determining the aliasing of an escape and Allocation. Failure to provide this function would result in a CARAT system being required to conservatively "pin" Allocations affiliated with the encoded escape. Pinning would maintain the correctness of the program (and of protection checks), but would limit the ability to move the affiliated Allocations.

8 RELATED WORK

There is a long chain of work on improving and extending the existing paging model, each of which attempts to address some measured deficiency [3, 6**? ? ? ? ? ? ? ? ? ? ?**]. In contrast, there is much less work on software-based memory management as an alternative.

Software-based memory management has a significant historical market impact [4, 47], and remains in use today in many forms. One form, automatic handle-based memory management with protection, dates back to the Burroughs B5000 [11, 34], which restricts programmers to specific high-level languages. A more recent incarnation is the IBM 801 [48], which combined physical addressing, a heavily restricted high-level language (PL.8), and a trusted compiler as the basis for protection ([48, pp. 240]).

The use of the compiler to implement software protections is not a new concept. EffectiveSan [18] 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) [12, 20, 51, 55] to all user programs regardless of trust. Many of the

[8] An infamous bane of garbage collectors worldwide.

innovations in SFI are orthogonal to the concepts of CARAT, and some of the optimizations in CARAT could potentially be deployed in SFI. Another alternative to achieving protection is proof-carrying-code (PCC) [42–44] 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, more specifically eBPF [1], 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.

Another related work we are aware of is the Software Isolated Process (SIP) [26] of the Singularity OS [27]. 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 [2], which we believe remains true. Zagieboylo et al [59] have more recently revisited the costs of software-based memory management compared to paging.

Another line of work to note is the CHERI capability model [13, 14, 17, 19, 38–40, 45, 56–58]. 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.

LibOSes [54], and unikernels [37] 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 [33], Theseus [10], and RedLeaf [41], 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 [16] and Twizzler [9] also approach memory management through software. Virtual Ghost 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, and MPX limits the number of regions that can be protected. Twizzler 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.

9 CONCLUSION

We have described CARAT CAKE, an alternative to paging that is based on compiler analysis and transformation and does not required hardware support to achieve protection and memory management. The machine is physically addressed, obviating the need for TLBs, pagewalkers, and similar increasingly expensive, problematic, and vulnerable hardware structures. Paging is becoming a problem. We argue it can be replaced instead of patched.

Our prototype implementation of CARAT CAKE extends a kernel to provide a Linux-compatible process abstraction that is able to achieve comparable performance to tuned paging implementations in that kernel, and in the mainstream Linux. The engineering costs of implementing the concept are manageable and arguably comparable to those of building a paging implementation.

Software-based memory management is common in managed languages. Advances in compiler technology make it feasible for unmanaged languages and as a replacement for paging. Importantly, an approach like CARAT CAKE can leverage future advances, and also provides the new opportunity for kernel-level management and protection of memory at granularities related to the code itself. For example, if we were to apply the CARAT CAKE guard pass to a monolithic kernel, it might be enable us to create internal boundaries, blurring the distinction between monolithic and microkernels.

ACKNOWLEDGEMENTS

This project was supported by the United States National Science Foundation via grants 1763743. 1908488, 2028851, 2119069.

A ARTIFACT APPENDIX

A.1 Abstract

Nautilus is an example of an Aerokernel, a very thin kernel-layer exposed (much like Unikernel) directly to a runtime system and/or application. An Aerokernel does not, by default, have a user-mode! There are several reasons for this, simplicity and performance among the most important. Furthermore, there are no heavy-weight processes—only threads, all of which share an address space. Therefore, Nautilus is also an example of a single address-space OS

(SASOS). The runtime can implement user-mode features or address space isolation if this is required for its execution model.

This *available-only* artifact of Nautilus has been modified to accommodate running with a CARAT address space abstraction (CARAT CAKE). CARAT CAKE is an extension of work found in PLDI '20 with the paper describing this work appearing in ASPLOS '22.

The concept of CARAT CAKE is to replace paging with a system that can operate using only physical addresses. Doing this enables the underlying system to have significant energy savings as well as allow new performance minded optimizations in both the micro-architecture and in software.

