Toast: A Heterogeneous Memory Management System

Programmability, Portability, Protection, and Performance

Maurice Bailleu*
Huawei Research
Edinburgh, United Kingdom

Dimitrios Stavrakakis
Technical University of Munich &
The University of Edinburgh
Munich, Germany

Rodrigo Rocha
Huawei Research
Edinburgh, United Kingdom

Soham Chakraborty
TU Delft
Delft, Netherlands

Deepak Garg
MPI-SWS
Saarbruecken, Germany

Pramod Bhatotia
Technical University of Munich
Munich, Germany

Abstract

Modern applications employ several heterogeneous memory types for improved performance, security, and reliability. To manage them, programmers must currently digress from the traditional load/store interface and rely on various custom libraries specific to each memory type, thus introducing programmability, performance, portability, and protection challenges.

To overcome these challenges, we propose Toast, a compiler-based approach that offers *a simplified programming model* based on the established load/store interface along with programmable error-handling and memory consistency enforcement mechanisms and a protection library for memory safety.

We implement Toast in the Clang/LLVM compiler framework accompanied by a runtime library, employing software storage capabilities and hardware-based protection mechanisms. Our evaluation based on four applications, which use heterogeneous memory types, shows that Toast improves the *programmability*, *portability*, and *protection* of applications, while offering *performance* on par with a hand-optimized version of the application.

Keywords

Heterogeneous memory, Memory management, Memory safety, Memory protection, Programmability, Portability, Performance

ACM Reference Format:
Maurice Bailleu, Dimitrios Stavrakakis, Rodrigo Rocha, Soham Chakraborty, Deepak Garg, and Pramod Bhatotia. 2024. Toast: A Heterogeneous Memory Management System: Programmability, Portability, Protection, and Performance. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Southern California, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3656019.3676944

1 Introduction

Modern applications employ heterogeneous memory types for performance, security, reliability, and domain-specific computing [5, 25, 44, 50, 53]. These heterogeneous memory systems span almost all aspects of the stack, e.g., network (RDMA [61]/DPDK [6]), storage (SPDK [10]/persistent memory [8]), secure enclaves [9], and accelerators [13]. Furthermore, NUMA [67] and the introduction of CXL-capable [37] devices further create a memory hierarchy due to their differences in access patterns and latency.

In theory, heterogeneous memory subsystems are accessible via DMA, i.e., allowing read and write directly from and to memory regions. However, in practice, these memory subsystems are accessed via a range of subsystem-specific auxiliary libraries, which force programmers to digress from the traditional *load*/*store* interface to access byte-addressable memory regions [53, 61]. This library-based approach leads to four significant challenges for heterogeneous memory management ("The 4P challenges"): *(i)* Programmability, *(ii)* Portability, *(iii)* Protection, and *(iv)* Performance.

Programmability challenges arise because programmers must learn and understand the APIs and libraries for each memory technology separately [56]. Moreover, the library-based approach introduces *portability* challenges when the underlying hardware evolves with the introduction of new technologies. Current approaches require a complete re-design of the software system to adapt to a new heterogeneous memory subsystem. Programmers have to rewrite their code to a great extent using different access patterns, libraries, and APIs. This is a cumbersome and error-prone task.

Furthermore, heterogeneous memory management introduces *protection* challenges as a programmer juggles different memory regions. A potential error during application development can lead to undesired code behaviors, such as sensitive information leakage to untrusted devices or mistakenly persisting temporary data.

Lastly, developers are pressed to achieve optimal *performance*, which is tough when they must deal with different libraries and their varied interfaces. At the same time, in the context of concurrent programs, it is important to ensure that the program remains correct under subtle memory consistency and persistency models [57, 58].

We consider the following problem: How do we define a heterogeneous memory programming interface, which provides *programmability*, i.e., an easy-to-learn and understand interface, *portability*, i.e., minimizing the effort involved in changing underlying technologies, *protection* against accidental data sharing between different memory regions, while providing *performance* on par with or exceeding existing libraries?

*This work was done when affiliated with The University of Edinburgh

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.
PACT '24, October 14–16, 2024, Southern California, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676944

```
event_loop(FILE log)
 int svr, clt;
 struct sockaddr_in svr_addr, clt_addr;
 svr = socket(AF_INET, SOCK_STREAM, 0);
 svr_addr = init_server_addr();
 bind(svr, &svr_addr, sizeof(svr_addr));
 listen(svr, 3);
 clt = accept(svr, &clt_addr, sizeof(clt_addr));
 char buf[msg_sz];
 for (;;)
  //Waiting for data to arrive
  read(clt, buf, msg_sz);
  if (is_write(buf))
   //Writing to storage device
   write(log, buf, msg_sz);
  //Acknowledge to clt
  write(clt, rsp, rsp_sz);
```

Listing 1: Using POSIX API to write a network stream to a file.

```
event_loop(uint64_t * log)
 rx, tx = get_queues()
 for (;;)
  //Waiting for data to arrive
  poll(rx);
  char * buf = get_buf(rx);
  if (is_write(buf))
    //Storage pointer
    uint64_t * log = next_log(buf, log_sz);
    //Writing to storage device
    *log = buf << 32 | len(buf);
    //Makes writing visible
    clflush(log);
    //Free buffer
    char * extra = next_free_buf();
    swap(buf, extra);
  write_response(tx);
```

Listing 2: .]Accessing network and storage device with DMA [53].

```
event_loop(T_log log)
 rx, tx = get_queues();
 for (;;)
  //Waiting for data to arrive
  T_net buf = get_buf(rx);
  if (is_write_request(*buf))
   //Writing to storage device
   log[idx++] = buf << 32 | len(buf);
  write_response(tx)
```

Listing 3: Toast version of the applications in Listings 1 & 2.

Overall, due to the heterogeneous nature of the devices that leverage DMA [37], designing a unified interface comes with inherent challenges. While heterogeneous memory can be accessed over the memory bus, due to device-specific implementations, additional actions might be required, including reading and writing to specific memory addresses before or after the data transfer and additional cache flushes. Furthermore, heterogeneous devices have different sources of errors and faults, leading to vastly different error-handling procedures. Moreover, these heterogeneous devices present a range of memory consistency and persistency semantics, incurring correctness issues for concurrent programs.

Current state-of-the-art approaches for API unification [13, 22, 42, 49, 70] mainly target specific device classes, e.g., GPUs/FPGAs, and introduce one API per use-case, hindering programmability and portability. Additionally, further approaches aim to optimize memory accesses in shared memory systems [14, 19, 35, 41]. However, they neither provide support for different memory layouts and access patterns nor explicit memory protection.

Notably, the emerging CXL technology [37] enables direct memory access in heterogeneous memory systems by relying on specialized CXL PCIe-attached hardware bridge devices. Therefore, CXL systems are restricted to the memory systems supporting these specialized hardware bridge devices. Moreover, CXL systems do not support various memory types, such as secure enclave memory, NIC memory (DPDK), SSD/NVMe memory (SPDK), etc. Finally, CXL does not inherently provide any built-in protection mechanism.

To this end, we propose Toast, *a simplified, generic programming model* based on the established load/store interface combined with an error handling mechanism and a protection library to isolate different memory regions. Toast consists of a compiler, based on Clang/LLVM [36, 68], and a run-time component. The compiler component lowers a high-level load/store interface to our lower-level run-time component. As part of our run-time component, we introduce the abstraction of ToastPtr, a pointer associated with a specific memory region that is manipulated using a single uniform interface in the programming language. Under the hood, considering the underlying memory consistency and persistency models, the Toast run-time transparently translates interface calls to region-specific library calls. Toast further provides optional programmable error-handling callback hooks to enable the programmer to handle memory device-specific errors.

ToastPtrs lift library/device-specific calls to a device–independent load/store interface at the language level. Thus, Toast eases *programmability* by reducing the technology-specific knowledge required from the developer. ToastPtrs are given semantics via a configuration file, which can be easily adapted for different technologies, thus improving *portability*. Importantly, once a configuration for a technology is correctly designed, the developer can reap its benefits across applications without additional effort.

Lastly, Toast incorporates two protection mechanisms to prevent programming errors related to unintended data sharing between memory regions; *(i)* a software-based mechanism designed as capability storage and *(ii)* a hardware-based mechanism using Intel's Memory Protection Keys (MPK) [39].

