CARAT: A Case for Virtual Memory through Compiler- and Runtime-Based Address Translation

Brian Suchy
Simone Campanoni
Nikos Hardavellas
Peter Dinda
Department of Computer Science, Northwestern University
Evanston, Illinois, United States

Abstract

Virtual memory is a critical abstraction in modern computer systems. Its common model, paging, is currently seeing considerable innovation, yet its implementations continue to be co-designs between power-hungry/latency-adding hardware (e.g., TLBs, pagewalk caches, pagewalkers, etc) and software (the OS kernel). We make a case for a new model for virtual memory, compiler- and runtime-based address translation (CARAT), which instead is a co-design between the compiler and the OS kernel. CARAT can operate without *any* hardware support, although it could also be retrofitted into a traditional paging model, and could leverage simpler hardware support. CARAT uses compile-time transformations and optimizations combined with tightly-coupled runtime/kernel interaction to generate programs that run efficiently in a physical address space, but nonetheless allow the kernel to maintain protection and dynamically manage physical memory similar to what is possible using traditional virtual memory. We argue for the feasibility of CARAT through an empirical study of application characteristics and kernel behavior, as well as through the design, implementation, and performance evaluation of a CARAT prototype. Because our prototype works at the IR level (in particular, via LLVM bitcode), it can be applied to most C and C++ programs with minimal or no restrictions.

This project is made possible by support from the United States National Science Foundation through grants CCF-1533560, CNS-1763743, CCF-1908488, and by equipment support from Intel Corporation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '20, June 15–20, 2020, London, UK
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385987

CCS Concepts: **• Software and its engineering → Operating systems**; **Compilers**; **• Computer systems organization → Architectures**.

Keywords: virtual memory, memory management

ACM Reference Format:
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 International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3385412.3385987

1 Introduction

Virtual memory has been one of the foundational technologies of computer systems ever since the implementation of paging in the Atlas Supercomputer in the 1960s [52]. Since then, paging has involved a careful hardware/software co-design spanning the hardware directly on the access path to main memory and the deepest levels of the kernel. Over the years, a steady backdrop of work has aimed at minimizing the cost (both latency and energy) of performing the address translation from virtual to physical. More recently, new innovation has been occurring in paging to better handle the combination of exploding physical memory sizes, challenging power/energy requirements, and workloads.

Historically, another approach to solving the problems that paging solves, namely protection and mapping (address binding), has been through purely software-based, language-restricted memory management. The late 1970s IBM 801 [66] was probably the earliest complete example of this, with Singularity [42] being the most recent. Partial implementations of software-based memory management for arbitrary user code, albeit lacking protection, have also been very successful. Handle-based memory management within widely-used early versions of Windows [64] and pre-2001 MacOS [5, Vol. 2, Ch. 1] was probably the most visible, and remains current in some of today's hardware MMU-free embedded systems.

We argue that it is time to revisit software-based memory management on large-scale machines running arbitrary code. The complexity and costs of the hardware support for paging are becoming limiting. 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) together already require almost as much area as L1 caches [11]. A TLB consumes a significant amount of power [24, 50, 63, 78], and is a prominent thermal hot spot [65]. Early studies find that TLB power consumption is as high as 15-17% of the chip power [47, 49]. Subsequent works corroborate these findings, with industry reporting that TLBs consume up to 13% of a core's power [30, 70], and later studies estimating that TLBs are responsible for 20-38% of the energy consumed by an L1 cache [10, 29]. Increasingly distributed TLBs are no panacea either, as there are significant overheads associated with keeping them coherent [4, 6, 11, 62, 76, 79].

Modern processors employ these hardware structures despite their high overheads because performance largely depends on fast address translation. But, what if we can get by without all this hardware? It is tantalizing to consider machines which do not need hardware such as TLBs, pagewalk caches, or pagewalkers, or at least can disable and power it down when the kernel desires. At the same time, compiler technology has been advancing [19, 46, 51, 55, 72, 74, 75, 80] and becoming more readily available through vehicles such as LLVM [53]. It may have advanced to the point at which one of the banes of software-based memory management, namely reliance on and trust of the programmer in the case of supporting arbitrary user code, may be surmountable.

In this paper, we reevaluate the design choice of the paging-based model of memory management that has a hardware/-software implementation (specifically, x64 running Linux) by considering replacing it with a purely software-based memory management model. We elaborate on our specific scheme, Compiler- And Runtime-based Address Translation (*CARAT*). CARAT's goal is to achieve the same protection and mapping capabilities for arbitrary code that paging provides, with similar overheads, but without hardware support. CARAT also exposes other potential advantages because it can operate at any granularity.

CARAT is based on specialized code injection, new optimizations, and code signing, all implemented in LLVM, and thus applicable to any language from which LLVM bitcode can be generated. Compiler middle-end transformations result in output code that operates using physical addresses, but that invokes a CARAT runtime on allocations and pointer escapes (references to allocations in memory), and guards memory references against kernel-supplied protection requirements. A key insight is that modern compiler optimizations allow these potentially very costly operations to often be eliminated or amortized, resulting in low performance overhead in normal execution of the code. The kernel upcalls into the CARAT runtime in order to change protections or move memory, as needed. These changes are then dynamically implemented by patching the memory image of the running program. A key insight is that such runtime operations are possible with minimal performance overhead.

Our contributions are as follows:

- We describe the design of CARAT in contrast to paging and discuss its potential advantages (§2).
- We present an empirical feasibility study of CARAT that shows that its prerequisites can be met (§3).
- We illustrate the design, implementation, and evaluation of a prototype CARAT system that allows experimentation on Linux (§4).

Overall, we make the case for CARAT, and indirectly, for revisiting software-based memory management.

2 Address Translation with CARAT

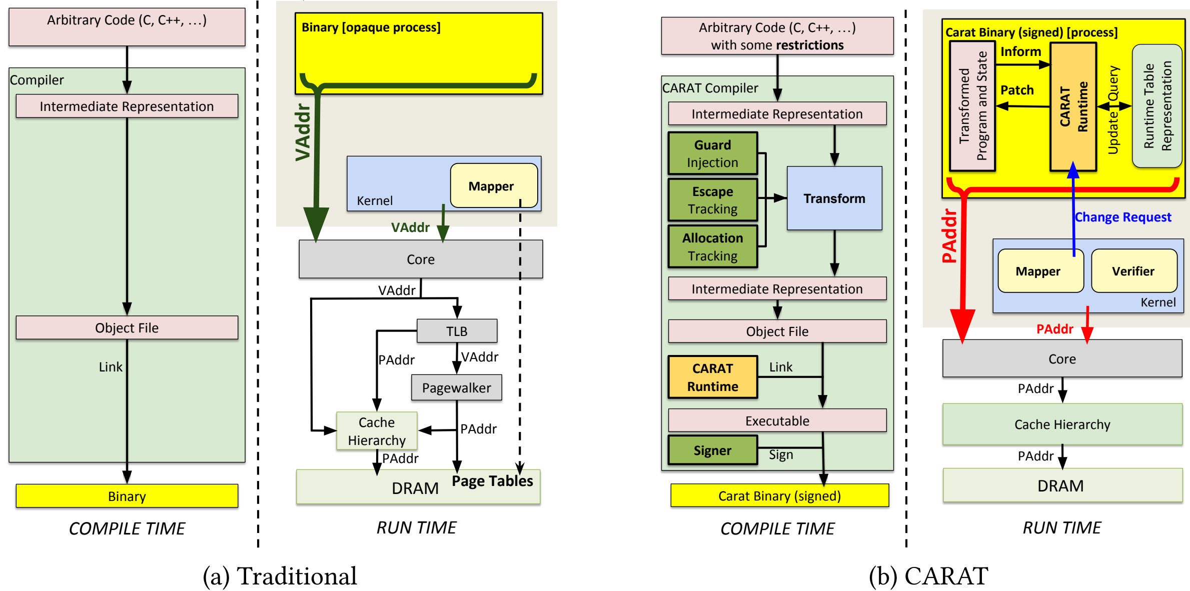
Figure 1 diagrams and compares the traditional address translation model with our proposed CARAT model. The figure also serves to put terminology we will use throughout the paper into context for both models. Our focus is on Intel/AMD x86 processors, so when we give specifics, it is with regard to these processors when operating in 64-bit mode (i.e.,"x64").

There are three important high-level differences between models to take note of. First, in the traditional model both the kernel and the process access memory using virtual addresses. In contrast, in the CARAT model, both use physical addresses. The second is that in comparison to the traditional model, the CARAT model requires substantially more work within the compiler toolchain and in the process's run-time environment. Finally, while the traditional model leans heavily on hardware support, and cannot be effective, or work, without it, the CARAT model requires no hardware support. Indeed, because physical addresses are used throughout in CARAT, there is even an opportunity to simplify and/or speed up the remainder of the memory system.