Nautilus w/ CARAT CAKE requires one of the following hardware:

- x86_64 machines (AMD and Intel)
- Intel Xeon Phi, either Knight's Corner or Knight's Landing
- A Hybrid Virtual Machine (HVM) in the Palacios VMM

Nautilus can also run as a virtual machine under QEMU, BOCHS, KVM, and in a simulated environment using Gem5

Nautilus w/ CARAT CAKE also requires the following software:

- clang/llvm 9.0+ (https://releases.llvm.org/download.html)
- grub version = 2.02+
- xorriso (for creating ISO images)
- qemu (for testing and debugging)
- NOELLE (https://github.com/scampanoni/noelle)
- wllvm or gclang (https://github.com/travitch/whole-program-llvm) (https://github.com/SRI-CSL/gllvm)

A.2 Artifact Check-List (Meta-Information)

- **Program: NAS and Parsec 3.0. These are included in "./src/test"**
- **Compilation: For compilation, the evaluator needs to have wllvm, clang/llvm 9+, and NOELLE. These are linked to in the abstract as publically available repositories or releases.**
- **Transformations: wllvm, CARAT CAKE compiler pass. wllvm provides llvm bitcode files of all objects compiled as a single bitcode file. The CARAT CAKE compiler pass instruments LLVM bitcode with calls/hooks into the runtime and kernel. wllvm is linked in the abstract. The compiler pass is available in "./src/llvm/carat".**
- **Run-time environment: Nautilus Kernel. Evaluator needs repository of CARAT CAKE compiled and any of the required hardware to run the CARATized kernel.**
- **Hardware: See abstract**
- **Experiments: Run "kernel_build_with_llvm_and_noelle.sh"**
- **How much disk space required (approximately)?: 1 GB**
- **How much time is needed to prepare workflow (approximately)?: Software requirements installation takes 30 minutes. Kernel compilation on the scale of 10s of seconds.**
- **How much time is needed to complete experiments (approximately)?: This is only an available artifact so the time needed is not applicable. However, in practice, the time needed is benchmark dependent and (in general) takes roughly the same time as standard compilation and execution of benchmarks.**
- **Publicly available?:**
  Yes, "10.5281/zenodo.5747303"
  or "https://github.com/SuchyB/CaratCakeArtifact"
- **Code licenses (if publicly available)?: MIT Opensource**
- **Workflow framework used?:**
  Compile Benchmark with wllvm,
  strip bitcode,
  compile with llvm and CARAT CAKE pass,
  link program with "nld-cxx-noelle-ubuntu",
  mount onto ramdisk.img,
  compile CARATized-Nautilus,
  run kernel with a VM or on actual hardware,
  from kernel shell type "exec /program.exe"
- **Archived (provide DOI)?: 10.5281/zenodo.5747303**

A.3 Description

A.3.1 How To Access. The codebase of CARAT CAKE (built on the Nautilus codebase) is publically available at the following git repository:

https://github.com/SuchyB/CaratCakeArtifact

Additionally, this has been archived via Zenodo and is available at:

10.5281/zenodo.5747303

A.3.2 Hardware Dependencies. Nautilus w/ CARAT CAKE requires one of the following hardware:

- x86_64 machines (AMD and Intel)
- Intel Xeon Phi, either KNC or KNL
- A Hybrid Virtual Machine (HVM) in the Palacios VMM

Nautilus can also run as a virtual machine under QEMU, BOCHS, KVM, and in a simulated environment using Gem5

A.3.3 Software Dependencies. Nautilus w/ CARAT CAKE also requires the following software:

- clang/llvm 9.0+ (https://releases.llvm.org/download.html)
- grub version >= 2.02
- xorriso (for creating ISO images)
- qemu or bochs (for testing and debugging)
- NOELLE (https://github.com/scampanoni/noelle)
- wllvm or gclang (https://github.com/travitch/whole-program-llvm) (https://github.com/SRI-CSL/gllvm)

A.4 Installation

To install Nautilus (and the software dependencies), there is a script located in the root directory of the CARAT CAKE repository named "kernel_build_with_llvm_and_noelle.sh" that can be run to automatically install prerequisites, set up, and compile a CARAT CAKE instrumented Nautilus kernel.

