Technical Report

UCAM-CL-TR-858
ISSN 1476-2986

Number 858

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

CHERI: A RISC capability machine for practical memory safety

Jonathan D. Woodruff

July 2014

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted March 2014 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

This work presents CHERI, a practical extension of the 64-bit MIPS instruction set to support capabilities for fine-grained memory protection. Traditional paged memory protection has proved inadequate in the face of escalating security threats and proposed solutions include fine-grained protection tables (Mondrian Memory Protection) and hardware fat-pointer protection (Hardbound). These have emphasised transparent protection for C executables but have lacked flexibility and practicality. Intel's recent memory protection extensions (iMPX) attempt to adopt some of these ideas and are flexible and optional but lack the strict correctness of these proposals. Capability addressing has been the classical solution to efficient and strong memory protection but it has been thought to be incompatible with common instruction sets and also with modern program structure which uses a flat memory space with global pointers. CHERI is a fusion of capabilities with a paged flat memory producing a program-managed fat pointer capability model. This protection mechanism scales from application sandboxing to efficient byte-level memory safety with per-pointer permissions. I present an extension to the 64-bit MIPS architecture on FPGA that runs standard FreeBSD and supports self-segmenting applications in user space. Unlike other recent proposals, the CHERI implementation is open-source and of sufficient quality to support software development as well as community extension of this work. I compare with published memory safety mechanisms and demonstrate competitive performance while providing assurance and greater flexibility with simpler hardware requirements.

Acknowledgements

I would like to thank:

- Simon W. Moore, my supervisor, who has supported this work with not only close and regular consultation but also tedious engineering effort on multiple occasions.
- Robert N. M. Watson who provided the initial concept and motivation and patiently debugged processor and peripheral problems to bring up the FreeBSD operating system using tools not fit for the task.
- Michael Roe who helped shape a sound instruction set model and supplied tests both routine and arcane.
- David Chisnall who provided compiler insight for the instruction set and has been very prompt with compiler extensions and corrections to support benchmarking.
- Robert Norton, Alan Mujumdar, Colin Rothwell, Alex Horsman, and Alexandre Joannou, fellow students who have suffered through extensions of my design to prove (and improve) its extensibility and usefulness to a wider community.
- Nirav Dave who introduced me to more subtle aspects of Bluespec.
- Gregory Chadwick for the initial implementation of the 64-bit MIPS.
- Peter G. Neumann who provided, among other things, historical perspective to this work.
- Ben Laurie for insightful encouragement from an industry perspective.
- Brooks Davis, Stacey Son, Steven Murdoch, Jonathan Anderson, Philip Paeps, Hassan Saidi, and the rest of the CTSRD team.
- My wife Mary who patiently endured lonely deadlines with our sweet and beautiful son Daniel.

Contents

Chapter 1

Introduction

Computer security depends on memory safety, and full memory safety is prohibitively expensive in current computer systems. Computer system security is becoming increasingly important as we entrust more to computer systems. Computer system security depends both on safe system design and on a sound execution of that design to ensure that a program can do only what it was designed to do. This requirement of sound execution for system security requires memory safety, that is, that a program can only access the memory addresses that it is intended to access. Current processor support for memory safety through paged memory is strong and has negligible overhead in the common case. This has spurred its widespread adoption in computer systems. However, paged memory is coarse-grained and can only guarantee that an address used by a program is allocated to that program but cannot guarantee that the address belongs to the referenced object, or is even associated with the current subroutine or library. While this model is useful for strongly distinguishing between protected system memory and program memory to provide program isolation and system stability in the face of untrustworthy programs, it is insufficient to ensure that programs behave according to their design, leaving the door open for a multitude of security exploits, including buffer overflows on the stack and heap to corrupt data and inject code. These shortcomings of traditional paged memory protection have been mitigated in many cases using managed languages where performance can be traded for correctness. However, the most commonly used programs cannot trade performance for correctness. These programs include the web browser, the operating system, and the managed language runtime itself.