2.1 Traditional Model

In the traditional address translation model of paging (Figure 1(a)), the compilation and linking process is simplified because it can target an abstract virtual address space that is independent of the actual machine's physical address space or how the kernel is currently using it. From the kernel's perspective, the process is mostly opaque, and the kernel's responsibility is to create the illusion of the abstract virtual address space. It does this jointly with the hardware by interposing on each and every memory reference.

Every memory reference, including an instruction fetch, has its virtual address (*VAddr*) translated to a physical address (*PAddr*) that is ultimately the address the memory system uses. On x64 and other processors, the VAddr is also used to immediately begin the cache line lookup in the highest level(s) of the cache hierarchy, but before any data is returned, the PAddr resulting from translation is used to determine if there was a cache hit. Typically, the L1 cache is virtually indexed, but physically tagged.



(a) Traditional (b) CARAT

Figure 1. Comparison of the traditional address translation (paging) model with the CARAT model.

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

VPN→PPN is a mapping from occupied virtual pages in the current virtual address space to the occupied physical pages in the physical address space. This mapping is determined by the kernel and it may change over time. Current systems represent mappings as radix trees.

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

2.2 CARAT Model

In the CARAT model (Figure 1(b)), the hardware can be considerably simplified, but the compilation and run-time environment are considerably more complicated. Because only physical addresses are used, the TLB and pagewalker can be eliminated. In a system that provides both the traditional and CARAT models, the kernel could switch between them with simple hardware support. For example, on x64, physical addressing could be reintroduced, allowing a kernel write to set CR0.PG to zero to disable paging, a capability already present when an x64 processor is run in 32-bit mode.

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

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

Obviously, if each relevant instruction truly were guarded, the overhead of CARAT would be abysmal. As we describe in more detail in Sections 3–4, the CARAT model heavily relies on compiler optimization technology, including new CARAT-specific optimizations, to eliminate, combine, or amortize guards in many situations. An important result is that this is possible in a wide range of programs.

It may seem strange to guard call instructions, but this is necessary since the call's push of the return address onto the stack could overrun a valid region. Additionally, the prologue and epilogue code the compiler produces for the callee may also perform stack accesses. A call guard verifies that all

such "hidden" stack accesses will be within a legal region. A failed guard involving the stack causes the kernel to be invoked. This provides a mechanism by which the kernel can implement seamless stack expansion, if desired. A compiler could also ensure that the control flow of calls and returns happens on a separate control flow stack, thus making a stack-oriented attack, even ROP, impossible.

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

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

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

The final compilation step involves signing the resulting binary with the credentials of the compiler toolchain, so that it is easy to validate that a specific compiler made the binary. A kernel can then determine whether to trust the binary based on the provenance of the compiler.

Run-time: CARAT processes and the kernel run within a single physical address space using physical addresses. This is a marked difference from the traditional model.

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

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

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

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

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

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

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

Architectural benefits: it is important to note that obviating the need for paging decouples the design of the L1D

cache from constraints imposed by the virtual memory. For example, modern core designs bend over backwards to hide the latency to access the level-1 DTLB through the use of a virtually indexed and physically tagged model, which is itself severely complicated by the fact that two virtual pages can be synonyms for the same physical page. x64 processors today handle this problem through the use of L1D cache associativities that are much higher than what is necessary for low miss rates (e.g., 64KB 16-way) [10]. Doubling the L1D capacity to 128KB or 256KB would require a 32-way or 64-way set associativity respectively, which may well be too slow and power-intensive. In contrast, CARAT would cut loose cache design from the anchor of synonyms and paging. L1D caches under CARAT could have far higher capacities and thus lower miss rates, while matching or surpassing current latency and energy requirements.

2.3 Security and Trust

The goal of CARAT is to provide the same protection capabilities as the traditional paging model: the application may only access physical addresses chosen by the kernel, and then only via the kernel's chosen access types. The prototype described in this paper protects regions (contiguous runs of physical addresses). The more regions in the application's address space, the higher the cost of this protection at run-time, which makes compile-time optimization (to eliminate access checks) and run-time adaptation (to minimize the number of regions) essential for performance. Also note that while the prototype operates at the page granularity, this is not strictly required by CARAT (Section 6), which may be an advantage for protection.

To achieve this protection, CARAT shifts the trusted computing base (TCB) from the traditional model's composite of the hardware and the kernel to the composite of the compiler and the kernel. Note that the hardware component here is massive given the need for low latency, plus x64 hardware paging is inherently complex (it is Turing-complete [7]), which suggests it itself is not immune from attack and, indeed, has been attacked [26]. Reducing the hardware footprint with the TCB is a potential advantage of CARAT.

Arguably, the traditional model's TCB already includes the compiler, as the compiler is used to build the kernel. Furthermore, other work, such as Keystone-Enclave [54] and the nVIDIA toolchain also shift responsibility within the TCB to include the compiler. There are two subtle points, however. CARAT relies on extensive compiler optimizations that are based on existing and new analysis techniques [19, 46, 51, 55, 72, 74, 75, 80]. These elements expand the size and complexity of the compiler beyond that needed to build a kernel in the traditional model. In the case of the prototype, by 63,168 lines of C++. Furthermore, while in the traditional model, analysis or optimization bugs when compiling the application can at worst damage the application, in the CARAT model, such bugs could lead to protection violations.

2.4 Undefined Behavior

Undefined behavior in C/C++ include integer overflow, unsequenced modifications, data races, loops that do not terminate, strict aliasing violations, memory safety violations, and alignment violations. The only undefined behaviors that concern CARAT are the last three. The additional code (the guards) added by CARAT to the program's original code guarantees protection even in the presence of these undefined behaviors. A guard validates each memory access against the kernel's permitted regions of physical addresses, and aborts on an out-of-bounds access. Hence, protection is guaranteed even in the presence of undefined behavior. A CARAT optimization only changes a guard when it can prove that this invariant is not changed. Finally, the last memory accesses to consider are those generated by the compiler's back-end. Such accesses are controlled by CARAT and trusted by design as CARAT includes the compiler in the TCB.

3 Feasibility

Can the CARAT model possibly be as—or more—effective than the traditional model? We now consider the requirements that must be met for CARAT to be feasible and effective, and provide empirical evidence that they are met.

Benchmarks and Testbeds: We use a set of benchmarks important to a variety of communities. From Mantevo [8, 39] we use the HPCCG benchmark. From NAS [45], we draw CG, EP, LU, and FT. From PARSEC [15] we use blackscholes, bodytrack, canneal, fluidanimate, freqmine, streamcluster, swaptions, and x264. Finally, from SPEC2017, we include deepsjeng_s, lbm_s, mcf_s, nab_s, namd_r, x264_s, xalancbmk_s, and xz_s. These were chosen to represent different kinds of interactions with the kernel virtual memory system.

All numbers we report, with with one exception, are collected on a dual-socket system with 2.3 GHz Intel Xeon E5-2695 v3 processors and 256 GB of DRAM spread across 2 NUMA zones. It runs RHEL Server release 7.7 with a 3.10.0 kernel. The only exception pertains to the results of Table 2 which are collected on a Dell T620 with two 1.8 GHz Intel Xeon E5-2603 v2 processors and 32 GB of DRAM spread across 2 NUMA zones. It runs Ubuntu 18 with a 4.18.0 kernel.

The Traditional Model is Expensive: Because of the interaction of cache hierarchies and address shown in Figure 1(a), every cycle lost in TLB misses is a cycle that may be lost in application performance. Growing TLB capacity is extremely challenging, resulting in level-1 TLBs that have stayed quite small (e.g., 64 DTLB entries in modern Intel processors [44]) and level-2 TLBs growing only glacially (e.g., from 1024 STLB entries on 2013's Haswell to 1536 on today's generation). The growth in reach has also been modest, culminating in perhaps 3 GB in today's TLBs. The combined pressure to keep delay, power and area at bay, ultimately hamper the growth of traditional address translation.

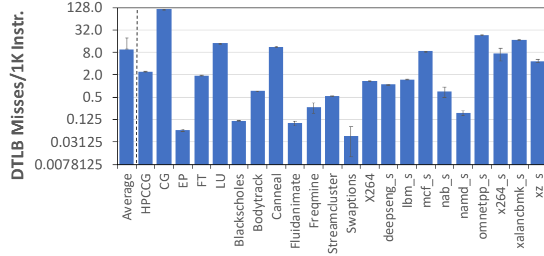


Figure 2. Level-1 DTLB misses per 1000 instructions.

While TLB reach remains uncomfortably low, data footprints continue to grow exponentially [1, 17, 37, 68], and workloads often suffer TLB misses and long-latency page walks to satisfy data and instruction fetch requests [12, 13, 24, 32, 38, 50, 63, 69, 78]. In Figure 2, we report our own measurements of the level-1 DTLB miss rates for our benchmarks, collected using Intel's PMU tools [44]. Some workloads incur more than 116 DTLB misses per thousand instructions. Despite the STLB, these misses still translate into one page walk per thousand instructions on average, and as many as 6 page walks for some workloads. Given that each page walk consumes 47 cycles on average for our workload suite, and sometimes as much as 108 cycles, it is easy to see why the cost of address translation continues to be a vexing problem. Prior studies largely corroborate the high cost of TLB misses, and the need to mitigate it has fueled much contemporary research [3, 9, 12–14, 24, 32, 38, 50, 63, 69, 78].