A.5 Experiment Workflow

From the root directory, LCP processes can be installed within the "./user/test" or "./src/test/" directory. This directory will be used in tandem with the contents of the "./user/framework" directory to produce LCP-CARATized processes.

A.6 Evaluation and Expected Results

To evaluate the artifact being available, perform the following steps:

- Clone the repository via git at:
  https://github.com/SuchyB/CaratCakeArtifact.git OR download the artifact from Zenodo at: 10.5281/zenodo.5747303
- Go into the root directory of the repository to be in the main code base.

- To access prerequisites for compiling, please run in the root directory: "./kernel_build_with_llvm_and_noelle.sh"
- Some of the scripts used to produce the CARATized LCP Benchmarks can be found in "./parsec_and_nas_build_scripts/

A.7 Notes

More information about the repository can be found within the README.md file in the repository. This can be viewed in a text editor OR on the github repository page.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] [n.d.]. What is eBPF? an introduction and deep dive into the EBPF technology. https://ebpf.io/what-is-ebpf

[2] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. 2006. Deconstructing Process Isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness (MSPC)*. 1âĂŞ10.

[3] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA 2017)*.

[4] Apple Corporation. 1985. *Inside Macintosh*. Addison-Wesley.

[5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinksi, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. 1994. *The NAS Parallel Benchmarks (NAS 1)*. Technical Report RNR-94-007. NASA.

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA 2013)*.

[7] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2012. Reducing Memory Reference Energy with Opportunistic Virtual Caching. *SIGARCH Comput. Archit. News* 40, 3 (June 2012), 297âĂŞ308. https://doi.org/10.1145/2366231.2337194

[8] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[9] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. 2020. Twizzler: a Data-Centric OS for Non-Volatile Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 65–80. https://www.usenix.org/conference/atc20/presentation/bittman

[10] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 1–19.

[11] Burroughs Corporation. 1961. *The Descriptor–A definition of the B5000 Information Processing System*. Technical Report BULLETIN 5000-20002-P. Burroughs Corporation, Detroit, MI. USA.

[12] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 45–58.

[13] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A Theodore Markettos, J Edward Maste, Robert Norton, Stacey Son, et al. 2017. CHERI JNI: Sinking the Java security model into the C. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 569–583.

[14] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. 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* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 117âĂŞ130. https://doi.org/10.1145/2694344.2694367

[15] KDE Open Source Community. 1998. The WebKit Open Source Project. https://webkit.org/

[16] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual Ghost: Protecting Applications from Hostile Operating Systems. *SIGARCH Comput. Archit. News* 42, 1 (Feb. 2014), 81âĂŞ96. https://doi.org/10.1145/2654822.2541986

[17] Brooks Davis, Robert NM Watson, Alexander Richardson, Peter G Neumann, Simon W Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, et al. 2019. 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*. 379–393.

[18] Gregory J. Duck and Roland H. C. Yap. 2018. 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)*.

[19] Nathaniel Wesley Filardo, Brett F Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, et al. 2020. Cornucopia: Temporal safety for cheri heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 608–625.

[20] GoogleNativeClient [n.d.]. Native Client. https://developer.chrome.com/native-client.

[21] Andrei Gorine and Alexander Krivolapov. [n.d.]. *A Kernel Mode Database System for High Performance Applications*. Technical Report.

[22] Kyle Hale. 2016. *Hybrid Runtime Systems*. Ph.D. Dissertation. Northwestern University. Available as Technical Report NWU-EECS-16-12, Department of Electrical Engineering and Computer Science, Northwestern University.

[23] Kyle Hale and Peter Dinda. 2015. A Case for Transforming Parallel Runtimes into Operating System Kernels. In *Proceedings of the 24th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2015)*.

[24] Kyle Hale and Peter Dinda. 2016. Enabling Hybrid Parallel Runtimes Through Kernel and Virtualization Support. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)*.

[25] Kyle Hale and Peter Dinda. 2018. An Evaluation of Asynchronous Software Events on Modern Hardware. In *Proceedings of the 26th IEEE International Symposium on the Modeling, Analysis, and Simulaton of Computer and Telecommunication Systems (MASCOTS 2018)*.