We evaluate Toast on four representative applications that access five different memory types via device-specific libraries: (a) a secure in-memory key-value (KV) store [25], (b) a replication protocol [23, 73], (c) a persistent shared log application [4] and (d) a persistent KV store [69]. Our evaluation shows that Toast improves *programmability*, *portability*, and *protection* of applications while incurring a mean *performance* overhead of 4.9 % compared to hand-optimized code. Toast has been made publicly available (https://github.com/TUM-DSE/Toast).

2 Motivation: The 4P Challenges

Modern applications employ multiple heterogeneous memory regions. These regions differ in trust (TEEs), latency (NUMA, CXL), and persistence (memory-mapped files and PM). Furthermore, applications often use DMA-capable devices (e.g., SPDK for SSDs, DPDK for NICs), which comprise another source of heterogeneity in the memory system. These devices expect specific data structures and read/write patterns to reach their full potential. Thus, adopting a new technology often requires that developers invest time to learn new device-specific library APIs and leads to the rewriting of major parts of an application to achieve the desired performance. Such changes are invasive, time-consuming, and error-prone.

For example, Listing 1 shows a code path that accepts network packages and writes them to a log on a file system using the POSIX

Table 1: Memory types

Memory Types	Persistence	Encryption	Consistency type	Access patterns	Library required
DRAM	Volatile	Optional	Architecture dependent	Random	None
Persistent memory	Non-volatile	Optional	Configurable	Random	Yes (e.g., PMDK [8])
Kernel-bypass storage	Non-volatile	Optional	Configurable	Sequential	Yes (e.g., SPDK [10])
Enclave memory	Volatile	Encrypted	Architecture dependent	Random	Yes (e.g, Intel SGX SDK [86])
NIC memory	Volatile	Optional	Protocol-dependent	Sequential	Yes (e.g., DPDK [6])
GPU memory	Volatile	Optional	Architecture dependent	Sequential, highly-parallel	Yes (e.g., CUDA [77])

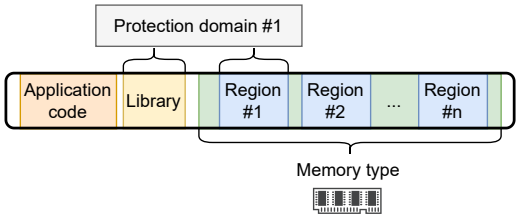

Figure 1: Virtual address space layout in Toast

API. Listing 2 shows the same program based on non-blocking communication between devices and CPU by introducing heterogeneous memory subsystems, i.e., Remote Direct Memory Access (RDMA) and Persistent Memory (PM).

These modern technologies use different interfaces, which results in little code reuse between implementations. They also have different abstractions for the user; on the one hand, POSIX provides common abstractions and on the other hand, the user has to manually add polling and cache-line flushes to ensure correct code behavior in the second implementation. Transitioning from one technology to another requires addressing the following challenges:

Programmability. Programmers must familiarize themselves with the libraries used in their application.For concurrent programs, developers must be aware of the concurrency primitives such as the atomics in C/C++ or the locking mechanisms to write correct and efficient concurrent programs. Furthermore, each memory type needs its memory allocator to ensure that the memory layout follows the specifications of the underlying device. This imposes significant programming challenges as each library creates pointers to its respective memory regions, which can be stored and later reused from other application parts. Pointers of two different libraries are *conceptually* distinct. Still, they are not distinguished by the language's type system and can, therefore, be inadvertently confused by the programmer, resulting in reliability and security vulnerabilities.

Portability. Adapting modern systems to use new technologies is challenging. The APIs to handle each memory type or device might be designed for or integrated with the application's logic. This can be mitigated through an abstraction layer. However, the design of such a layer is a non-trivial task. On top of that, the developers must carefully perform the deployment of an abstraction layer throughout the application. Thus, an application gets strongly tied to a specific abstraction layer, hindering portability.

Performance. The appeal of modern heterogeneous memory systems is their superior performance compared to conventional abstractions. However, their focus on performance often leads to a tight coupling of the application logic with the technology to take advantage of techniques like zero-copy or asynchronous calls. Moreover, these devices offer different memory consistency models with varying performance properties. This requires careful optimizations, which are often tedious and error-prone.

Protection.

Efficient resource access management is important when dealing with multiple memory regions. Currently, developers are responsible for correctly handling the pointers returned by various libraries. Calling a library with a pointer from another can lead to information leaks (e.g., revealed keys) or memory corruption (e.g., buffer overflow attacks). Therefore, providing a form of memory region-level isolation is crucial to protect against pointer misuse, i.e., a mechanism to ensure correct access to different memory regions only via their respective library pointers.

3 Overview

Programming languages offer a well-known abstraction for accessing local memory, namely, the *load/store* model. Pointers are a central part of this model, providing ease of programmability when communicating with the local memory.

Device libraries often expose pointers to DMA'ed memory regions to programmers to enable zero-copy operations. However, this requires developers to use specific functions to access memory safely. This, in turn, results in inconsistencies in the developer's *mental* model, as they have to interact with pointers in vastly different ways. Additionally, the idiosyncrasies of each memory type (e.g., persistence granularity) as well as the different access patterns they require (e.g., writing to network queues) make both the unification of the various memory types' interfaces and the memory type-specific error-handling process quite challenging. To highlight the level of complexity, we present typical memory type examples and their unique characteristics in Table 1.

3.1 The Toast Programming Model

To provide similar levels of *programmability*, *portability*, and *performance* as those offered by local memory, we propose Toast. Toast introduces the concepts of *memory type*, *memory region* and *protection domain*. A *memory type* is a set of address ranges in an address space, which is accessed uniformly via a single set of API calls. A memory type may be mapped to *RAM*, *devices*, or *special memory* like PM or enclave memory. A *memory region* is an address range in a memory type, e.g., the Tx/Rx queue for a NIC. A *protection domain* is a set of access rights to address range mappings. Figure 1 shows the relationship between the different concepts.

Toast *refines* (annotates) the pointer type with the memory type, thus creating a separate pointer type for every memory type. It transparently injects code, calling the corresponding Toast runtime library (see § 4.5) for the annotated pointers to support normal dereferencing while guaranteeing the right order of library calls. On top of this, Toast provides memory *protection* mechanisms to prevent accidental memory mishandling due to programming errors. For the concurrent programs Toast provides an option to specify the consistency model to be preserved for code generation.

Table 2: Toast APIs and error handling routines

Runtime library APIs	Description
write()	Writes data to memory
read()	Fetches data from memory
err_handler(stack,...)	Called by the Toast runtime library in case of an erroneous memory access
Error handling routines	
retry()	Retry the last operation
continue()	Handler corrected the data
abort()	Cleans up the current context, and returns an invalid pointer

ToastPtr. Toast's programming model is based on the fact that devices interact with the CPU over the memory bus with load/store operations. Most programming languages offer these operations in the form of assignments, e.g., the operator '=' in C. However, developers cannot use them directly when they interact with devices, as devices are usually accessed via specific low-level libraries.

A ToastPtr is a pointer to a memory region. ToastPtrs contain, in addition to the address, the memory type as well as the protection domain, which enable Toast to check memory safety violations when a ToastPtr is dereferenced. The Toast compiler transforms pointer (de)references to read and write calls to the Toast runtime library, which acts as a proxy layer between ToastPtr operations and the low-level runtime library for different devices. Moreover, for the concurrent programs, based on the consistency models specified in the configuration, the Toast compiler generates the required library calls that enforce the desired consistency model.

Error handling. Error handling is an integral part of any application. Pointers are notoriously bad at communicating errors as they only have two states: the invalid null and the valid non-null. Toast allows developers to register error handlers. In case of an error (e.g., a device initialization error, a failed integrity check), Toast calls the respective error handler with a pointer to the program's call stack, a source code position, error information from the underlying device, and a pointer to internal memory, *e.g.,* transmission buffers, hash values, etc., which can be used in the error handler to recover from an error or to collect debug information.

An error handler returns one of the following states to Toast (see Table 2): ***retry*** signals Toast to retry the operation; ***continue*** means that the handler corrected the error in the internal data, and Toast returns the corrected data to the caller; ***abort*** instructs Toast to clean up the current action and return an invalid pointer.

Device configuration. In Toast, the developer defines the device configuration once. It can then be reused across different applications seamlessly, provided that the ToastPtrs referring to this device are annotated correctly in the code. Thus, different projects can adopt and combine existing configurations, promoting generality and re-usability. Precisely, the device configuration includes *(i)* the memory type, *(ii)* the name of the memory type's proxy library, and *(iii)* the header files to locate the appropriate library functions.