Protection and Mapping can be Separated: Traditional address translation by paging as in Figure 1(a) solves essentially two problems. The first is mapping, which allows the kernel to dynamically move memory content located in physical memory. Such dynamic movement of content also includes the ability to allocate space for new content and to migrate content to/from physical memory (swapping, demand paging, etc). The second problem the traditional scheme solves is protection. It allows a (possibly overlapping) partition of the *physical address space* into parts that the process can access, and parts it cannot. This partition is also informed by the type of access needed (on x64, {none, read, read+write} × {none, exec} × {user, kernel}). While traditional paging achieves protection by considering the page table entry (PTE) associated with the virtual address, it is the binding of the physical address and the access type encoded in the PTE itself that is the heart of protection.

Protection and mapping enable a range of kernel mechanisms and services, such as shared libraries, copy-on-write, shared memory, automatic stack expansion, debugging, distributed shared memory, garbage collector accelerators, and zero-copy I/O. It is feasible to implement these on a CARAT system. The essential concept, of causing the kernel to be invoked on a user-level memory reference to a page that the kernel previously set up to be inaccessible, can also be built in CARAT. By moving the page address out of any guard region, or by patching the escapes of all allocations on the page to point to a non-canonical address, the kernel can set the trap needed to regain control when the page is touched.

Table 1. Effectiveness of Compiler Optimizations.

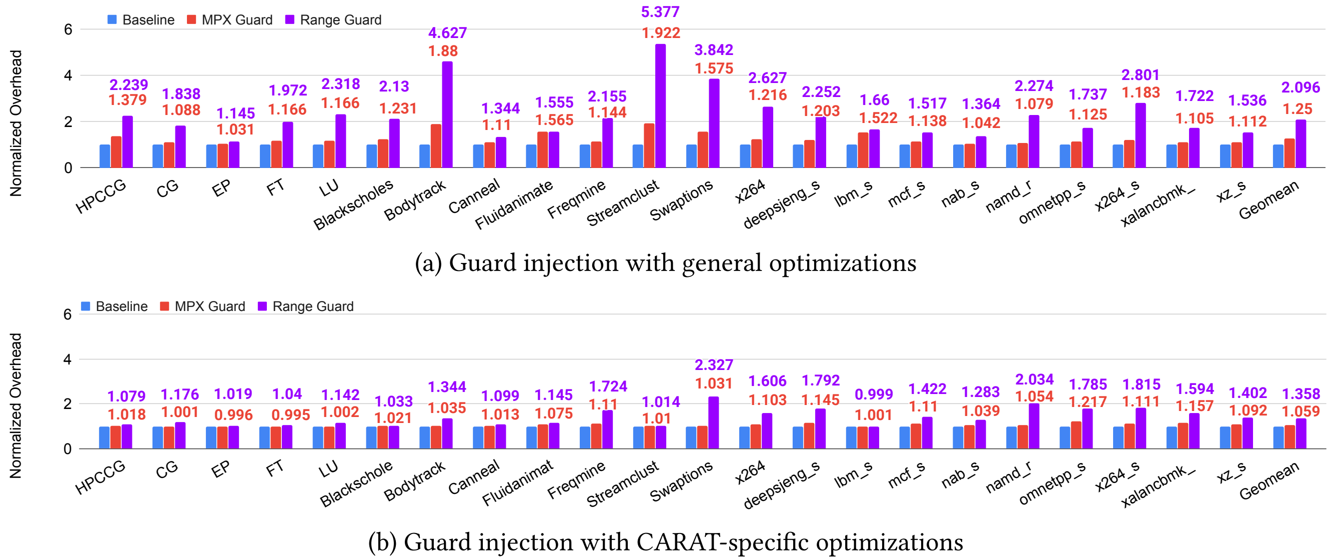
Benchmark	Opt. Total Guards	Untouched Guards	Opt. 1	Opt. 2	Opt. 3
HPCCG	0.405	0.297	0.073	0.035	0.606
CG	0.723	0.218	0.130	0.375	0.278
EP	0.439	0.245	0.108	0.085	0.561
FT	0.592	0.175	0.096	0.321	0.408
LU	0.757	0.133	0.035	0.589	0.243
Blackscholes	0.320	0.249	0.071	0	0.680
Bodytrack	0.213	0.136	0.048	0.030	0.787
Canneal	0.587	0.390	0.197	0	0.413
Fluidanimate	0.402	0.210	0.192	0	0.604
Freqmine	0.798	0.513	0.102	0.183	0.202
Streamcluster	0.400	0.316	0.083	0	0.600
Swaptions	0.343	0.279	0.064	0	0.657
x264	0.685	0.515	0.098	0.071	0.315
deepsjeng_s	0.583	0.431	0.091	0.061	0.417
lbm_s	0.762	0.066	0.098	0.598	0.238
mcf_s	0.804	0.406	0.098	0.300	0.196
nab_s	0.534	0.315	0.145	0.075	0.466
namd_r	0.764	0.555	0.055	0.154	0.236
omnetpp_s	0.665	0.408	0.235	0.021	0.335
x264_s	0.660	0.495	0.089	0.076	0.341
xalancbmk_s	0.781	0.488	0.227	0.065	0.219
xz_s	0.687	0.442	0.148	0.097	0.313
Arith. Mean	**0.587**	**0.331**	**0.113**	**0.143**	**0.414**

While mapping and protection are active when executing in the kernel, they are configured to be much more lax. In a modern kernel, the entire physical address space is mapped into the kernel virtual address space, which already allows any kernel code to access any physical address. Hence, the CARAT model of having the kernel run directly using physical addresses presents little difference.

The traditional scheme, as applied in x64, also serves as a place to encode the semantics (cache-ability, write-combining, etc) of the underlying physical addresses, in the form of the page attribute table (PAT) extension. Note that PAT coexists with another model, the memory type range registers (MTRRs) which operate independently of paging. The MTRR mechanism would be used on a CARAT system.

Tracking for shared libraries would require that LLVM bitcast-level typing information be preserved in the library file and made available for the runtime to track. More generally, enough information must be exist to track pointer escapes across process boundaries.

Protection can be Maintained through Other Mechanisms: In CARAT, only physical addresses are used. Hence, we must be able to prevent invalid accesses to physical memory. We do so, conceptually, by guarding each memory reference in the program using compiler-injected code and compiler-maintained invariants. While the simplest form of this obviously has an impossibly high overhead, it is important to understand that compiler technology can greatly



(a) Guard injection with general optimizations

(b) Guard injection with CARAT-specific optimizations

Figure 3. Overhead of protection is very small with modern compiler optimizations.

improve the overhead in most cases, and that hardware assistance, both current and future, could improve it further.

We consider the physical address space to be partitioned into regions, with a set of regions being available to the process. Note that because of the ability to dynamically move memory content, these regions are not fixed. A guard needs to verify that an address is within some valid region.

In the simplest case, the process has a single region. This can be achieved using a linkage model similar to a "dark capsule" [25]. In this model, the default stack is placed below the text, resulting in a contiguous range of addresses. Additional stacks are allocated from the process heap, and shared libraries are loaded into heap-allocated space.

The single-region model is not intrinsic to CARAT, but it does represent an optimal case for protection via guards. Given that most processes tend to have a small number of contiguous regions in their memory map (Section 5), a multi-region guard searches a smaller space compared to the space of PTEs in the traditional model. Much more important than the complexity of the guard is the ability to avoid invoking it in the common case by using compile-time analysis.

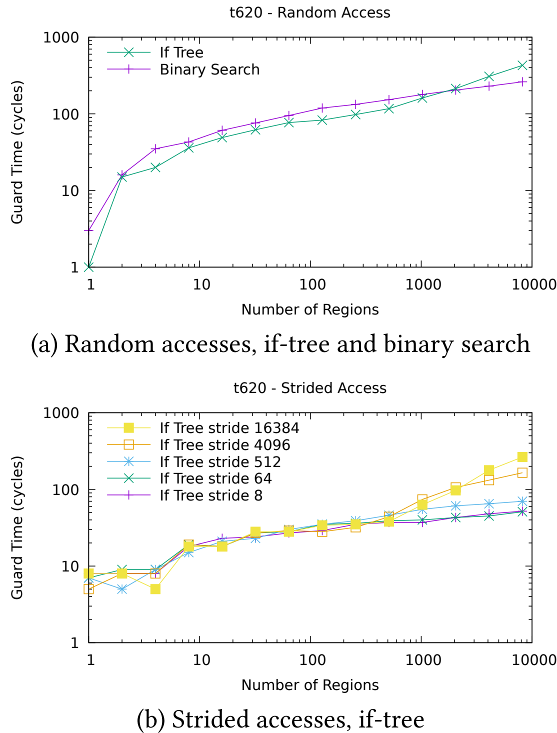
Note that run-time software-only guards cannot protect against instruction fetches from disallowed addresses. A description of how this is handled as given in Section 2.2. Issues regarding stack-modifying control flow are also described there, motivating the need for guarding call instructions.