1.1 Existing proposals for fine-grained memory safety

Two classes of solution have been proposed to solve the high-performance memory safety problem. Firstly developers may use a modified compiler to enforce some level of memory safety without changing the high performance programming model. These solutions include LLVM AddressSanitizer [73] and the CCured compiler [58]. This approach can help with debugging but is unlikely to be used at runtime without significant overheads comparable to managed-language programs[1]. Memory safety for C programs often has a higher cost than the same safety for a managed language because transparent compatibility with the C conventions must be maintained, which forces out-of-band data structures and

[1]Serebryany *et al.* found AddressSanitizer to have an average overhead of 73% with an average memory overhead of 337%.

communication. Managed languages are natively designed for memory safety and can therefore account for safety natively in binary interfaces.

The second class of solutions use hardware extensions to provide fine-grained memory protection with reduced software overhead. These proposed solutions, including Mondrian Memory Protection [91], Hardbound [28], and Intel's iMPX [43], have attempted to be transparent at the cost of flexibility, performance, and/or correctness. Hardbound demonstrated that it was possible to accelerate a bounded-pointer pointer model in hardware without affecting existing pointer layout and values, but required complex and mandatory operations on every pointer load and store to maintain the integrity of the model. Intel, a key supporter of the Hardbound research, adopted some of its key ideas in their iMPX extensions but made most of the operations both manual and optional. Making the operations manual in the ISA allowed them to fit cleanly into a largely RISC implementation. Making the operations optional allowed incremental adoption of the protection features so that a program could protect some structures, but not all. However, the manual and optional implementation of the Hardbound model loses crucial memory safety guarantees, limiting iMPX to a debugging facility rather than a tool for writing provably safe programs. Commercial implementations need a practical approach that does not depend on unrealistic mechanisms for strong correctness. Furthermore, the cost of the Hardbound model scales with the number of pointers and not with the number of protection regions, so a coarse-grained sandbox model in which all pointers are to the same region will have nearly the same overhead as a fine-grained use case with separate regions for each object.

Mondrian memory protection proposed a protection table similar to a TLB but which conceptually has word-level granularity. The Mondrian model has somewhat been adopted by memory protection units (MPUs) for embedded processors that do not need a TLB but which benefit from memory protection. The most complex feature of the Mondrian proposal was a hardware walker for the byte granularity table, and this has not been implemented commercially. Without the table walker, commercial MPUs are optimised to protect a few regions of address space (ARM MPUs can have up to 16 entries) and will sandbox a program with no overhead but will incur frequent table fills when protecting individual program objects. The classic solution to efficient and strong memory safety in hardware is capability-based addressing, which was developed in the 1960s and 1970s [27, 90, 40]. Capability-based addressing is an approach that accesses memory through unforgeable references. Classical capability addressing machines used complex microcode to implement capability addressing with references being similar to hardware supported file descriptors [71, 90]. These machines used highly complex instruction sets to manage protection but were not the practical compiler targets that the later RISC instruction sets proved to be. In 1994, the M-Machine proposal from MIT showed that capability addressing could be done in a RISC instruction set with a high-performance pointer model, and within a flat address space [13]. While the M-Machine was extremely efficient for the protection it provided, it required a total conversion of all software on the system to the capability addressing model. Thus it did not demonstrate that a capability mechanism could be used as a target for compilers of common languages due to its use of a proprietary instruction set and lack of availability to the research community. This shortage strengthened the community's impression that capability machines did not fit common programming models [91].