Workflow. Figure 2 presents the flow of a Toast application and Table 2 shows the Toast API after the compiler transformation. The developer provides the Toast compiler with the code and a configuration file with the information for interacting with memory types. During compilation, the pointer (de)reference operations are transformed into *read()* and *write()* calls to the Toast runtime library ①. A dereference of a ToastPtr is lowered to a call to the Toast proxy library ②, which performs the necessary checks. Then, if the user enables the Toast protection mechanism, the appropriate protection checks and access rights management operations are executed ③ and Toast calls the underlying library ④.

If the library returns normally, the Toast proxy provides the DMA'ed area to the user's code. If an error occurs, the proxy library informs the error handler registry ⑤, collects the necessary information, and triggers an error handling event ⑥. The error handler returns to the error registry either a *retry*, *continue* or *abort* indication, which is forwarded to the proxy library. Finally, the proxy library returns with either a valid or an invalid pointer.

Toast requires a device-specific implementation of its proxy library. However, this effort has to be done once per DMA-capable device and is reusable across all applications.

3.2 System Model

Fault model. We assume data is shared between different software components, *e.g.,* libraries. However, not all components are allowed to access all data. We consider each memory region dedicated to a component part of a protection domain. We also assume that accidental sharing of information across protection domains (e.g., without explicit pointer casting) is a critical fault. Importantly, Toast assumes that programmers do not have malicious intent and only prevent inadvertent programming errors.

Programming model. Toast is designed for heterogeneous memory types with different access semantics. We assume that the system has a unified address space, i.e., the address does not contain information about the type of memory it refers to. In the underlying system, memory is accessed via memory type-specific interfaces, e.g., device-specific library APIs or specific CPU instructions, not via direct, common assignment operations. Note that when porting an application to Toast, its logic remains unchanged. The developer only needs to adapt memory accesses to use ToastPtr and incorporate device-specific error handling. If the application is optimized to use specific hardware features (e.g., XPLine for PM), Toast preserves the optimizations as the access patterns remain intact. Importantly, the current Toast prototype targets C/C++ applications. However, Toast's techniques are language-independent and they can be implemented at the LLVM IR level. In this way, Toast can still be applied to languages providing high-level abstractions.

Type system. Toast modifies existing types of the type system. However, since the original type can still exist in the code base, Toast preserves the existing ones but also enriches the type collection of the language. Thus, it extends the language by embedding information that results in transformed types while maintaining the original ones for compatibility purposes.

Memory model. ToastPtr does not enforce any memory ordering specification for the shared memory accesses. Therefore, ToastPtr seamlessly follows the memory consistency models enforced by the underlying C/C++ concurrency primitives. Our framework allows specifying a memory model from sequential consistency, release-acquire, or relaxed by the programmers. Based on the specification, ToastPtr enforces memory orders of its internal memory accesses.

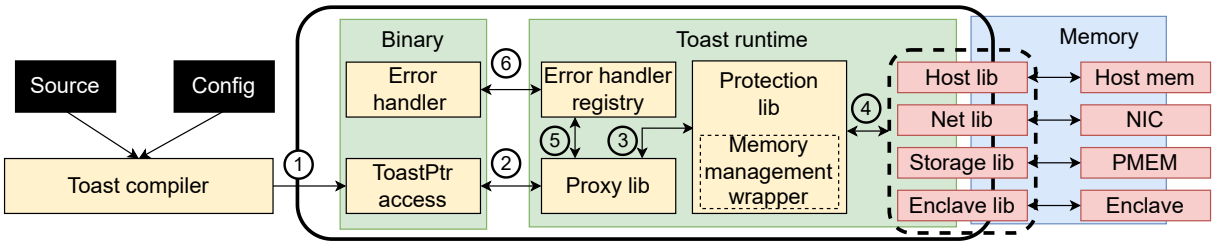

Figure 2: Toast overview: The compiler creates the binary and links it with the runtime libraries ①. The binary dereferences a ToastPtr ②, which results in the proxy library communicating through the protection library ③ with the devices ④. In an error case, the proxy library informs the error handler registry ⑤, which collects information for the handler and calls it ⑥.

3.3 Example Revisited

We illustrate the Toast programming model (Listing 3) using the simple example from Section 2 that involves a network and a storage interface. In this example, the programmer has to perform network and storage management with different interfaces and semantics. We observe that the actual task concerns only copying data received from the network (e.g., sockets, NIC) to the storage (e.g., SSD, PM).

Listing 3 shows the example code of Listing 1 and Listing 2 transformed with Toast. It abstracts away the POSIX APIs, and the RDMA and PM library calls as well as the device-specific operations (e.g., polling, flushing). The intended *logical functionality* becomes decoupled from device-specific library calls, allowing the programmer to focus only on the logical operations when programming and debugging. Toast relies on the configuration files to rewrite the simplified code and produce the expected correct binary.

Config(uration) file. Toast encapsulates the device configuration in a config file that contains a set of rewriting rules for device-specific library calls. Each pointer type is associated with a header file that incorporates implementing the desired operations (e.g., pointer dereference, store operation). An example configuration file for Listing 3 is shown below:

```
"network" : {
  "header" : "toast_runtime/toast_network_ptr.h",
  "type" : "ToastNet:NetworkPtr",
  "consistency" : "[SC/RA/Relaxed/NA]"
},
"log" : {
  "header" : "toast_runtime/toast_log_ptr.h",
  "type" : "ToastLog:LogPtr",
  "consistency" : "[SC/RA/Relaxed/NA]"
}
```

The configuration file needs to be created once per device type and can be shared between projects. The underlying libraries can also be modified depending on the use case. The Toast compiler parses this file and considers the user annotations and the ToastPtr attributes that provide the compiler with the device and operation information. Then, the compiler replaces the respective operations based on the rewriting rules given in the configuration header files.

4 Design & Implementation

4.1 Toast Compiler

Toast requires pointer annotations for individual memory types. We extend the list of attributes to support the namespace toast, which contains pointer annotations, and the attribute toast::event for event handler callbacks, i.e., error handling. Each attribute takes an additional user-defined parameter, which further specializes in the type of the ToastPtr and event handler. Thus, Toast lifts knowledge of the pointer's memory type into the type system.

Since different devices expose different APIs, the compiler has to deal with various interfaces. The programmer provides a json configuration file, which instructs the compiler to replace specific Toast annotations with calls to their respective libraries, which are placed in an internal but easily extendable database.

The Toast compiler is built into the clang frontend (v. 16). Toast leverages clang's code generation to hook its plugin and provide warnings and errors in case of a misuse of its attributes. It initially collects a list of pointers annotated by the programmer with an attribute introduced by Toast. Then, it internally changes the types of these pointers to ToastPtr. Thereafter, it scans the AST for uses of the pointers, differentiating between read and write accesses. This allows Toast to insert the corresponding read or write functions and checks for each memory type. Additionally, the compiler understands a set of common memory functions, like memcpy, memset, strcpy, which are replaced with optimized library functions, providing the user-familiar standard library functions with optimized implementations.

Changing the types of pointers may have further implications, e.g., the return type or an argument of a function may change. The Toast compiler tries to infer the necessary changes. The Toast compiler identifies these functions and creates a copy for every ToastPtr type calling the specific function, which is necessary as the libraries contain different read/write calls. This feature also incentivizes code reuse, as the same function can be used for various memory types. The copies of the function are transformed the same way as user-annotated pointer accesses. In cases where the Toast compiler cannot change the code itself, e.g., when the definition of the function or the caller of the function is in a different compilation unit, Toast requires additional function signature annotations. The Toast compiler further registers functions that are annotated as error handlers in the error handler registry of the runtime library.

Toast allows users to define their attribute parameters. Thus, a parameter is not coupled to a specific technology and can be as generic as *Net(work)*. The user supplies the Toast compiler with a configuration file that maps the parameters to technologies. This file provides the compiler with information about which library calls and checks to perform. The compiler can find all code paths where a ToastPtr of a specific type is used by re-compiling the whole code base. Then, it scans the AST to find patterns defined in the configuration and adds the necessary function calls and checks.