Three forms of guards are produced. A load or store guard checks the physical address and extent of a data reference against the current valid region(s). A call guard precedes a call and verifies that the stack pointer, when offset by the maximum stack footprint of the compiler-produced code of the target function, is within the bounds of a valid region.

Figure 3 shows overheads of single-region guards introduced by CARAT in the absence of other CARAT-introduced code. We consider guarding stores, loads, and calls with two techniques of implemention. The straightforward technique simply does a bounds check using compare and branch logic (like an upper/lower bound if statement), which unfortunately introduces register pressure and memory references. We also consider implementing the guard test assuming Intel's MPX [43, Vol. 1, Ch. 17]. The MPX variant of the technique (MPX Guard) does the bounds check assuming a bounds check instruction, which compares an address to a bounds register in a single cycle without register pressure or introducing new memory references. In all cases, an out-of-bounds or illegitimate address raises a general protection fault via a non-canonical address reference or MPX violation.

Figure 4 shows the performance of pure software guards for the worst case (random access pattern) and the best case (strided reference pattern) as a function of the number of regions. Two mechanisms are considered: basic binary search and an "if-tree", which lays out the search statically. Hardware assistance, such as the protection components of rangeTLB [32]) or multiplexor trees [40] could be leveraged to enhance performance. However, the most important factor is to avoid the need for a guard in the first place.

Figure 3(a) shows the overheads for guard injection when only readily-available compiler optimizations are applied to the codebase. In Figure 3(b), we also apply CARAT-specific optimizations (Section 4.1). Using the CARAT-specific optimizations combined with MPX hardware support, we can readily see that the overhead of protection is small (~5.9%). Even without MPX hardware support, the CARAT-specific optimizations lead to low overheads in most cases simply by eliminating or amortizing guards. Table 1 shows how each CARAT-specific optimization impacts the overhead. Column "Opt. Guards" is the fraction of original guards that statically remain after the optimizations. The column "Untouched



(a) Random accesses, if-tree and binary search

(b) Strided accesses, if-tree

Figure 4. Performance of multi-region software guards.

Guards" is the fraction of original guards that could not be optimized at all. The columns "Opt. 1", "Opt. 2", and "Opt. 3" show the fraction of original guards that were optimized through hoisting, scalar evolution, and redundancy elimination, respectively, which are described in Section 4.1.1.

The upshot here is that the combination of compiler techniques and available hardware features (MPX) make it possible to *fully* guard all of our benchmarks with little overhead!

Typical Mappings Change Slowly with Time: In the traditional address translation model of paging (Figure 1(a)), a change of protection or mapping is a constant-time operation, the modification of a page table entry and an invalidation. In contrast, in the CARAT model, this is only true for protection—a region change is a modification of a region entry. A mapping change could require the patching of a non-constant number of pointers (addressed later) and a high rate of mapping changes could exacerbate that problem.

To study the rate of mapping changes, we run our benchmarks on the quiescent, dedicated machine with instrumentation to capture paging-related behavior. Of course, the rate of mapping changes on a Linux platform depends on numerous factors, including the overall workload set on the machine, the memory pressure, the use of protection-based allocation mechanisms such as first-touch NUMA allocation, copy-on-write mechanisms and demand-paging, and the extent to which the kernel is reoptimizing the selection of physical pages for example for NUMA optimization, kernel same-page merging, or transparent huge pages.

Our instrumentation includes three components. In the first, *static footprint capture*, we examine the composite of the ELF executable of the benchmark and all shared libraries that it immediately depends on to find the sizes of all LOAD sections. A LOAD section indicates a portion of the address space that the kernel loader is obligated to eventually allocate and either fill from file contents (e.g., text and data sections), or to fill with zeros (e.g., bss sections). We refer to this space as the static footprint. The static footprint represents allocations and mappings that could be done by the kernel immediately at program launch.

The second component, *initial mapping capture*, happens when the benchmark is run. Using interposition on the static constructor mechanism with a preload library, we pause the program shortly after `exec()` completes. At this point, we capture an initial page table snapshot using a previously published tool [25]. The initial snapshot represents the page allocations and mappings that have actually been done by the Linux kernel at program launch.

Our final component, *dynamic paging capture*, generates a trace of page table changes from initial mapping capture through the end of execution. This is done with a kernel module that uses the Linux MMU notifier [22, 23] interface. The intended purpose of this interface is to allow external MMUs (e.g., IOMMUs, SRIOV MMUs, and hypervisor MMUs (VTLBs) for virtual machines), to be informed of changes that Linux makes to the CPU MMUs affecting a given address space. We use it to learn of changes Linux makes to a process.

It is important to understand that the MMU notifier interface essentially provides us with two kinds of events that are of interest. One event is a PTE change in which an existing, valid PTE now points to a different physical page (a page move). The second, more common, event is a generalization of a TLB invalidation that provides a range of addresses.

Note that PTE changes (like changing invalid to valid) due to page allocations are not immediately visible in this model. Since the relevant PTE was marked invalid before the allocation, no PTE change event is sent—the PTE should already be invalid in any secondary MMU. No invalidation event is sent either, because Linux need not invalidate the CPUs' own TLBs either—an invalid PTE cannot possibly be loaded in any TLB, and hence need not be invalidated. As a consequence of these limitations, we also track the physical size of the process address space. This lets us derive the PTE changes that correspond to allocations.

Table 2 shows our results. Across the board, physical page movements are incredibly rare events. This may seem surprising given the commonplace use of demand paging and copy-on-write, but our methodology counts such activity toward the allocation rate—a faulting access, for example a write to a read-only copy of a copy-on-write page, triggers an allocation. Once a physical page is allocated and mapped into a virtual page, the probability is extremely small that the kernel will decide to shift its contents to a different page,

Table 2. Page (4KB) Allocation and Movement Rates

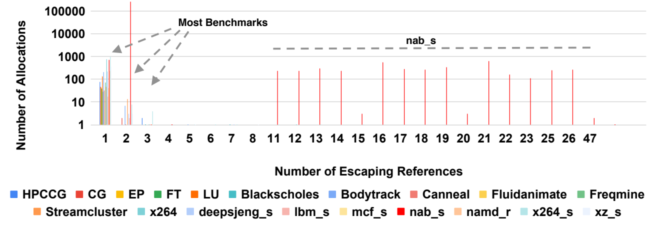
Benchmark	Static Footprint	Initial Pages	Page Allocs	Page Moves	Exec Time	Alloc Rate	Move Rate
HPCCG	1,185	836	408,106	3	34.9s	12,027/s	< 1/s
CG	283,307	837	279,148	0	401.8s	697/s	0/s
EP	1,628	837	347	0	706.6s	2/s	0/s
FT	1,713,821	845	1,712,775	0	717.4s	2,388/s	0/s
LU	174,649	835	173,685	0	2250.5s	78/s	0/s
blackscholes	795	463	156,335	1	242.9s	645/s	< 1/s
bodytrack	1,538	831	33,145	3	415.2s	82/s	< 1/s
canneal	1,222	861	240,210	3	177.7s	1,356/s	< 1/s
fluidanimate	1,208	838	129,470	3	540.6s	241/s	< 1/s
freqmine	1,743	832	458,431	23	1943.6s	237/s	< 1/s
streamcluster	1,208	834	29,570	52	547.9s	55/s	< 1/s
swaptions	1,406	833	804	0	852.9s	< 1/s	0/s
x264	1,695	862	21,703	0	831.8s	28/s	0/s
deepsjeng	4,410	817	1,760,911	0	1538.7s	1,145/s	0/s
lbm	1,406	800	1,071,100	0	4020.8s	267/s	0/s
mcf	1,422	830	1,646,263	1	2581.9s	638/s	< 1/s
nab	1,612	841	147,376	0	9785.3s	15/s	0/s
namd	1,925	855	45,063	0	998.3s	46/s	0/s
x264 pass1	1,744	843	43,342	0	128.2s	345/s	0/s
x264 pass2	1,744	826	11,851	0	412.6	31/s	0/s
x264 seek500	1,744	847	43,140	0	420.8	105/s	0/s
xalancbmk	3,610	875	122,171	0	1287.7s	96/s	0/s
xz cld	1,514	810	1,084,486	0	4066.9s	267/s	0/s
xz cpu2006	1,514	848	3,933,456	0	4462.2s	882/s	0/s
Geo. mean						**159/s**	**< 1/s**
Harm. mean						**14/s**	**< 1/s**

and update the corresponding virtual page. A second reason why page movements are very rare is that our programs fit into physical memory, hence we rarely see page swaps.