1.2 Contributions

I demonstrate that it is possible to implement a capability model:

- Within an existing RISC ISA without disturbing unmodified executables
- Which has an obvious and efficient RISC hardware implementation
- Whose performance scales elegantly from array safety to sandbox enforcement
- As a practical compiler target
- Which allows incremental adoption of protection by legacy programs
- Which maintains safety properties regardless of the model used

To demonstrate these contributions I have designed and built the CHERI processor[2] which introduces mandatory capability addressing to MIPS without disturbing existing operating systems and programs [88, 94]. CHERI uses a capability addressing model which allows programs to create unforgeable references to their address space and use them in place of pointers. The CHERI model is scalable, enabling ubiquitous use of capabilities to guard every reference in a program or allowing simple sandboxing of all references to a portion of address space. CHERI has been implemented as a high quality opensource FPGA design to provide a community research platform. Previous hardware proposals were either implemented in proprietary silicon, such as the M-Machine, or in a hardware simulator and none of these implementations became the basis for comparative research. CHERI is not a fragile research prototype but runs the full FreeBSD operating system and a large library of applications with reasonable performance.

1.3 Publications

Peer reviewed publications —

Jonathan Woodruff, Simon W Moore, Robert N M Watson, *Memory Segmentation to Support Secure Applications*, Engineering Secure Software and Systems Doctoral Symposium, February 2013

(Contributes to Chapter 3)

Jonathan Woodruff, Robert N M Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, Michael Roe *The CHERI capability model: Revisiting RISC in an age of risk*, ISCA 2014, June 2014

(Contributes to Chapter 4, Chapter 5, and Chapter 7)

Invited paper —

A Theodore Markettos, **Jonathan Woodruff**, Robert N M Watson, Bjoern A Zeeb, Brooks Davis, Simon W Moore, *The BERIpad tablet: open-source construction, CPU, OS and applications*, Asia-Pacific Workshop on FPGA Applications, November 2013

(Contributes to Chapter 5)

[2]I was the hardware lead on the CHERI project, designing the instruction set for integration with MIPS and building the implementation. Robert Watson and David Chisnall provided key consultation from the operating system and compiler sides respectively, and Michael Roe consulted for formal correctness. Ben Laurie also consulted from an industry perspective.

1.4 Dissertation overview

I will begin in Chapter 2 with discussing historical capability machines in light of their contributions to the CHERI architecture. Chapter 3 will describe CHERI, the capability extension of the 64-bit MIPS instruction set. In Chapter 4 I will present a conceptual comparison and limit study for our approach to discuss the advantages of our RISC capability model over competing models and to demonstrate that our RISC capability approach is performance competitive with other hardware proposals for memory safety. In Chapter 5 I describe the CHERI implementation as a mature, stable, and accessible open source design to allow for reproduction and extension of this work. In Chapter 6, I will examine the software model for using the capability ISA as well as the security benefits that are gained by straightforward use of the extensions for memory safety. In Chapter 7, I will present benchmarks using memory protection under FreeBSD on CHERI showing performance overhead characteristics for capability memory protection.

Chapter 2

History of hardware support for memory protection

The CHERI RISC capability segment model presented in this dissertation follows from memory segmentation in early processors [27], especially as it developed to support capability systems. This work also shares the modern context with a variety of memory protection strategies, both software only and with hardware support. The first section of this chapter will detail the historical foundations of our current work and the second section will present other modern approaches to strong memory protection.

2.1 History of memory segmentation

In the 1960s and 1970s the expanding size of computer programs made obvious the need for memory protection and virtualisation [25, 70]. Memory protection was needed to safely share data structures between routines that might not be correct, especially during development. Virtualisation allowed the program to be much larger than the core memory and also allowed convenient sharing of routines and data between programs without relocation.

Programmers of the time identified memory segmentation as a direct solution for their requirements. Segmented memory was not addressed as a single space but as a list of objects which could be independently protected and whose location in physical memory could be abstracted from the programmer [27]. An address in such a system is composed of both a segment number and a word number. Each segment had a physical location in memory and might have been resident only in second level storage waiting for a reference before being relocated into core memory.