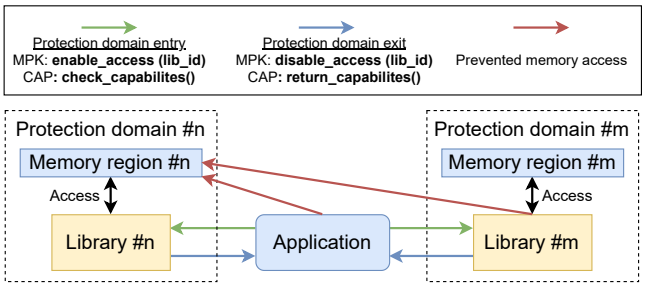

Figure 3: Toast protection mechanism: On a protection domain transition (green and blue arrows), the appropriate capability checks or the enabling/disabling of the access for a protection key are performed. Toast further prevents access to inappropriate protection domains (red arrows).

4.2 Memory Protection

Toast aims to prevent information leaks due to mixups between pointer types and libraries. For this, Toast defines protection domains inside the virtual address space (VAS) of the application compiled with the Toast compiler. We implement two different versions of protection libraries in Toast, shown in Figure 3. The Toast protection library intercepts every call that triggers a protection domain transition, e.g., library calls, and ToastPtr dereferences. Note that the mechanism of the actual transition depends on the chosen configuration. The protection library also introduces appropriate checks to determine the validity of each memory access.

Memory safety model. Toast partitions the memory into memory types, each one associated with its own protection domain. These types are restricted to specific access patterns that can be used to define a region where bound checking can be enforced.

At protection domain transitions, Toast checks pointers for spatial and temporal validity, i.e., the pointer's internal memory type matches the memory type of the pointer's address. Further, the pointer's protection domain should match the protection domain being transitioned into. Toast prevents the dereference of the provided pointer in case these conditions are not fulfilled. A programmer can explicitly transform any pointer with a pointer type cast, thus supporting zero-copy approaches.

Toast does not enforce memory safety within a library, as this requires instrumenting every dereference within the library, which can cause significant overheads. Currently, Toast only provides protection against accesses in non-intended memory regions via its protection mechanisms. However, Toast can increase the safety guarantees through its configuration, such as by enabling the MPK protection library, to enforce safety inside the library as well.

I: Software-based capability storage. Capabilities are an efficient method for resource management and fine-grained access control, suitable for security-critical systems [29, 48, 52, 64, 65, 76, 90, 91]. A capability refers to an object or resource with its access rights. When an application attempts to access a resource (e.g., storage) managed by a capability, the system examines the current capability rights and permits the access or aborts the operation based on them.

In Toast, every pointer in the user code gets transformed into a capability and is represented as a capability object (CapObj). A capability has an *epoch* and an *address*. Toast leverages the fact that both x86 and ARM require the use of a canonical pointer form

Table 3: ToastPtr accesses to library calls to x86 for consistency and persistency specifications.

Toast		Library calls	x86
Config.	**Access**		
SC	W(x)	W(x,SC)	W(x);mfence
	R(x)	R(x,SC)	R(x)
RA	W(x)	W(x,rel)	W(x)
	R(x)	R(x,acq)	R(x)
Relaxed	W(x)	W(x,rlx)	W(x)
	R(x)	R(x,rlx)	R(x)
Default	W(x)	W(x,NA)	W(x)
	R(x)	R(x,NA)	R(x)
Persistent, non-SC (M)	W(x)	W(x,M);clflush(x)	W(x);clflush(x)
Persistent, SC (M)	W(x)	W(x,SC)	W(x), mfence

and stores the epoch in the higher unused bits. Thus, the user code cannot directly dereference a capability, as it is an invalid memory address. Further, the address is still encapsulated in the capability and allows correct pointer arithmetic operations. This is similar to fat pointer approaches where metadata is encoded within the enhanced pointer representation. However, Toast capabilities have the same size as raw pointer on the corresponding platform. Therefore, ToastPtrs do not suffer from some of the disadvantages of common fat pointer designs, such as the increased amount of required memory or the additional cache pressure.

Toast's capability protection mechanism splits the address into indices to a multi-level table with a configurable width, inspired by the multi-level page table design. Each table level includes metadata, i.e. a CapObj, to infer the access rights and check whether the memory region was revoked, or a pointer to the next level. A CapObj contains *(i)* the epoch in which the memory region was created, *(ii)* its access rights, i.e., R/W access, *(iii)* the prefix of the capability to convert it to its canonical form, and *(iv)* a protection domain ID.

For a capability to be valid, its corresponding CapObj must have the same protection domain ID as the capability. The capability's protection domain ID is stored in its type and, therefore does not add extra data to the run time. Furthermore, the epoch of the capability should be equal to the epoch of the CapObj, and the epoch that is stored with every protection domain. This allows Toast to perform fast revocation of memory regions and only requires deleting CapObj from the capability storage in the event of an overflow of the epoch counter, which should happen very rarely. Note that Toast increments the epoch of a protection domain whenever it is completely removed from the current execution, e.g., unloading the corresponding library or re-initializing it.

II: Hardware-assisted protection. MPK [39] is an x86 ISA extension allowing page-level access control. It leverages 4 bits of every page-table entry for a *tag*. The allocation and release of a protection key and the page tagging operation require elevated privileges and, therefore, are performed via system calls. However, a process can change the granted permissions for the pages tagged with a specific key in userspace by updating a special register (PKRU).

Toast assigns each protection domain its memory protection key. Additionally, every protection domain has its unique allocator per application thread. This implies that each library operates in different address ranges. Toast's MPK-based protection library

leverages this region segregation and intercepts the allocation functions (e.g., malloc, realloc, free) and the mmap/munmap operations. In this way, Toast can tag the pages that a library allocates or maps with the appropriate memory protection key.

On a protection domain transition, Toast identifies the protection key of the new protection domain and enables access for the memory regions tagged with this specific key while disabling the rest. Protection key 0 is an exception as it is never disabled. It provides metadata essential for the program's execution. With this approach, access to a memory region belonging to a different protection domain results in a segmentation fault triggered by MPK. Since MPK's access control is thread-local, application threads can legitimately interact with different libraries simultaneously.

Currently, Toast supports up to 15 protection domains per application, equal to the available memory protection keys excluding the default one. This limitation can be lifted using software tools [80].

4.3 Memory Consistency Enforcement

Memory consistency models provide a contract between the programmer and the underlying system. This also applies to the ToastPtr while it accesses shared memory in a concurrent program. Programmers may specify a particular memory consistency model in the configuration file for the ToastPtr accesses. Next, the ToastPtr compiler generates code for the respective platform, following the translations in Table 3. Currently, our configuration file allows the programmer to specify a memory model in sequential consistency (SC), release-acquire (RA), and relaxed (Rlx). Given these models, ToastPtr generates C/C++ atomic access library calls [57, 58] that finally generate respective x86 instructions as shown in Table 3. Note that, Toast does not currently provide a mechanism to ensure memory consistency across heterogeneous memory regions. However, Toast could be extended to support special fences spanning multi-memory regions, enforced by the runtime library.

For SC configuration, a write (W) access by ToastPtr is translated to an atomic write access with SC memory order that in turn generates a write access with a trailing mfence in x86 [2]. An SC read (R) access by ToastPtr is translated to an atomic read with SC order that finally generates a read instruction in x86 [2]. For RA configurations, the release-write and acquire-read operations by ToastPtr result in respective library calls and generate write and read accesses in x86. Similarly, for Rlx configuration, the memory accesses by ToastPtr result in respective memory accesses with memory order relaxed (rlx) and finally generate the respective memory access instructions in x86 [2]. By default, the ToastPtr accesses result in non-atomic (NA) memory accesses that generate write and read instructions in x86 [2]. Following these translation schemes, the generated program follows the x86-TSO model [79].

In addition, if a memory location is marked as persistent, then the persistent non-SC write accesses by ToastPtr generate trailing clflush operations on the same location along with the respective write operations. As an *mfence* is stronger than *clflush* operation, we do not generate *clflush* operations for persistent SC accesses. In this case, the generated x86 program follows the Px86 persistent x86-TSO model [84].