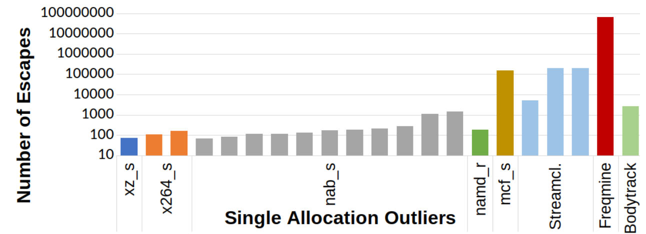
Allocation and move rates are of interest for the feasibility of CARAT. The move rate is the lower bound on the patching event rate. The allocation rate is related to the demand on the injected allocation tracking code. In a CARAT system supporting demand paging, copy-on-write, etc, allocations also trigger patching events, hence allocation rate also serves as an *extreme* upper bound on the patching event rate.

Some applications, exemplified by NAS LU, CG, and FT, have the potential to be preallocated. As Table 2 shows, their static footprint and allocations are nearly identical. FT, for example, operates on large arrays that are implemented as global variables and which become ELF LOAD sections corresponding to bss. CARAT could optimize the handling of a program like FT by allocating, and patching, the entire static footprint at `exec()` This mechanism would amortize the patching costs of LU, CG, FT, and similar programs.

There are Few Escapes per Allocation: Traditional address translation does not have to track pointer escapes at all, but this is an essential element of CARAT. CARAT must not only track each escape as it occurs, but, when handling page movement, it must also patch each escape of each pointer to an allocation in the migrating page. The more escapes per allocation that exist, the greater the cost of tracking and patching, both in terms of space and time.



(a) Histogram of allocations with less than 50 Escapes



(b) Histogram of allocations with more than 50 Escapes

Figure 5. 90% of allocations across all benchmarks contain 10 or fewer escapes. There are very few allocations with more than 50 escapes across all benchmarks.

Figure 5 shows histograms of the total number of escapes per allocation in our benchmarks over the entire run time of the program. To produce these results, we used an instrumented version of the CARAT prototype described in Section 4. For a majority of the benchmarks, the number of escapes per allocation is tiny (usually 0, 1, or 2 escapes in the worst case). The outlier in Figure 5(a) is the application nab_s from the SPEC2017 benchmarks suite.

Figure 5(b) shows the histogram of all allocations that contain more than 50 escapes. there are only 22 total allocations across all of the benchmarks that contain more than 50 escapes. This accounts for *significantly* less than 1% of all allocations across all benchmarks.

Tracking Allocations/Escapes uses Little Memory: CARAT, unlike the traditional model, must dedicate space to maintain query-able data structures that contain the current allocations and escapes. It could be the case that the space costs are insurmountable. To see if this is a real problem we looked at the physical memory footprint of each benchmark to see how much overhead was needed to keep track of all the program's allocations and escapes. As before, an instrumented version of the CARAT prototype was used. The data structures employed are described in Section 4.

Figure 6 shows compares the memory footprint of the baseline programs (no CARAT) and the CARAT-instrumented programs. The memory overhead to track allocations and escapes is small. It has a geometric mean of 61.9%, but, typically, the overhead is negligible. The mean is inflated due to Swaptions (it would be 19% if we limited all overheads to 5x). The absolute overheads of the worst case benchmarks are: swaptions (18 GB), bodytrack (100 MB), and nab_s (400 MB).

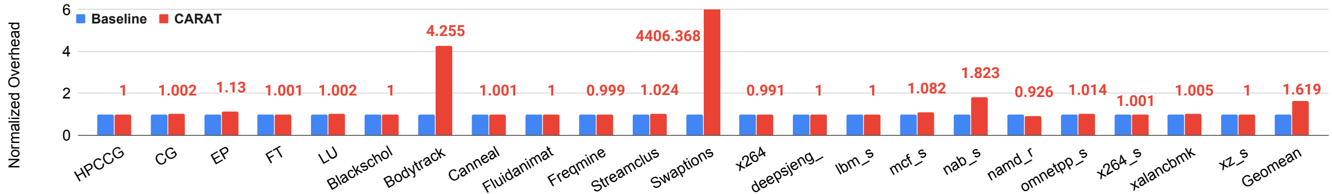


Figure 6. Memory overhead of tracking.

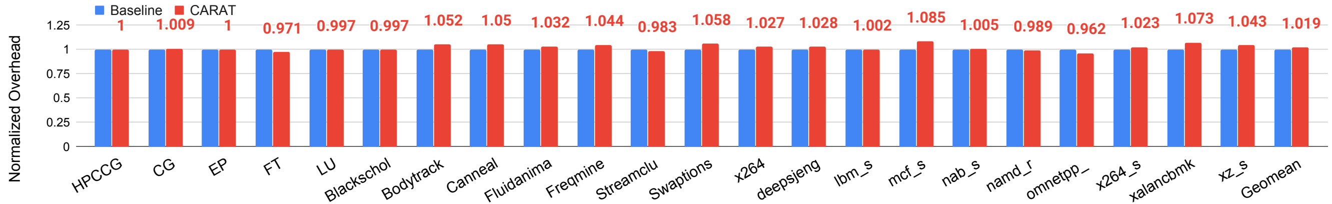


Figure 7. Time overhead of tracking allocations & escapes.

Tracking Allocations/Escapes has Low Time Overhead: It is also possible that the time costs of tracking allocations and escapes, something the traditional model does not need to do at all, could make the CARAT model infeasible. To test this point, we ran our benchmarks using an instrumented version of the CARAT prototype (Section 4).

Figure 7 shows our results. It is clearly the case that the tracking overhead is negligible and therefore a nonissue. In fact, the geometric mean of the overhead is only 1.9%! Furthermore, even streamcluster, which raised minor concerns when it came to escapes per allocation and memory overheads, is not the odd one out here. While streamcluster has many escapes from a small number of allocations, these occur early, and then no further escapes happen.

4 Prototype

We have developed a prototype of CARAT within the context of the LLVM compiler toolchain, the Linux kernel, and x64 processors. The intent of the prototype is to approximate a future hardware environment in which it is possible to deactivate paging, and a compiler/kernel combination that takes advantage of it. The data we used in our feasibility argument of Section 3 was collected using this prototype.

Our prototype consists of middle-end LLVM passes, the addition of a runtime that is linked to the Linux executable the compiler produces, and a kernel module that integrates with the runtime to execute change requests. At a high level, our compiler injects bitcode into the program that implements guards for protection, tracks allocations and pointer escapes, and assists in the patching necessary for page or object movement. The runtime maintains data structures that can be queried to determine matching allocations and escapes. When triggered by the kernel module, the runtime performs the needed change in protection or mapping.

Figure 8 shows the run-time structure of the prototype and illustrates how a page movement (mapping change) is executed. A protection change is a simpler variant of this process that does not involve patching of program pointers.

4.1 Compiler

The CARAT compiler is responsible for three tasks: injecting and optimizing guards to enforce protection, injecting and optimizing memory allocation and pointer escape tracking code in anticipation of mapping changes, and signing the final binary to establish trust between the compiler and kernel. Guard and tracking code injection relies on modern code analyses and transformations (described next). Code signing uses the same scheme implemented as Microsoft .NET, which signs the generated CIL bytecode [28].

The CARAT compiler implements protection and tracking services by targeting the LLVM IR. Only the memory and call instructions in this representation need to be considered.

4.1.1 Protection. CARAT guarantees that all memory instructions access only memory previously allocated by the program (indirectly via the kernel). To establish this invariant, CARAT conceptually injects guard code before each memory instruction to validate its address range at run time. The guard code compares the prospective address range with a set of valid regions established by the kernel and written by it into the program's address space via interaction with the runtime, as described later. An out-of-bounds check aborts to the kernel.

To reduce the overhead of guards, CARAT implements several compiler optimizations and their related code analyses to avoid redundant checking of addresses. CARAT code analyses include a Program Dependence (PD) analysis [31], a CARAT-specific data-flow analysis, and a value range analysis for pointer definitions [16]. The PD analysis relies on 15 memory alias analyses [19, 20, 34, 34, 53, 58, 71, 73] that we have combined using the alias chaining feature of LLVM (which implements a best-of-N approach). To further improve the impact of alias analyses on the PD analysis of

CARAT, we modified the LLVM internals to allow alias queries between functions (something not allowed in mainline LLVM). CARAT-specific code transformations include hoisting guards outside loops, merging guards that check consecutive memory regions, and eliminating redundant guards. These optimizations roughly add an overhead of about 22% to the compilation time and, for our benchmarks, added only minutes to compilation time in the worst case. We now further describe these code analyses and transformations and their relationships.