Paged segmentation — Pure memory protection by segmentation was directly implemented by several systems, including the Burroughs B5000 and derivatives and eventually the Intel 286. However, a simpler approach, *paging*, implemented on the Manchester Atlas [48], was proposed concurrently with segmentation. A paged memory used regular sized memory blocks for virtualisation and, to some extent, protection of memory. Paging was very practical for a general memory virtualisation system as it separated the specifics of program structure from the problem of core memory allocation. Paging allowed memory blocks to be freely swapped in and out of core memory without fragmentation which required complex placement routines. However segmentation was still seen as necessary within the larger virtual "single-level store" enabled by paged memory for protection of shared structures. Dennis gave four justifications for the need for segmentation on top of a paging system, as used in Multics on the GE-645 [26]:

1. A computation should have unique names for all objects and should not need to relocate objects for lack of namespace.
2. Data objects should be arbitrarily expandable without relocation.
3. Shared objects should also have a shared name.
4. A protection mechanism should operate in namespace to permit access to information by a computation only in an authorised manner

These four arguments justified the design of the virtual memory system of the GE-645 which ran the Multics operating system [9, 64]. However, modern computers have vast, paged virtual address space and support multiple address spaces for disparate domains of protection. Today's computers also have permissions on TLB entries to control access on a page granularity. As a result, item 1 from Dennis' list is no longer applicable because there is no shortage of namespace. Item 2 has mostly been worked around by high-level languages taking advantage of plentiful memory and excess speed by allocating new objects for each node of an expandable structure with new objects not necessarily contiguous in namespace. More performance-sensitive code would use static allocation when possible. Item 3 does not hold between address spaces, which are the modern domains of protection, as each address space has its own name for a shared object. However the common case of a shared objects in a single address space of course allows a shared name (or address) for an object, though without protection.

Modern support for item 4 is not as strong as the first three, and is the subject of this dissertation. "Access to information in an authorised manner" is coarsely addressed by modern paging hardware at the page granularity by per-page permissions. The paging model only allows distinction between protection domains at the process level, and thus protected sharing of memory requires an application to be spread between multiple address spaces.

Role of languages in protection — It should be noted that protection here is of names and addresses as seen by machine code executing on a modern processor. Object protection as seen by the user has largely been accomplished by high-level languages with static analysis of syntax. If the programmer names an object that is not in his artificially small namespace, he will receive a syntax error. This namespace protection greatly simplified development but was not enforced in hardware and was easily bypassed by assembly routines which were common in early computing. This especially motivated the large number of machines that used segment and capability based addressing. Today's higher-level languages render the lack of protection of the underlying architecture almost unnoticeable to many programmers. However, while Java and other managed languages strongly enforce their namespace using both static and dynamic techniques, the widely used C and C++ languages expose underlying hardware addresses which can easily bypass the language namespace model. C and C++ programs include all major operating systems and web browsers as well as the managed language runtimes which need to expose the underlying address space for performance reasons. Therefore, while many programmers may not directly manipulate machine addresses, nearly every program depends on some code segments that expose the underlying addressing mechanism that is unlikely to be completely safe.

2.1.1 Protection of segment descriptors

Segment descriptors need to be protected since arbitrary modification of descriptors allow a process to bypass memory protection. Dennis and others at the time considered segment descriptors as aggregated in a global table rather than distributed in general-purpose memory. This descriptor table was a kind of global namespace that could be accessed by programs but was generally modified in a privileged mode[1]. This global descriptor table model was implemented most famously in IA-32.

Eventually high-level programming languages with structured programming enforced limited namespaces in the language model [22]. Computers built for use with high-level languages that supported segmentation generally allowed segment descriptors to be stored in a stack rather than in a global table to facilitate structured programming with its fine-grained, automated namespace changes. Protection of segment descriptors in a stack in main memory rather than in a dedicated segment required tagged memory in which every word of memory is tagged as either containing a segment descriptor or general-purpose data. This tag allows the processor to discern between general-purpose data and segment descriptors in order to protect descriptors from unauthorised modification. Segmentation implementations without a global list but using tagged memory include the Burroughs B5000 and the IBM System/38.