4.4 Crash Consistency and Thread Safety

Crash consistency. Toast preserves the crash consistency properties of the underlying invoked libraries. Toast allows the developers to tailor read and write operations (i.e., *ld/st*) to the device-specific library calls. Thus, they are eligible to use the appropriate APIs, based on the desired crash consistency guarantees, to interact with the storage or even encapsulate storage modifications within transactions, depending on the chosen framework (e.g., SPDK, PMDK). With this approach, Toast can maintain the crash consistency properties, regardless of the granularity of the device, if the suitable device-specific library is chosen.

```
1  void PM_write(toast_ptr_type *ptr, toast_ptr_type new_data) const {
2    TX_BEGIN(pm_pool) { /* pm_pool is set at the init phase of TOAST */
3      /* the actual transaction code... */
4      PM_snapshot_direct(pm_pool, ptr, sizeof(*ptr));
5      *ptr = new_data;
6    } TX_ONABORT {
7      /* executed only if the transaction fails or is aborted by an error */
8      /* __option__ = retry, continue or abort */
9      call_PM_event_handler(__option__, {pm_pool, ptr, sizeof(*ptr)});
10   } TX_END
11 }
```

Listing 4: Error handling and crash consistency in Toast.

For instance, in case of updates in PM that exceed the atomicity boundary of 8 bytes, Toast can wrap PM modifications inside software transactions provided by PMDK [8, 40] through the Toast PM pointer implementation specified in the configuration file, which, in turn, ensures that the updates are performed in a crash-consistent manner. This process is shown in Listing 4. The *PM_write* function is invoked when the Toast application updates a PM object through a PM pointer (*ptr*). First, a PMDK transaction is initiated (Line 2), where the content of the object is snapshotted in the undo log (Line 4) before its actual update (Line 5). This snapshot ensures that in case of a crash, the application can recover the object to a consistent state. In case of a transaction abort (Line 6), Toast calls the registered event handler (Line 8) with one of the retry, continue or abort options (Table 2) accompanied with error-related data. However, note that Toast is generally designed to target memory systems beyond persistence (e.g., NIC, SGX enclave memory).

Thread safety. Toast inherently provides the same thread-safety guarantees with the underlying device-specific libraries. While it does not introduce any additional race conditions, it cannot provide data isolation without the intervention of the developers. It remains a developer's responsibility to ensure that the performed *reads/writes* to a memory region are properly synchronized. Toast does not interfere with the traditional locking mechanisms (e.g., mutex). Thus, the programmers can effortlessly leverage them out of the box when implementing the desired ToastPtr operations, as they do in typical applications. Note that if sophisticated locking mechanisms are required by a library, their logic needs to be embedded into Toast runtime, which will be reflected in the injected code through the Toast compiler. This design choice enhances the developers' flexibility, allows for synchronization optimizations (e.g., by only placing locks wherever they are mandatory), and promotes compatibility with existing applications.

4.5 Toast Runtime Library

The runtime library implements a unified API for different memory types and inserts necessary run-time checks. Implementing the runtime library depends on the configuration of the technologies chosen by the system designer.The API is implemented once for

Table 4: Toast case-studies (§ 5) with memory types and LoC for the original version compared with the Toast version.

	Memory types					LoC		
	NIC	Unprotected	PM	Enclave	DRAM	Original	Toast	Reduction
Secure in-memory KVS		✓		✓		110	105	4.5 %
Replication protocol	✓	✓		✓		893	852	4.6 %
Persistent log	✓		✓		✓	123	120	2.4 %
Persistent KVS	✓		✓		✓	225	182	19.1 %

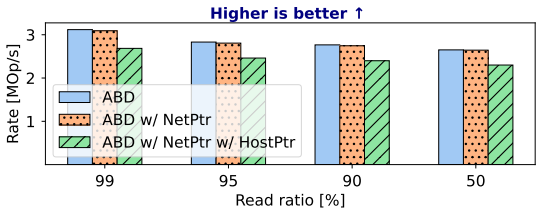

Figure 4: Overhead of ToastPtr on the throughput of the replication protocol with ToastPtr using the YCSB benchmark for different read/write ratios.

each supported technology and can be reused in different projects. To decrease the implementation effort, Toast provides templates for commonly used patterns, which can be combined and extended.

Furthermore, the runtime library contains a map of registered error handlers. The runtime library invokes the error handler with appropriate parameters, handles its return code, and implements the retry and abort functionalities.

By factoring the low-level implementation into a runtime library (instead of implementing it directly in the compiler), we increase the extensibility of Toast, as this makes the addition of a new device technology easier. Additionally, to stay compatible with as many code bases as possible, the Toast runtime library does not use any libraries except the C++ standard library.

5 Application Case-studies

To evaluate Toast, we port four representative applications that use different memory areas (see Table 4). These applications cover typical programming scenarios using heterogeneous memory types and, thus, can highlight Toast's achieved properties.

Secure in-memory KVS. The adoption of trusted hardware resulted in a redesign of secure KVSes [25, 26, 63] to place keys and values in different memory areas to alleviate the memory restrictions of TEEs and improve their performance. We port a secure in-memory KVS [25] that accesses both enclave memory in TEEs and untrusted host memory. The in-memory KVS judiciously partitions the keys (enclave memory) and values (untrusted host memory) using pointer-based data-structures, e.g., skip lists [82]. We replace these pointers with ToastPtrs to manage the memory accesses the data in the untrusted memory, while preventing information leaks.

Replication protocol. Replication is a standard recipe for fault tolerance. To this end, we adapt an implementation of the ABD replication protocol [23, 73], based on the Avocado project [25], to Toast. To provide a secure distributed in-memory KVS, the secure network stack differentiates between the untrusted NIC and trusted enclave memory. It further uses untrusted host memory to store a copy of the requested values, preventing value lifetime inconsistencies and enclave pressure. We port the ABD implementation to use ToastPtr for both the network interface and the untrusted host memory buffers.

Persistent distributed shared log. Shared logs are used to establish the order of operations in distributed systems [27, 59]. The distributed servers are able to read/write entries from/to the log, which is also replicated over multiple nodes, guaranteeing fault tolerance. We port a persistent shared log implementation [60] to ToastPtr. Our log application [4] uses sockets for network communication between the system's nodes and PM as storage. We port both to use ToastPtr.

Persistent KVS. Persistent KVSs are used to store large amounts of data in storage devices (e.g. HDDs, SSDs). Persistent storage technologies present a high overhead compared to in-memory solutions, leading to the emergence of new technologies On top of that, persistent KVSs require fast networking to communicate with clients [8, 10], like PM, and userspace drivers, like SPDK [10]. Like storage technologies, network stacks have shifted to userspace [6, 44], which requires applications to differentiate between pointers to storage, network devices, and normal memory. As our use case, we port a MICA implementation [69] running with eRPC [60]. Here, we adapt the network stack to use Toast.

6 Evaluation

We evaluate Toast across four axes: programmability (§ 6.2), performance (§ 6.3), portability (§ 6.4) and protection (§ 6.5).

6.1 Experimental Setup

Experimental testbed. We perform our experiments on a cluster of 5 machines with Intel(R) Core(TM) i9-9900K CPUs, each with 8 cores (16 HT), 64 GiB memory, 32 KiB (L1D, L1I), 256 KiB (L2), 16 MiB (L3) caches, and Intel Corporation Ethernet Controller XL710 for 40GbE QSFP+ (rev 02) NICs.

We measure the performance of the in-memory KVS from our case study and run the micro-benchmark for the protection libraries on a machine with an Intel(R) Xeon Gold(TM) 5317 CPU, with 12 cores (24 HT), 256 GiB memory, 512 KiB (L1D), 384 KiB (L1I), 15 MiB (L2), 18 MiB (L3) caches, as the i9-9900K of our networking setup does not support MPK.

Methodology and baseline. As Section 5 explains, we port four applications to use Toast. We measure Toast's overhead by comparing the performance of Toast versions to that of unmodified versions. We use the YCSB [16, 38] benchmark for the replication protocol and the in-memory KVS. We perform experiments with various read/write ratios (100%, 99%, 90%, 50%, 0% R) and different value sizes (128 B—2 KiB). For the persistent shared log and persistent KVS, we use the benchmarks provided by the applications.

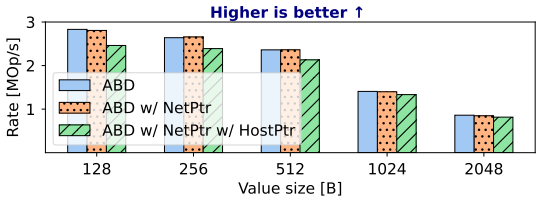

Figure 5: Overhead of ToastPtr on the throughput of the replication protocol with ToastPtr being used for the networking or networking and unprotected memory in the YCSB benchmark for different value sizes.