Optimization 1: Hoisting Guards: The first set of optimizations is related to hoisting guards out of loops when the address being checked is loop-invariant. The loop-invariant analysis determines when a store or load instruction's address is loop-invariant and will hoist the guard into the preamble of the loop (i.e., the pre-header of that loop). We have enhanced the default loop invariant analysis of LLVM to rely on the PD analysis of CARAT; this significantly improved the detection of loop invariants. Finally, this optimization recursively applies itself to hoist guards to the outermost loop possible.

One thing to note is that call instructions are also checked to guarantee their stack allocation frames are within the valid memory ranges, as described in Section 3. To reduce the overhead of call guards, the CARAT compiler hoists a call guard out of a loop if it can prove that no stack allocation occurs within that loop, much like a loop-invariant address.

Optimization 2: Merging Memory Guards: The second optimization is merging guards of memory instructions that access consecutive memory locations. In more detail, if a loop address is statically determined to check a range of addresses during the lifetime of the loop, we can hoist the guard out of the loop and perform a single check on the lowest and highest address that the variable address will alias during the loop execution. This optimization relies on a standard value range analysis [16], which we have implemented relying on both the standard scalar evolution analysis available in LLVM and the PD analysis of CARAT.

Optimization 3: Removing Redundant Guards: CARAT implements a new ad hoc analysis, *Address Checking for Data Custody analysis (AC/DC)*, to avoid re-checking previously checked addresses. AC/DC is performed using the available expression data-flow equations [2] where expressions are pointer definitions. Pointer definitions (defs) are instructions that define an IR variable that holds a pointer. Such defs are identified using memory alias analysis when the related variable escapes the function in which it was declared (i.e., if the pointer gets stored in memory). Given that $GEN[i]$ is the specific pointer def performed by the instruction i, and $KILL[i]$ is the set of defs that could modify the same IR variable, then $IN[i] = \bigcap_{p \in predecessors} OUT[p]$ and

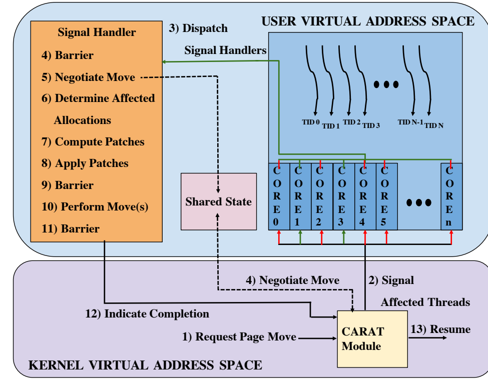


Figure 8. CARAT prototype executing a page movement.

$OUT[i] = (IN[i] - KILL[i]) \cup GEN[i]$. These equations guarantee that if a pointer def is available in the IN set of an instruction i, then the def is performed before i and is still valid independent of which execution path was taken to reach i; they do not need to be guarded. Hence, by guarding only memory instructions that do not have their related pointer def in their IN set, CARAT nonetheless can still guarantee that all memory instructions are protected.

4.1.2 Tracking.

To track dynamic heap allocations, the CARAT compiler injects a callback to the runtime just after every call to an allocation function (e.g., `call malloc(...)`). The callback notifies the runtime about the address of the new memory block and its size. A similar callback is injected just before every call instruction to a heap deallocation function (e.g., `call free(...)`). The runtime handles static and stack allocations as well. Stack expansions are triggered by a call guard that aborts to the kernel. Static allocations are recorded at program load time.

To track pointer escapes the CARAT compiler identifies each LLVM IR instruction that could store a pointer in memory and injects a callback to the runtime to account for it.

4.2 Runtime

The CARAT runtime is linked into the program at the LLVM IR level, and thus is jointly optimized along with the program and the protection and tracking code that we inject. It plays a central role in tracking, protection, and mapping.

Tracking: The runtime maintains an understanding of the program's static and dynamic memory allocations and pointers. It does so by updating a hard-state data structure, the *Allocation Table*, in response to callbacks made by the injected code. The Allocation Table keeps track of all allocations that the program makes, including static allocations (global and bss variables), and the stack. Each entry in the Allocation Table also contains a set of all variables that contain a pointer, or escape, to the respective allocation, which we call the *Allocation to Escape Map*.

The Allocation Table is currently implemented as a C++ red/black tree whose key is the address of an allocated block

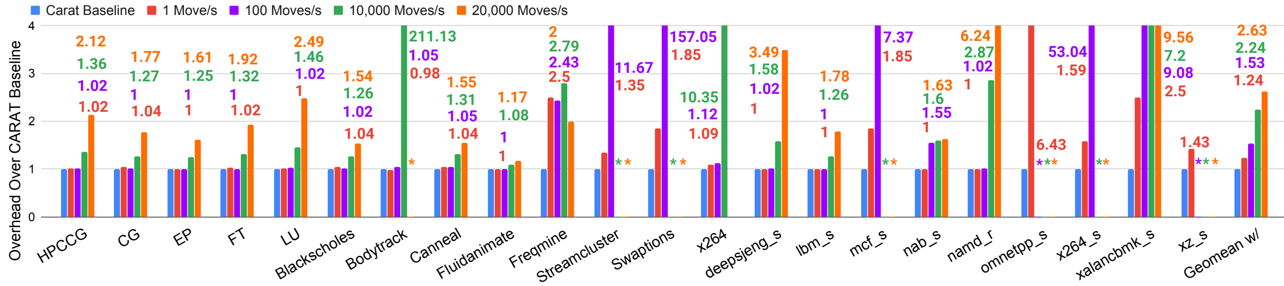


Figure 9. Worst-case page movement overheads. Geomean cut offs: 1 Page/s - 2x, 100 Pages/s - 3x, 10,000 to 20,000 Pages/s - 4x.

of memory, and whose value is the length of the block. The mappings from an allocation to the escapes, the Allocation to Escape Map, is currently implemented as a C++ unordered set and contains pointers to the addresses that contain a pointer copy of the allocation. Total tracking overhead is on the order of 1.9%. (Figure 7).

The process of building the data structures can currently be done in two ways. The first involves updating structures as the program runs, while the second involves updating in batches . We use the first method when we are tracking allocations, and the second when tracking the escapes. The Allocation Map changes slowly, while the Allocation to Escape Map changes quickly. By batching the later, we can mitigate redundant/outdated work.

Protection: The runtime provides a landing zone in which the kernel module places an array of memory regions and their attributes that is ordered by address. A guard can then search for an address range within this array using binary search. For a small enough number of regions, this can be optimized into checking via Intel MPX in a single-cycle.

Mapping: Figure 8 illustrates the operations that occur when a page movement occurs. Steps 4–12 are done by the runtime. Triggered by the kernel, all relevant threads in the program enter a signal handler (dumping their register state on the stack) and synchronize (a "world stop"). They then negotiate a move with the kernel module. Given the final source and destination page(s), the runtime finds all affected allocations, and updates the Allocation Table to reflect their new destinations. It then uses the Allocation to Escape Map to find all pointers into these objects, and patches these pointers to reflect the new addresses. Next, the handler updates all the on-stack register contents that are affected by the move. Next, it moves the data and frees the data at the old location. Lastly, the threads synchronize again, inform the kernel that the operation is complete, and then return from their signal handlers.

4.3 Kernel Module

The primary purpose of the CARAT kernel module is to coordinate the threads of the process and to negotiate a page movement. At one point, our prototype involved having the module compute and execute patches directly (steps 4–12 of Figure 8), but having a signal handler do this within the CARAT runtime is simpler.

When the CARAT executable starts, its runtime registers itself with the kernel module, and the two agree on a communication channel that consists of a shared memory and a signal. On a page movement request (step 1), the module places the parameters into the shared state and then signals every thread within the process (step 2). This has the effect of forcing the threads to dump their register state on their signal handler stacks, where it can be patched. The CARAT signal handler is the entry point to the runtime for the patching process, described earlier.

In step 4, the runtime, executing in the signal handler, and the kernel module negotiate the move. Note that because we manage memory in units of pages, a page movement request on the part of the kernel may expand. For example, an allocation on the page may span to the next page. Since allocations must move in their entirety, the runtime will notify the module that both pages need to be moved. The kernel module can then veto or approve the move. A protection change is done in the same manner, except with no negotiation process.

4.4 Evaluation of Page Movement (Mapping)

We previously described the low memory and run-time costs/overheads for tracking and guarding in the CARAT prototype (Section 3). What remains is to understand the costs of protection and mapping changes in the prototype. Since a protection change involves a simplified version of the process for a mapping change, we focus here on mapping changes (page movements).

Figure 9 shows the run-time overhead for each of our benchmarks when CARAT is actively moving the process's pages as it runs. Here, the program executes while we select a worst-case page to move, invoke the CARAT process for moving it, and repeat. By worst case we mean the runtime selects a page that overlaps the allocation with the most pointer escapes. We do this at different rates. Recall from Table 2 that the typical page movement rate on Linux while running these benchmarks is *much* less than 1/s. At 1/s, the

Table 3. Worst-case Page Movement Costs in Cycles.