2.1.2 Development of capability systems

In the year after Dennis defended the need of segmentation in computer systems [26], he co-authored the seminal paper on capability systems with Van Horn [27]. This paper formally described pseudo-instructions for a cohesive memory protection framework in which every process possessed a list of keys, or *capabilities*, which gave it access to objects in the computer system. A capability is an unforgeable token of authority which could be used by a program component and which could be manipulated according to rules and delegated to another program component. One process could *grant* a segment capability to another process and could *enter* another process, which would pass control to the second process and change the list of available capabilities.

Dennis and Van Horn considered that the most common capability would be the *segment capability*, thus one could consider that the capability model was largely an effort toward safe and efficient management of segmentation. While the idea of capability system design has been applied to software on all levels, with or without traditional segmentation support, this original view is useful from the computer architecture perspective since any attempt at supporting capability systems with special hardware will look much like segmentation.

PSOS and capability system verification — Computer scientists spent much effort to prove that capability systems could be correct and safe. For example, in the mid 1970s, the Stanford Research Institute constructed a formal model of a Provably Secure Operating System, PSOS [61, 63, 60, 32, 62]. PSOS used multiple layers of increasing complexity to implement a full operating system where layer 0 was *capabilities.* The capability layer in PSOS required only that unique numbers could be created and compared. PSOS used

[1]The segment descriptor table was effectively a directory of global variables. This global directory followed from early problems encountered by teams of programmers struggling with the complexity of large programs in a shared memory. These developers wanted protection of memory space but expected to reference shared objects, or global variables, directly in assembly language.

a paged segmentation design with a segmentation layer above a paging layer [60]. The designers assumed that both of these layers would be in hardware due to performance requirements but that the higher layers could be implemented in software. Using only the capability primitive, segments as well as the rest of a general-purpose operating system was expected to be provably secure.

Capability implementations — While Dennis and Van Horn's pseudo instructions could have been implemented as low-level routines on top of computers with segmentation support, such as the Bourroughs B5000 which is referenced in the paper, several computers were soon purpose built to use the capability paradigm [53] [59], including commercial systems from Plessey [31], IBM [40], and Intel [36]. Many of these hardware capability systems were the pinnacle of program-centred, segment-based computers as their memory protection systems often did not depend on a central authority for inter-domain transactions, but on a set of hardware enforced rules that allowed safe, direct sharing between program components. These hardware capability systems did not dominate the computer industry because they sacrificed performance for memory safety in a time when RISC processors were demonstrating how much performance was available to simplified integrated circuit computers [36]. However, we can learn from these practical capability machines in our attempt to build a comprehensive segment system for modern RISC processors. I will summarise the major historical pure capability systems below to illustrate the breadth of approaches used to implement the capability segment model. In the description below I have emphasised the introduction of features that relate to the current implementation.

The Chicago Magic Number Machine — The Chicago Magic Number machine was an ambitious project at the University of Chicago that began in 1967 but was never completed [75, 53]. This machine was an attempt at a hardware implementation of a capability machine. In addition to 16 general-purpose registers, the Magic Number Machine had 6 capability registers with several fields including base, length, and permissions. Memory operations provided a capability register number and an offset into the capability that was either a general-purpose register or a static, immediate value. Capabilities referenced memory regions that contained either data or capabilities, but not both. A complete capability list for a program included capabilities in the 6 capability registers and any capabilities stored in memory addressable through those capabilities.

Plessey system 250 — The Plessey System 250 was the first commercial capability system and was a multi-processor system for extreme availability with a mean time between failure of 50 years [31, 53, 89]. The Plessey 250 had 8 general-purpose registers and 8 capability registers, C0-C7. C6 pointed to all capabilities for the current process and C7 pointed to the capability for the current program counter. There were also special-purpose capability registers for the interrupt blocks, a process stack, and the system capability table. In the Plessey 250 a process implemented protected procedure calls by giving away its "C6" containing its entire security context with the enter bit set. Another process could then "enter" a method in that context but could not directly touch its data. The Plessey 250 demonstrated that a capability system with a non-hierarchical, distributed trust model works well on a multiprocessor system.