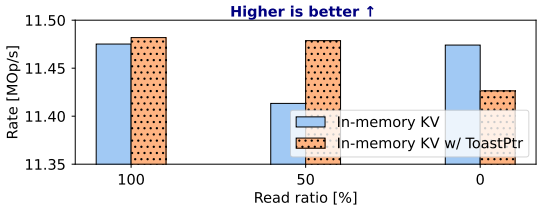

Figure 6: Overhead of ToastPtr on persistent KVS compared to hand optimized version for different read ratios.

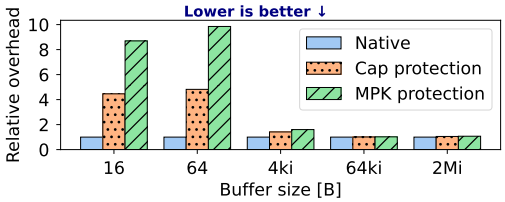

Figure 7: Runtime overhead of Toast protection mechanisms w.r.t. to the native execution for protection domain transitions performing a *memcpy* with various buffer sizes.

6.2 Programmability

Q1: *How easy is designing applications using the ToastPtr abstraction?* To answer this question, we count the lines of code (LoC) modified in Toast compared to the original hand-written version for each Toast use case (see Table 4).

The Toast version reduces the number of LoC in every application by 2.4 %-19.1 %. Further, Toast simplifies or eliminates complicated function calls for buffer resizing or message enqueueing.

> **Q1 takeaway:** Besides reducing LoC, Toast also improves programmability by allowing the programmer to manage DMA-capable devices used in an application with the same interface, instead of having to learn and employ device-specific APIs.

6.3 Performance

Q2: *What is the overhead of using Toast compared to manually optimized code?* To answer this question, we compare Toast versions of our four applications to the hand-optimized (unmodified original) versions by measuring their throughput.

Secure in-memory KVS. We run the YCSB benchmark with 400 MOps over 10 M distinct keys following a uniform key distribution with different read-write ratios.

In a read heavy workload (99 % reads), the overhead introduced by ToastPtr is 2.8 % which increases to 11.9 % for write heavy workloads of 50 % writes and reads. This overhead is mainly due to Toast not caching decrypted data in the trusted memory but repeatedly decrypting it on every access. As a write can generate two accesses to the same buffer, this effect is more noticeable in write-intensive workloads. Toast's overhead could be reduced by creating temporary objects with lifetimes greater than individual operations. Precisely, Toast can be extended to allow for caching of values to optimize data transfers. However, the current prototype does not use such a technique, as this optimization would affect the synchronization semantics among threads, which mandates careful consideration.

Replication protocol. We compare the performance of the hand-optimized ABD replication protocol to two Toast counterparts. One counterpart uses ToastPtr to access the NIC to perform network communication, while the other uses ToastPtr to access unprotected memory for its internal KVS. We run the YCSB benchmark with different read/write ratios (Fig. 4) and different value sizes (Fig. 5). We run the protocol on all five servers. The benchmarks were configured with 1.2 GOp over 2.5 M distinct keys following a uniform key distribution. We measure the overhead of ToastPtr on the performance of ABD for the read ratios of 99 %, 95 %, 90 %, and 50 % with a value size of 128 B. The overhead of Toast for the networking library is 0.84 % for a read ratio of 99 %, which shrinks down to 0.23 % as the write ratio increases to 50 %. Using ToastPtr also for the unprotected memory increases the overhead to 13.2 % and 13.8 % for read ratios of 50 % and 99 %, respectively. Like the secure in-memory KVS, the increased overhead of ToastPtr for unprotected memory is due to the caching of data in protected memory in the hand-optimized version of the ABD protocol.

The overhead of the NIC-only use of ToastPtr is generally not affected by the value size and is stable between 0.6 and 0.8 % until the value size exceeds the MTU size (1500 B) of the network packets, e.g., at value size 2 KiB, the overhead is 1.5 %. Exceeding the MTU size requires making an additional copy of each value (in eRPC) to split into multiple packets.

Shared log. We run the shared log application with 1 server thread and 2 clients, each having 8 threads, the largest configuration the benchmark allowed. The entry size ranges from 64 B to 2 KiB. Figure 8 shows the throughput of both the original and the Toast version. Toast performs on par with the original version for all log entry sizes, with a mean performance difference of around 1.8 %.

Persistent KVS. We measure the throughput of the persistent KVS using a server application with 16 threads and 4 clients each with 16 threads to generate the workload. We used a uniform key distribution and read-ratio of 0 %, 50 %, 100 %. Figure 6 shows that the original and Toast versions have similar performance.

> **Q2 takeaway:** Toast introduces negligible performance overhead on the ported applications (< 2.8 % relative to the original version). However, since Toast provides a generic programming model without specialized optimizations (e.g., selective data caching), higher overheads can be observed in some cases.

6.4 Portability

Q3: *How easy is it to switch underlying technologies with Toast?* To evaluate this, we present two case studies of the network and storage libraries.

Network library. We highlight the portability of the network stack from traditional sockets to eRPC [60] for the persistent shared log.

The Toast version requires changing 71 LoC, while porting the original code requires changes in 141 LoC. This 50 % decrease occurs due to the different implementations of asynchronous calls between the versions. Toast's 71 LoC can be reduced further by introducing a unified asynchronous call interface in a future Toast version.

Storage library. We port the same persistent shared log library from using memory-mapped files to using a PM library (PMDK [8]). The Toast version requires changes to 19 lines, all in the initialization phase. The hand-written port requires the same changes and an additional 20 LoC in the storage backend logic, including changing specialized memcpy and synchronization methods.

Q3 takeaway: Toast significantly simplifies the porting process of an application to use a different underlying technology. Our experiments show that Toast can reduce the number of modified LoC by up to 50 %.

6.5 Protection

Q4: *What are the implications and trade-offs, in terms of performance and safety, of Toast's protection mechanisms?* To provide an answer, we evaluate the performance overheads introduced by the capability- and Intel MPK-based protection mechanisms (§ 4.2). First, we design a microbenchmark that repeatedly performs calls to a wrapped *memcpy* function through a linked library call, thus performing protection domain transitions. Our microbenchmark uses two protection domains. Each *memcpy* function call operates on memory regions accessible from the protection domain of its call site. Additionally, we apply the protection mechanisms to the Toast skip list and shared log. Note that, like the MPK-based version, the capability-based version is configured to provide protection guarantees at a page-size granularity in our experiments.

Microbenchmark. We configure our microbenchmark to copy 20 GB of data between protection domains. We vary the copied buffer size to highlight the cost of the domain transitions. Experiments with smaller buffer sizes require more *memcpy* operations and, consequently, more transitions to the protection domain of the linked library. We measure the total time required till all the data has been copied. The presented results indicate the mean of 100 runs for each configuration.

Figure 7 illustrates the relative slowdown of the capability- and MPK-based protection libraries compared to the native execution of our microbenchmark. For small buffer sizes (16 B and 64 B), the capability protection mechanism is 4.46-4.81× slower than the baseline. The respective values for the MPK version are 8.69-9.85×. This large slowdown is caused by the frequent, short-running transitions between the protection domains, which, in turn, result in more checks and pointer cleanups in the capability version and more costly updates of the PKRU that can lead to pipeline stalls in the MPK version. However, as the buffer size increases, the Toast protection mechanisms induce lower overheads. When copies are performed at the granularity of a page (4 kB), the overheads are 41 % and 60 %, for the capability- and the MPK-based approach, respectively. Lastly, we observe that for even larger buffers (64 kB and 2 MB), Toast's protection libraries incur only 1–7 % slowdown since the domain transition overhead is dominated by the longer *memcpy* operations.

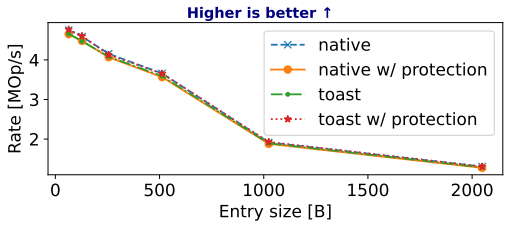

Figure 8: Throughput of shared log for different log entry sizes with and without protection library enabled.

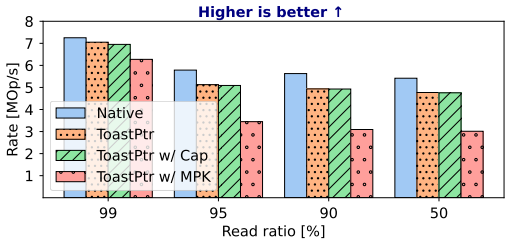

Figure 9: Overhead of different protection library implementation for the in-memory KVS for various read ratios.