[26] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. 2007. Sealing OS Processes to Improve Dependability and Safety. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys)* (Lisbon, Portugal). 341–354.

[27] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Operating Systems Review* 41, 2 (April 2007), 37–49.

[28] SRI International Ian A. Mason. 2018. https://github.com/SRI-CSL/gllvm.

[29] H. Jin, M. Frumkin, and J. Yan. 1999. *The Open MP Implementation of NAS Parallel Benchmarks and Its Performance (NAS 3)*. Technical Report NAS-99-011. NASA.

[30] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. 2002. Generating physical addresses directly for saving instruction TLB energy. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 185–196.

[31] Kenneth C. Knowlton. 1965. A Fast Storage Allocator. *Commun. ACM* 8, 10 (Oct. 1965), 623âĂŞ624. https://doi.org/10.1145/365628.365655

[32] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[33] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 234âĂŞ251.

[34] Hank Levy. 1984. *Capability-Based Computer Systems*. Digital Press.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[36] Jiacheng Ma, Wenyi Wang, Aaron Nelson, Michael Cuevas, Brian Homerding, Conghao Liu, Zhen Huang, Simone Campanoni, Kyle Hale, and Peter Dinda. 2021. Paths to OpenMP in the Kernel. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC) (Supercomputing)*.

[37] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. 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)*. 461–472.

[38] A Theodore Markettos, John Baldwin, Ruslan Bukin, Peter G Neumann, Simon W Moore, and Robert NM Watson. 2020. Position Paper: Defending Direct Memory Access with CHERI Capabilities. (2020).

[39] Alfredo Mazzinghi, Ripduman Sohan, and Robert NM Watson. 2018. Pointer provenance in a capability architecture. In *10th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2018)*.

[40] Kayvan Memarian, Victor BF Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert NM Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

[41] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 21–39.

[42] George Necula. 1997. Proof-carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 1997)*.

[43] George Necula and Peter Lee. 1996. *Proof-Carrying Code.* Technical Report CMU-CS-96-165. School of Computer Science, Carnegie Mellon University.

[44] George Necula and Peter Lee. 1996. Safe Kernel Extensions Without Run-time Checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI 1996)*.

[45] Peter G Neumann. 2018. Fundamental trustworthiness principles. *New Solutions for Cybersecurity* (2018).

[46] Omni OpenMP Compiler Group, University of Versailles Saint Quentin en Yvlines. 2014. NAS Parallel Benchmarks 3.0—Unofficial OpenMP C Version. https://github.com/benchmark-subsetting/NPB3.0-omp-C.

[47] Charles Petzold. 1988. *Programming Windows.* Microsoft Press.

[48] George Radin. 1983. 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.

[49] Michael Rainey, Ryan Newton, Kyle Hale, Ryan Newton, Nikos Hardavellas, Simone Campanoni, Peter Dinda, and Umut Acar. 2021. Task Paralell Assembly Language for Uncompromising Parallelism. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. to appear.

[50] Tristan Ravitch. 2016. https://github.com/travitch/whole-program-llvm.

[51] David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. 2010. Adapting software fault isolation to contemporary CPU architectures. (2010).

[52] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (July 1985), 652âĂŞ686.

[53] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. 2020. 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)*. 329âĂŞ345.

[54] Hajime Tazaki. [n.d.]. An introduction of library operating system for Linux (LibOS).

[55] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP 1993)*.

[56] Robert NM Watson, Robert M Norton, Jonathan Woodruff, Simon W Moore, Peter G Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, et al. 2016. Fast protection-domain crossing in the CHERI capability-system architecture. *IEEE Micro* 36, 5 (2016), 38–49.

[57] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 457–468. https://doi.org/10.1109/ISCA.2014.6853201

[58] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G Neumann, Simon W Moore, Robert NM Watson, et al. 2019. Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 545–557.

[59] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. 2020. The Cost of Software-Based Memory Management Without Virtual Memory. *CoRR* abs/2009.06789 (2020). arXiv:2009.06789 https://arxiv.org/abs/2009.06789