Cambridge CAP — The Cambridge CAP computer was a practical capability machine that was developed and used between 1970 and 1977 at the University of Cambridge [59, 53]. The CAP computer also had a single global table of capabilities in the system and the capabilities available to each process in the system mapped into this global table. This resulted in a complex system of indirection for capability references which was made tenable using a capability cache. Each process in the CAP computer had 16 capability segments mapped from the system-wide list in which it stored its capabilities. Upon a protected procedure call, 11 of these capability segments were taken from the new process and 5 were preserved from the calling process as parameters to the procedure call.

Hydra — Hydra was a software capability system developed by Carnegie-Mellon University beginning in 1971 [95, 53]. Hydra used capabilities to implement an object-based operating system. Hydra encoded capability permissions as a bit vector in a capability where setting a bit always indicated granting of privilege and clearing a bit always implied losing privilege. Allowing only AND operations on the permission bit vector ensured only reduction of privilege. One of these bits was required for a process to retain the capability in its local namespace and if the bit was cleared, the capability could only be retained in the present invocation. Such "ephemeral" capabilities provided a simple method of revocation for parameters passed to untrusted procedures.

IBM System/38 — The IBM System/38 was an enterprise system developed for very long term support and was first released in 1979 [10, 40, 53]. IBM presented high-level capability based instructions to software and implemented these instructions with large sequences of microcode. The underlying hardware running System/38 programs has changed completely several times since its introduction but the software interface has remained compatible. System/38 uses a 64-bit address space and is able to generate a unique ID (and thus a unique address) for every new capability, thus avoiding security problems caused by address space reuse. To protect capabilities in memory System/38 maintains tags on every 32-bit word to ensure that capabilities may not be manipulated by general-purpose loads and stores. Capabilities are 128 bits long and must be aligned in memory to prevent misaligned capability fragments from being interpreted as new capabilities. System/38 programs are supported by current IBM Power Systems which emulate this behaviour in a virtual machine.

Intel iAPX 432 — The Intel iAPX 432 was a second major commercial effort to develop a capability based hardware system [53]. Launched in 1981, just as RISC processors were being introduced, the iAPX 432 implemented capabilities using a small execution engine and a large body of microcode. Every capability could contain both data and other capabilities in exclusive regions. The first 13 capabilities below the main pointer had hardware defined functions and the first 6 data locations above the main pointer were also hardware defined. However all operations were implemented using 4 capability registers of which only 3 were general-purpose. These choices of hardwired complex functionality over performance left the iAPX 432 to fail in the market against less complex processors which provided much better performance [36, 41].

2.1.3 Software implementations in the RISC era

As the RISC model of processing became dominant, researchers turned to pure software implementations of capability ideas. With respect to software capability systems on RISC, I have found it useful to examine Paul Karger's PhD dissertation from 1988 [47]. Karger's PhD work was sitting at the intersection of the retiring Cambridge CAP capability computer and the promise of skyrocketing performance from RISC processors. Though Karger primarily presents a capability machine built with VAX microcode, he also discusses a hypothetical capability system for a RISC machine. The themes he discusses presage the next twenty years of capability system research.

Software segment descriptors — Section 15.1.2 of Karger's dissertation is titled "Segment Descriptors in Software Only". In this section he first points out that page translation is more efficient than segmentation due to requiring only concatenation for address calculation rather than both addition and comparison. If necessary, permissions could be checked in parallel with the memory access with independent instructions which could invoke an exception handler on violation. However, Karger argued that this was rarely necessary at run time because "Bounds checking on data structures, as opposed to bounds checking required for security, can better be done by the compiler, either by flow analysis on the source code, proving that bounds violations cannot occur, or by including run-time checks in the generated code".