Shared log. We run the shared log application with a setup identical to that of § 6.3. We divide the shared log application into networking and storage protection domains, with the networking buffer having to pass through the protection domain switch. To evaluate the overhead of the protection domain switch, we run the benchmark in four configurations: the original shared log application, the same application without ToastPtr but with the protection library, the application with ToastPtr but without the protection library, and a configuration using both ToastPtr and protection library. We execute the experiments as described in § 6.1.

Figure 8 shows the overhead of the capability-based protection library in the shared log application. The capability protection library adds 2.2 to 2.5 % overhead to the native solution. The capability protection library version even performs slightly better than the unprotected Toast version having 1.5 to 2.5 % higher average throughput. The low overhead of the capability version in this application is expected, as most memory is allocated by the user code and then supplied to libraries.

Secure in-memory KVS. We run the YCSB benchmark with 400 MOps sampled uniformly from 10 M distinct keys. Figure 9 shows the overhead of the different protection libraries for different read-write ratios, with a key size of 8 B and value size of 128 B. The capability version has an overhead of 1.5 % for the 99 % read workload. With higher write ratios, the overhead shrinks to 0.3 %. The difference in the overhead of the capability version in read-heavy workloads compared to write-heavy workloads is mainly due to read operations having to perform a full capability storage lookup as the library provides the read buffer and, therefore, needs to be transformed into a capability. However, write operations can assume a fast path as the write buffer is allocated in the user code, and the user explicitly provides the buffer to the library. This does not require a costly lookup and rewriting of the pointer. The MPK-based protection library version incurs a slowdown of 11.0 – 37.3 % for the various workloads. We observe that the overhead decreases as the read ratio increases. This is expected as the fewer put operations

imply less frequent memory allocations and a smaller application memory footprint, leading to fewer page tagging operations.

Q4 takeaway: Toast allows developers to choose which protection mechanism suits their application better depending on the memory access patterns and the desired memory-safety granularity. The overheads of the mechanisms will vary for each case but remain reasonable.

7 Related Work

OS memory management. The OS provides drivers to communicate with devices [7, 12] on the kernel side and sockets/file descriptors on the userspace side. However, modern systems prefer userspace libraries to directly communicate with the device and manage heterogeneous memory areas for improved performance, as in DPDK [6], RDMA [44, 50, 61] and eRPC [60] for remote calls, SPDK [10] for SSDs, or PMDK [8] for PM. However, there is no unifying abstraction across these libraries.

Unified API efforts like oneapi [13, 22], memif [70], EXOCHI [89] and SYCL [42, 49] focus on specific device classes, e.g., GPUs/FPGAs, and introduce one API per use-case. While our idea of unifying APIs is similar, Toast goes further by lifting the communication completely into the compiler and removing special API calls, while also considering the safety aspect.

Additionally, Toast strives to be generic (i.e., handle every type of device that can be mapped to memory). Further, if Toast is implemented on the LLVM IR level, it can be language agnostic and be used in various toolchains. On the other hand, systems such as SYCL [42, 49] or EXOCHI [89] aim to provide a programming framework for heterogeneous accelerators and are strongly binded to C/C++ applications.

On top of that, Toast complements CXL [37] in terms of *(i)* memory types and *(ii)* access properties. Firstly, Toast targets a broader range of memory types that are beyond supported by CXL devices, including, but not limited to, secure enclave memory regions, NIC memory or SPDK buffers. Secondly, CXL devices do not provide access properties such as a protection mechanism.

Memory Consistency and persistency. Memory consistency models are widely studied for C/C++ programming languages [28, 34, 62, 66], compilers [32, 33, 75], and architectures [20, 21, 79, 83]. Based on these models, compiler transformations and mappings to the architectures are proven correct for different models including x86 [28, 81, 85]. These results have resulted in the mapping schemes in [2] which we follow in Toast's approach. More recently, persistency properties in the Intel x86 architecture are explored and the properties of the persistent accesses are formalized in Px86 [84] which we follow in the Toast compiler.

Compiler-based memory management. Compiler-based approaches are used in shared memory systems to optimize memory accesses, e.g., UPC [35], OpenMP [14], HPF [19], OpenCL SVM [41]. In particular, OpenCL SVM enables the host and device portions of an OpenCL application to seamlessly share pointers and complex pointer-containing data structures. However, SVM is strictly restricted to the OpenCL programming model.

Other research has looked into using DMA support in the compiler for heterogeneous compute units [45], static analysis [24], compiler-based approaches [47, 74], secure memory management [71, 78], or even programming language Verona [15]. However, none of these approaches deals with different memory layouts and access patterns based on heterogeneous memory types.

Software-based protection and isolation. Software capabilities have been studied for intra-process memory isolation by introducing capabilities to memory areas and system calls to either threads [31, 72] or objects that can be held by a thread [55]. Two other common techniques to provide isolation are Software Fault Isolation (SFI) [88] and Control-Flow Integrity (CFI) [18]. Other software-based approaches rely on sandboxing to prevent illegal accesses, which cannot be proven correct statically [46, 51, 94].

Toast does not provide strict memory isolation between different components. Instead, it aims at preventing erroneous sharing of sensitive data. It limits code injection to protection domain transitions and does not require every access to be secured, reducing the amount of injected code and performance overhead, while also being easier to integrate.

Hardware-based protection and isolation. CHERI [91], IBM System 38 [30, 54], M-Machine [43] and ARM MTE [1] are examples of hardware support for fat pointers. Other hardware approaches are Page Groups, e.g. HP PA-RISC [17], Intel MPK [11] and ARM Domains [3] that tag memory areas, and Mondrian Memory Protection [92] that separates access rights from translation metadata. These approaches can force access to specific memory regions to go through designated access control gates, thus preventing erroneous accesses. Another hardware-based approach is capability storage systems, such as Intel iAPX 432 [93] and CODOM [87]. This is different from fat pointer schemes, as metadata is stored in multi-level tables. In contrast to these approaches, Toast capabilities do not require hardware support. Further, Toast protection aims to unify the access APIs of different kinds of memory, while maintaining easy-to-use for the programmer.

8 Conclusion

We present Toast, a compiler-based abstraction for heterogeneous memory management. Toast builds on the observation that although accesses to heterogeneous memory require different libraries with vastly different interfaces, all interfaces essentially perform the same basic task of loading data from or storing data in a memory region. Toast makes this uniform for the programmer by introducing the abstractions of memory types and the pointer type ToastPtr, that work with familiar load and store operations. Further, Toast provides programmable error handling callbacks and memory consistency enforcement mechanisms as part of its programming model. Lastly, Toast offers a selection of protection libraries to prevent accidental memory handling errors by developers. Our evaluation based on four applications, which use heterogeneous memory types, shows that Toast improves programmability, offers memory safety, and eases portability to new libraries/memory types, with low to moderate overhead relative to hand-optimized code.

Software availability. Toast is publicly available along with its entire setup (https://github.com/TUM-DSE/Toast).

Acknowledgments

We thank our shepherd and the anonymous reviewers for their helpful comments. This work was partially supported by a Schwerpunktprogramm (SPP) (ID: 2377) from Deutsche Forschungsgemeinschaft (DFG) and an ERC Starting Grant (ID: 101077577).