Benchmark	Page Expand	Patch Gen. & Exec.	Register Patch	Allocation & Mem. Movement	Prototype Cost	Prototype w/o Expand Cost	Total Cost	Prototype w/o Expand / Total Cost
HPCCG	33755	21637	67	32693	55459	21704	88152	0.2462
CG	35319	19458	134	26121	54911	19592	81032	0.2418
EP	16875	13831	66	11537	30772	13897	42309	0.3285
FT	29110	15385	164	29336	44659	15549	73995	0.2101
LU	33675	14130	321	37349	48126	14451	85475	0.1691
Blackscholes	16686	12276	225	10182	29187	12501	39369	0.3175
Bodytrack	46704	8408	130	76856	55242	8538	132098	0.0646
Canneal	38952	12530	170	8821	51652	12700	60473	0.21
Fluidanimate	8475	2750	120	12137	11345	2870	23482	0.1222
Freqmine	23941	10330	51	21097	34322	10381	55419	0.1873
Streamcluster	31503	16780	60	24733	48343	16840	73076	0.2304
Swaptions	65805	2715	206	214785	68726	2921	283511	0.0103
x264	88122	9992	235	186558	98349	10227	284907	0.0359
deepsjeng_s	39332	10229	221	71227	49782	10450	121009	0.0864
lbm_s	12561	9844	44	7523	22449	9888	29972	0.3299
mcf_s	459841	27561	303	677193	487705	27864	1164898	0.0239
nab_s	7609060	3518	36	9636192	7612614	3554	17248806	0.0002
namd_r	53651	16338	97	70399	70086	16435	140485	0.117
omnetpp_s	444237	1010	18	569406	445265	1028	1014671	0.001
x264_s	888996	15131	332	3162014	904459	15463	4066473	0.0038
xalancbmk_s	493725	3511	114	922548	497350	3625	1419898	0.0026
xz_s	43306	6158	43	76809	49507	6201	126316	0.0491
Geo. Mean	**67467**	**9051**	**110**	**79214**	**88097**	**9206**	**178647**	**0.0515**

overhead of CARAT page movements on program run-time is negligible to small. We also show overheads for rates of 100/s (> 100× the measured move rate), and 10,000–20,000/s (almost double our maximum measured allocation rate).

At 10,000–20,000 page moves per second, the average overhead is between 124% and 163%. It is important to note here that we are doing *movements* at these rates—in fact, a page allocation is not the same as a page movement, as the sum total of CARAT processing that would occur for actual page allocation would be a single Allocation Table update to reflect the program allocation that triggered the page allocation. The overhead of actual page allocation is **much much** lower.

For some of the applications, moving memory at rates in excess of the expected rate causes unacceptable execution times. For example, Bodytrack has a measured expected page movement rate of 0.007/s, but actually can operate with <5% overhead even at 100/s (four orders of magnitude higher than the measured rate). It is only when we escalate the rate to 10,000/s (six orders of magnitude) that the overhead becomes unacceptable and measurement becomes infeasible (Bodytrack at 10,000/s runs for 14.5 hours compared to a baseline of 4 minutes). The asterisks in the figure indicate scenarios in which measurement was infeasible for this reason. The important point is that the overhead of moving pages at expected rates (<1/s) is *negligible*. Additionally, the overheads shown are conservative, for reasons we will discuss next.

Table 3 breaks down the average cost of a CARAT page movement into its constituent parts for each application (the rows). The columns show the number of cycles spent in the following: "Page Expand" is the time to find all the allocations associated with the target page, and then expand to a set of surrounding target pages that have no allocation that breaches the boundaries of the set. "Patch Gen. & Exec" is the time to find and update all the escapes from allocations on the target page set. Note that this is typically quite small. The always minuscule "Register Patch" time is spent patching registers that happen to contain an escape. "Allocation and Movement Cost" is the time to allocate a new memory block for the target page set and copy the data to it. "Prototype Cost" sums the first three columns, and is the cost of the prototype, ignoring the cost of allocating and moving the memory (which is not counted for paging either.) "Prototype w/o Expand Cost" further removes the "Page Expand" cost. "Total Cost" is the measured overhead.

The "Page Expand" cost exists and the correlated "Allocation and Movement Cost" is often large because of the impedance mismatch of tracking at allocation granularity and moving at page granularity. "Prototype w/o Expand / Total Cost" is the fraction of the total cost that is not due to this mismatch. Note that this is typically quite small. If CARAT operated entirely at the granularity of allocations (described in more detail in Section 6), we would expect the overheads of Figure 9 to be scaled by these fractions. We would expect a 95% average (geometric mean) reduction.

5 Related Work

We have integrated broadly-scoped related work throughout the paper. A description of the challenges of modern TLBs and the community's efforts to address them is at the start of Section 3. Here we focus on work that is more tightly scoped

to the CARAT model.

Although we considered the CARAT model from the perspective of replacing paging, it can more generally be considered to be a form of automatic handle-based memory management. Manual handle-based memory management was commonplace in early, highly successful PC OSes, such as Windows [64] and pre-2001 MacOS [5, Volume 2, Chapter 1], and remains in use today on some embedded hardware. Two clear advantages are arbitrary granularity of operation and physical addressing. Two clear disadvantages are a lack of protection and reliance on the programmer to use the handle interface correctly, which is almost as challenging as doing locking correctly. CARAT attempts to remove these disadvantages, to make protected handles automatic.

Automatic handle-based memory management with protection dates back at least to work to leverage the descriptor-based operation of the Burroughs B5000 [18],[56, Chapter 2] by restricting programmers to specific high-level languages. More influential to our thinking is the IBM 801 [66], which combined physical addressing, a heavily restricted high-level language (PL.8), and a trusted compiler as the basis for protection ([66, pp. 240]). The most recent work in this vein that we are aware of is the Software Isolated Process (SIP) [41] of the Singularity OS [42]. 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 to these models, CARAT's goal is to support arbitrary languages and code through the IR and concomitant transforms of a modern compiler.

The use of the compiler to implement software protections is not a new concept. EffectiveSan [27] 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 or page/object movement features. It is further evidence of the feasibility of using software checking for protection. CARAT can also be thought of as extending Software Fault Isolation (SFI) [21, 33, 67, 77] to all user programs regardless of trust. In fact, many of the innovations in SFI are also potential candidates for further optimization of CARAT, and some of the optimizations in CARAT could potentially be deployed in SFI. Finally, another alternative to achieving protection is proof-carrying-code (PCC) [59–61] 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 the compiler trying to generate proofs about the safety of the code.

Kadayif et al. [48, 49] developed compiler/TLB co-designs that make it possible to skip the normal TLB lookup path when the virtual address is within a set of shortcuts installed by the compiler-generated code. Considerable power and energy could be saved. Like CARAT, this work heavily leverages the compiler to reduce requirements for the hardware. However, it still requires a TLB to function. In contrast, CARAT's goal is to achieve purely physical addressing, avoiding virtual memory support in hardware altogether.

DINAMITE [57] implements a form of CARAT's allocation tracking to support bug-finding, but it is limited to dynamic memory allocations (malloc). CARAT tracks *all* allocations and escapes, and uses this information for memory management.

6 Conclusions and Future Work

It is time to revisit software-based memory management in this era of the every-increasing complexity of the hardware support structures that underlie traditional virtual memory. To support this general assertion, we have made a case for compiler- and runtime-based address translation (CARAT), a specific model in which only physical addresses are used. We contrasted CARAT with the traditional paging model and explained its advantages and disadvantages in Section 2. We then demonstrated CARAT's feasibility through empirical means in Section 3. Finally, in Section 4 we described and evaluated a prototype CARAT system, showing that it can provide the benefits of the traditional model (protection and mapping)—with no hardware support—with low overhead.

Kernel-level Implementation: Our prototype has several limitations that we are now working on ameliorating. The first is that it is a largely user-level implementation that operates on top of Linux's complex paging apparatus. We are currently building a second version of CARAT within the Nautilus kernel [35, 36], a framework in which x64 paging is ignored using identity-mapping, and all code runs in kernel mode for performance reasons. Our intent is to reintroduce protection in this kernel using CARAT.

JITs and Other Language Models: Our prototype does not support self-modifying code, inline assembly, or other language features or concepts that do not flatten into the LLVM middle-end. We do not believe this is an intrinsic limitation of the CARAT model, but rather a question of pushing the model into LLVM front- and back-ends, as well as into run-time code generators such as a JIT. However, to do this will require new methods for optimizing protection. We are studying this matter.

Allocation Granularity: A big limitation of our prototype is that it operates with pages instead of the only natural allocations done by the program. As a consequence, it must obey both the page-level semantics of the Linux kernel, and the allocation-level semantics of the application. This necessitates the negotiatiation with the kernel that will not break the allocation-level semantics. We are working on a version of CARAT that discards the page abstraction, eliminating negotiation and allowing allocation granularity management.