Karger's proposed approach of using language analysis plus run-time checks has been thoroughly explored in the years after his thesis. Software fault isolation, SFI, recommended this approach for sandboxing using binary analysis and modification in 1993 [86]. Java seeks to validate every memory reference, either by statically proving correctness or inserting run-time checks. Managed languages in general take this approach to enforce their segment model in the absence of hardware support for segmentation. Java takes this approach even on IA-32 where some support for segmentation exists. Cyclone is a safe dialect of C that employs the same strategies with hints from the programmer in the form of pointer types [45] and CCured infers pointer types from unmodified C [58]. A recent proposal, BGI, implements such analysis or run-time checks for C executables to verify that kernel modules access a restricted subset of memory on processors without special support for isolation [14]. Furthermore Google's Native Client uses this approach to validate that machine code distributed to the web browser respects the software view of segmentation with no special hardware support [96]. While the performance of these systems is surprisingly high, less than 13% on the SPEC2000 [96], the penalty is too great for the most performance critical paths in the system. Unfortunately performance critical components attain that distinction because they are the most used paths in the system and should therefore be the most secure.

Protection domain crossing — Section 15.2 of Karger's dissertation is titled "Context Switching" and deals with protection domain transitions. The majority of this section assumes that a protection domain transition means switching address spaces as is the case in RISC processors with a simple TLB. Therefore the hardware optimisations he describes for protection domain transition are optimisations for address space switching, which culminate in address space numbers for tagging TLB entries to avoid flushing.[2] Karger admits that this approach is only useful when protection domains are coarse but argues that protection domains rarely need to be fine-grained as evidenced by the trend of software on the CAP computer, which only used two levels of protection despite supporting hierarchical domains. Karger's model of using address spaces with TLB enforcement to defend protection domains of a capability system rather than to isolate independent programs has also been explored by a number of systems.

EROS — The EROS project used this model and was released in the late 1990s [74]. EROS was a microkernel operating system which presented a simple capability API to host protection domains which included components which were traditionally in the trusted

[2]Karger's last context switching optimisation is to allow the user to directly reference different address spaces, making the address space properly part of the virtual address. This is done in systems with hardware segmentation such as the IBM System/38 at the expense of making virtual addresses large. This extension crosses from traditional RISC into special hardware support for segmentation to support capabilities and so is not mentioned above.

kernel. As Karger had proposed, EROS used independent address spaces for each domain of protection. Furthermore the supervisor was invoked for all inter-domain communication as the trusted capability API depended on the hardware supervisor mode for protection. Since the hardware page table system was used to enforce capability access control, all data capabilities had a fixed size of 4096 bytes, the size of one page. If data was not available in the page table, the operating system was invoked and would check whether the desired address was within a capability owned by the current process. Thus the supervisor mode was completely trusted to both implement inter-process communication and to interpret capability permissions for each process.

The EROS developers requested two hardware features to accelerate capabilities: a fast privilege boundary crossing mechanism and a tagged or software-managed TLB architecture. Though tagged TLBs soon followed, the excessive supervisor calls for intra-application communication and even intra-system communication limited the EROS kernel's popular uptake.

Capsicum — Capsicum, presented in 2010 [87], allowed the process capability model to be used within existing operating systems. Capsicum is an extension of the UNIX API that allows programmers to express application partitions using a capability model alongside traditional applications and on a traditional kernel. Capsicum uses UNIX file descriptors as capabilities to delegate privilege to a constrained process so that a protection domain may access only those system and application resources which it requires. As with EROS, all inter-domain communication is implemented with inter-process system calls. Capsicum has been received with enthusiasm due to being interoperable with legacy UNIX systems while presenting a simplified API for application partitioning. CHERI draws inspiration from Capsicum's success in creating a capability model that can be adopted incrementally within a complex traditional computer system.

2.2 Processor support for memory protection

We have discussed segmentation and capability systems which are the direct inspiration of the current work. However, these are not the only mechanisms to support memory protection. This section will begin with the mechanisms which are currently in use and will proceed to new proposals for hardware support for memory protection.

2.2.1 Current mechanisms for memory protection

Current mechanisms for memory protection rely on the protection ring mechanism in modern processors [72]. Generally only two rings are used, supervisor and user, though a lower ring has been recently introduced for a hypervisor to administrate operating systems. User mode instructions are not trusted to manipulate memory protection primitives directly, rather routines that execute in the supervisor domain are trusted to maintain tables and registers on behalf of user code. Frequent recourse to supervisor mode is prohibitively slow so more frequent memory protection functions are implemented by trusted hardware mechanisms and these are in turn trusted to be administrated from supervisor mode.