References

[1] [n. d.]. Armv 8.5 - A: Memory Tagging Extension. https://documentation-service.arm.com/static/624ea580caabfd7b3c13e23f?token=. Last accessed: Oct 2020.
[2] [n. d.]. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.
[3] [n. d.]. Domains. https://developer.arm.com/documentation/ddi0211/k/memory-management-unit/memory-access-control/domains. Last accessed: Oct 2020.
[4] [n. d.]. eRPC: a log store application. https://github.com/erpc-io/eRPC/tree/master/apps/log. https://github.com/erpc-io/eRPC/tree/master/apps/log Last accessed: Jan, 2022.
[5] [n. d.]. eRPC-Raft. https://github.com/erpc-io/eRPC/tree/master/apps/smr.
[6] [n. d.]. Intel DPDK. http://dpdk.org/.
[7] [n. d.]. Intel Network Adapter Driver for PCIe* 40 Gigabit Ethernet Network Connections under Linux*. https://www.intel.com/content/www/us/en/download/18026/intel-network-adapter-driver-for-pcie-40-gigabit-ethernet-network-connections-under-linux.html. Last accessed: Jan, 2022.
[8] [n. d.]. Intel Persistent Memory Development Kit (PMDK). https://pmem.io/pmdk/.
[9] [n. d.]. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.
[10] [n. d.]. Intel Storage Performance Development Kit. http://www.spdk.io.
[11] [n. d.]. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html. Last accessed: Oct 2020.
[12] [n. d.]. Linux* Base Driver for Intel Gigabit Ethernet Network Connections. https://www.intel.com/content/www/us/en/support/articles/000005480/ethernet-products.html. Last accessed: Jan, 2022.
[13] [n. d.]. oneAPI. https://www.oneapi.com/. Last accessed: Dec, 2021.
[14] [n. d.]. OpenMP: The OpenMP API specification for parallel programming. https://www.openmp.org/. Last accessed: Dec 2021.
[15] [n. d.]. Project Verona: Research programming language for concurrent ownership. https://microsoft.github.io/verona/. Last accessed: Oct, 2022.
[16] [n. d.]. YCSB. https://github.com/brianfrankcooper/YCSB.
[17] 1990. *PA-RISC 1.1 Architecture and Instruction Set: Reference Manual.* Hewlett Packard. https://books.google.de/books?id=UahBuAAACAAJ
[18] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. (2009).
[19] V. Adve, Guohua Jin, J. Mellor-Crummey, and Qing Yi. 1998. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing.*
[20] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 8 (2021), 54 pages. https://doi.org/10.1145/3458926
[21] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752
[22] J.Gold Associates. [n. d.]. oneAPI: Software Abstraction for a Heterogeneous Computing World. https://jgoldassociates.com/White_Papers/OneAPI_Whitepaper.pdf.
[23] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-passing Systems. *J. ACM* (1995).
[24] Oren Avissar, Rajeev Barua, and Dave Stewart. 2001. Heterogeneous Memory Management for Embedded Systems. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASE'01).*
[25] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *2021 USENIX Annual Technical Conference (ATC'21).*
[26] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *17th USENIX Conference on File and Storage Technologies (FAST).*
[27] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A Distributed Shared Log. (2013).
[28] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL'11.* ACM, 55–66. https://doi.org/10.1145/1926385.1926394
[29] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09).* Association for Computing Machinery, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579
[30] Viktors Berstis. 1980. In *Proceedings of the 7th Annual Symposium on Computer Architecture (ISCA '80).* http://doi.acm.org/10.1145/800053.801932
[31] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. (2008).
[32] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating optimizations of concurrent C/C++ programs. In *CGO'16.* ACM, 216–226. https://doi.org/10.1145/2854038.2854051
[33] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *CGO '17.* IEEE, 100–110.
[34] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. 3, POPL (2019). https://doi.org/10.1145/3290383
[35] Wei-Yu Chen, C. Iancu, and K. Yelick. 2005. Communication optimizations for fine-grained UPC applications. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05).*
[36] Clang: a C language family frontend for LLVM [n. d.]. Clang: a C language family frontend for LLVM. https://clang.llvm.org/. https://clang.llvm.org/ Last accessed: Jan, 2021.
[37] CXL™ Consortium. August 29, 2024. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/.
[38] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC).*
[39] LWN-Jonathan Corbet. [n. d.]. Memory Protection Keys. https://lwn.net/Articles/643797/. https://lwn.net/Articles/643797/ Last accessed: Oct, 2022.
[40] Intel Corporation. [n. d.]. C++ Transactions for Persistent Memory Programming. https://www.intel.com/content/www/us/en/developer/articles/technical/c-plus-plus-transactions-for-persistent-memory-programming.html.
[41] Intel Corporation. [n. d.]. OpenCL™ 2.0 Shared Virtual Memory Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/opencl-20-shared-virtual-memory-overview.html.
[42] Intel Corporation. [n. d.]. What is SYCL? Quick Guide to SYCL Implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/quick-guide-to-sycl-implementations.html#gs.56e52p.
[43] William J Dally, Stephen W Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S Lee. 1994. *M-Machine architecture v1.* Technical Report. 0. Technical Report–MIT Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology.
[44] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*
[45] Alexandre Eichenberger, K. O'Brien, Peng Wu, Tong Chen, P. Oden, Dan Prener, J.C. Shepherd, Byoungro So, Zehra Sura, A. Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. 2005. Optimizing Compiler for the CELL Processor. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05).*
[46] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06).*
[47] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21* (Virtual, USA).
[48] Google. [n. d.]. Fuchsia. https://fuchsia.dev/. https://fuchsia.dev/ Last accessed: Oct, 2022.
[49] The Khronos® Group. [n. d.]. SYCL Overview. https://www.khronos.org/sycl/.
[50] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16).* 202–215.
[51] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.* (2017).
[52] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. 2019. SemperOS: A Distributed Capability System. In *2019 USENIX Annual Technical Conference (USENIX ATC 19).* USENIX Association, Renton, WA, 709–722. https://www.usenix.org/conference/atc19/presentation/hille
[53] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. 2018. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*
[54] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. 1981. IBM System/38 Support for Capability-Based Addressing. In *ISCA.*

[55] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[56] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur-Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. 2012. High-Performance Design of HBase with RDMA over InfiniBand. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*.

[57] ISO/IEC 14882. 2011. Programming Language C++.

[58] ISO/IEC 9899. 2011. Programming Language C.

[59] Minwen Ji, Alistair Veitch, and John Wilkes. 2003. Seneca: Remote Mirroring Done Write. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*.

[60] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[61] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[62] Jeehoon Kang, Hur, Chung-Kil, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL '17*. ACM.

[63] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-Memory Key-Value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)*.

[64] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. SeL4: Formal Verification of an Operating-System Kernel. *Commun. ACM* 53, 6 (jun 2010), 107–115. https://doi.org/10.1145/1743546.1743574

[65] Genode Labs. [n. d.]. Genode. https://genode.org/. https://genode.org/ Last accessed: Oct, 2022.

[66] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI 2017*. 618–632. https://doi.org/10.1145/3062341.3062352

[67] R.P. LaRowe, C.S. Ellis, and M.A. Holliday. 1992. Evaluation of NUMA memory management through modeling and measurements. *IEEE Transactions on Parallel and Distributed Systems*.

[68] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. San Jose, CA, USA, 75–88.

[69] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[70] Felix Xiaozhu Lin and Xu Liu. 2016. Memif: Towards Programming Heterogeneous Memory Asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*.

[71] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*.

[72] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[73] N. A. Lynch and A. A. Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing (FtCS)*.

[74] Deepak Majeti, Rajkishore Barik, Jisheng Zhao, Max Grossman, Vivek Sarkar, Dieter Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L. Scott, and Josef Weidendorfer. 2014. Compiler-Driven Data Layout Transformation for Heterogeneous Platforms. In *Euro-Par 2013: Parallel Processing Workshops*. Springer Berlin Heidelberg.

[75] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI'13*. ACM, 187–196. https://doi.org/10.1145/2491956.2491967

[76] R. M. Needham and R. D.H. Walker. 1977. The Cambridge CAP Computer and Its Protection System. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles* (West Lafayette, Indiana, USA) *(SOSP '77)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/800214.806541

[77] NVIDIA. [n. d.]. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.

[78] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. 2019. CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*.

[79] Scott Owens. 2010. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *ECOOP*. 478–503.

[80] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

[81] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.* 3, POPL (2019). https://doi.org/10.1145/3290382

[82] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communication of ACM (CACM)* (1990).

[83] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[84] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (dec 2019), 31 pages. https://doi.org/10.1145/3371079

[85] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *PLDI'12*. ACM, 311–322. https://doi.org/10.1145/2254064.2254102

[86] sdk, sgx, intel [n. d.]. Intel Software Guard Extensions SDK for Linux. https://01.org/intel-softwareguard-extensions. https://01.org/intel-softwareguard-extensions

[87] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with Code-centric memory Domains. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*.

[88] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. (1993).

[89] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. 2007. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 156–166. https://doi.org/10.1145/1250734.1250753

[90] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2012. A Taste of Capsicum: Practical Capabilities for UNIX. *Commun. ACM* 55, 3 (mar 2012), 97–104. https://doi.org/10.1145/2093548.2093572

[91] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*.

[92] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*.

[93] Ian H. Witten and John G. Cleary. 1983. An introduction to the architecture of the Intel iAPX 432. *Software & Microsystems* (1983).

[94] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*.