References

[1] 2010. The Data Deluge. *The Economist* (25 February 2010). https://www.economist.com/leaders/2010/02/25/the-data-deluge

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. *Compilers, principles, techniques*. Addison Wesley.

[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] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 27–39. https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit

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

[6] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 273–287. https://doi.org/10.1109/PACT.2017.38

[7] Julian Bangert, Sergey Bratus, and Rebecca Shapiro andSean W. Smith. 2013. The Page-Fault Weird Machine: Lessons in Instruction-less Computation. In *Proceedings of the 7th USENIX Workshop on Offensive Technologies (WOOT)*.

[8] Richard Barrett, Michael Heroux, P. Lin, C. Vaughan, and A. Williams. 2011. Mini-applications: Vehicles for Co-Design. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC 2011)*.

[9] 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)*.

[10] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2012. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, OR) *(ISCA '12)*. 297–308.

[11] A. Bhattacharjee. 2017. Preserving Virtual Memory by Mitigating the Address Translation Wall. *IEEE Micro* 37, 5 (Sep. 2017), 6–10. https://doi.org/10.1109/MM.2017.3711640

[12] A. Bhattacharjee. 2018. Breaking the Address Translation Wall by Accelerating Memory Replays. *IEEE Micro* 38, 3 (May 2018), 69–78. https://doi.org/10.1109/MM.2018.032271063

[13] A. Bhattacharjee, D. Lustig, and M. Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 62–63. https://doi.org/10.1109/HPCA.2011.5749717

[14] A. Bhattacharjee and M. Martonosi. 2009. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. 29–40. https://doi.org/10.1109/PACT.2009.26

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

[16] Johnnie Birch, Robert van Engelen, and Kyle Gallivan. 2004. Value range analysis of conditionally updated variables and pointers. *proceedings of Compilers for Parallel Computing (CPC)* (2004), 265–276.

[17] R. E. Bryant, R. H. Katz, and E. D. Lazowska. 2008. Big-Data Computing: Creating revolutionary breakthroughs in commerce, science, and society. *White Paper, Computing Community Consortium Committee, Computing Research Association* (22 December 2008). http://cra.org/ccc/resources/ccc-led-whitepapers/

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

[19] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. 2014. HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 217–228. http://dl.acm.org/citation.cfm?id=2665671.2665705

[20] S. Campanoni, T. M. Jones, G. Holloway, G. Y. Wei, and D. Brooks. 2012. HELIX: Making the Extraction of Thread-Level Parallelism Mainstream. *IEEE Micro* 32, 4 (July 2012), 8–18. https://doi.org/10.1109/MM.2012.50

[21] 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.

[22] Jonathan Corbet. 2008. Memory Management Notifiers. https://lwn.net/Articles/266320/.

[23] Jonathan Corbet. 2017. A Last-Minute MMU Notifier Change. https://lwn.net/Articles/732952/.

[24] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[25] Peter Dinda and Akhil Guliani. 2017. Dark Shadows: User-level Guest/Host Linux Process Shadowing. In *Proceedings of the 5th IEEE International Conference on Cloud Engineering*.

[26] Christopher Domas. 2015. The Memory Sinkhole. In *Proceedings of Black Hat USA 2015*.

[27] 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)*.

[28] ECMA 2005. *Standard ECMA-335 Common Language Infrastructure (CLI)* (3rd ed.). ECMA, Rue du Rhone 114 CH-1204 Geneva. http://www.ecma-international.org/publications/standards/Ecma-335.htm

[29] Magnus Ekman, Per Stenström, and Fredrik Dahlgren. 2002. TLB and Snoop Energy-reduction Using Virtual Caches in Low-power Chip-multiprocessors. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design* (Monterey, CA) *(ISLPED '02)*. 243–246.

[30] Dongrui Fan, Zhimin Tang, Hailin Huang, and Guang R. Gao. 2005. An Energy Efficient TLB Design Methodology. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design* (San Diego, CA) *(ISLPED '05)*. 351–356.

[31] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[32] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. 2016. Range Translations for Fast Virtual Memory. *IEEE Micro* 36, 3 (May 2016), 118–126. https://doi.org/10.1109/MM.2016.10

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

[34] Bolei Guo, Matthew J Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I August. 2005. Practical and accurate low-level pointer analysis. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 291–302.

[35] 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)*.

[36] Kyle C. Hale and Peter A. Dinda. 2015. A Case for Transforming

Parallel Runtime Systems Into Operating System Kernels (short paper). In *Proceedings of the 24th International ACM Symposium on High Performance Parallel and Distributed Computing, (HPDC 2015)*.

[37] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. 2011. Toward Dark Silicon in Servers. *IEEE Micro* 31, 4 (July-August 2011), 6–15.

[38] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/3173162.3173194

[39] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. *Improving Performance Via Mini-Applications.* Technical Report SAND2009-5574. Sandia National Labs.

[40] C. Hetland, G. Tziantzioulis, B. Suchy, K. Hale, N. Hardavellas, and P. Dinda. 2019. Prospects for Functional Address Translation. In *Proceedings of the 27th IEEE International Conference on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[41] 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.

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

[43] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel.

[44] Intel Corporation 2019. *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation.

[45] 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.

[46] Nick P. Johnson, Jordan Fix, Stephen R. Beard, Taewook Oh, Thomas B. Jablin, and David I. August. 2017. A Collaborative Dependence Analysis Framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, Piscataway, NJ, USA, 148–159. http://dl.acm.org/citation.cfm?id=3049832.3049849

[47] Toni Juan, Tomas Lang, and Juan J. Navarro. 1997. Reducing TLB Power Requirements. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design* (Monterey, CA) *(ISLPED '97)*. 196–201.

[48] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam. 2004. Compiler-directed physical address generation for reducing dTLB power. In *Proceedings of the IEEE International Symposium on the Performance Analysis of Systems and Software (ISPASS)*. 161–168.

[49] 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.

[50] Vasileios Karakostas, Jayneel Gandhi, Adrian Cristal, Mark Hill, Kathryn McKinley, Mario Nemirovsky, Michael Swift, and Osman Unsal. 2016. Energy-efficient address translation. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 631–643.

[51] George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. 2018. An efficient data structure for must-alias analysis. In *Proceedings of the 27th International Conference on Compiler Construction.* ACM, 48–58.

[52] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. 1962. One-Level Storage System. (April 1962).

[53] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[54] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanovic. 2019. Keystone: An Open Framework for Architecting TEEs. arXiv:1907.10119 [cs.CR]

[55] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization.* ACM, 126–137.

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

[57] Svetozar Miucin, Conor Brady, and Alexandra Fedorova. 2016. End-to-end Memory Behavior Profiling with DINAMITE. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*.

[58] Vaivaswatha Nagaraj and R. Govindarajan. 2015. Approximating Flow-sensitive Pointer Analysis Using Frequent Itemset Mining. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (San Francisco, California) *(CGO '15)*. IEEE Computer Society, Washington, DC, USA, 225–234. http://dl.acm.org/citation.cfm?id=2738600.2738629

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

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

[61] 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)*.

[62] Mark Oskin and Gabriel H. Loh. 2015. A Software-Managed Approach to Die-Stacked DRAM. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. 188–200. https://doi.org/10.1109/PACT.2015.30

[63] Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. 2018. SEESAW: Using Superpages to Improve VIPT Caches. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 193–206.

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

[65] Kiran Puttaswamy and Gabriel H. Loh. 2006. Thermal Analysis of a 3D Die-stacked High-performance Microprocessor. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI* (Philadelphia, PA) *(GLSVLSI '06)*. 19–24.

[66] 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.

[67] 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).

[68] Semiconductor Industry Association and Semiconductor Research Corporation. 2015. Rebooting the IT Revolution: A Call to Action. *NSF Workshop Report* (September 2015).

[69] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. 180–192. https://doi.org/10.1109/ISCA.2018.00025

[70] Avinash Sodani. 2011. Race to Exascale: Opportunities and Challenges. In *Keynote at the 44th Annual IEEE/ACM International Symposium on*

Microarchitecture (Porto Alegre, Brazil) *(MICRO 44)*.

[71] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 32–41.

[72] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. ACM, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

[73] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

[74] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. 2012. Build code analysis with symbolic evaluation. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 650–660.

[75] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*. Springer, 489–510.

[76] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. 2011. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 340–349. https://doi.org/10.1109/PACT.2011.65

[77] 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)*.

[78] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-aware TLBs. In *Proceedings of the ACM/IEEE 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. 698–710. https://doi.org/10.1145/3307650.3322223

[79] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek Bhattacharjee. 2017. Hardware Translation Coherence for Virtualized Systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. 430–443. https://doi.org/10.1145/3079856.3080211

[80] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by Level: Making Flow- and Context-sensitive Pointer Analysis Scalable for Millions of Lines of Code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/1772954.1772985