Page table

Modern processors that are intended to run general-purpose operating systems usually implement a translation table in hardware. This table of page mappings is often called a Translation Lookaside Buffer(TLB). This table automatically translates every memory reference from a virtual address space into a physical address. The operating system kernel maintains a list of valid translations and populates the hardware table which performs automatic memory translation. If the table does not have a kernel-authorised translation for a virtual memory address a hardware exception is thrown. If the virtual address space is sparse, very few addresses will be valid and the paging mechanism can be used to verify valid memory references to a high probability, though at a page granularity. Using read, write, and execute permissions for each entry, operating systems can also enforce usage properties for memory regions.

While the page table is an excellent protection mechanism when the entire address space is a single domain of protection, the page table does not efficiently support multiple domains of protection within an address space, since user code is normally given access to its entire address space at run time. Allowing multiple domains of protection within an address space would require flushing the page table, or at least a portion of it, when crossing between protection domains in the same address space to ensure that only the allowed subset of pages is available to the new domain [82]. This additional complexity could easily be more expensive than simply using new address spaces for each domain.

Page tables can give some bounds checking support at a page granularity if structures are always separated by unallocated space. While this is useful in modern systems, finer grained bounds checking support is necessary in order to verify that memory references conform to the precise expectation of the program.

Protection table

Embedded systems often do not need memory translation or cannot tolerate the non-determinism of translation and therefore support only flat access to physical memory. However, these systems often have an even higher interest in memory protection than their general-purpose counterparts due to their use in critical control systems. To address this need, embedded processors often use supervisor managed protection tables which can enforce domains of protection within the physical address space. For example the ARM Cortex-R series features a memory protection unit (MPU) which holds a table of memory regions which may be accessed with properties for each region [5]. Unfortunately this table only allows region sizes that are a power of two with a minimum size of 32 bytes. However, embedded customers are demanding stronger protection. To address this demand, newer embedded processors, such as the Qorivva processors from Freescale, have arbitrarily sized protection regions with byte granularity [34].

Paged segmentation

For software composability and more modular protection, some systems support variable length segments within paged virtual address space. This scheme provides the user access to a list of segment descriptors which can be used to reference virtual addresses. Examples of systems that used paged segmentation include the GE-645 used for the Multics project [64] and the popular IA-32 ISA [44].

This paged segmentation mechanism has been used to run different processes in the same address space, avoiding the need to flush a non-tagged TLB between context switches[3]. However the segment mechanism can also be used to protect objects in a single process. Both Shinagawa [76] and Chiueh [18] used IA-32 segmentation to build systems that could sandbox modules in an address space to replace software fault islolation (SFI) [86].

While general segmentation architectures like IA-32 could use segments to protect individual objects in a program, I am not aware of any attempts to implement this in modern systems. This can be attributed at least partly to the IA-32 segmentation mechanism not matching the model used by modern compilers for managing object references. Also IA-32 depends on higher privileged rings for safe manipulation of segment table properties, so a path through an escalated privilege level is necessary when managing segments and when crossing between domains of protection.

Hardware bounds checking

Bounds checking is a very common operation used to enforce fine-grained protection. In an effort to ensure that programs access structures correctly, some high-level languages such as Java require bounds checks on every array access [20] and many C and C++ developers require bounds checks on every reference to improve code quality [29, 50]. Unfortunately implementing consistent bounds checking can often carry a penalty over 100% for array-heavy kernels [50, 30].

While segmentation hardware can be used for bounds checking, common segmentation hardware is not intended for fine-grained segmentation. For example creating and destroying segments on an Intel Pentium III costs 263 cycles even when the segment table is in user space [50]. Furthermore, selecting a different segment which is already in the table requires at least 4 cycles on the same system.

Due to these drawbacks of existing mechanisms, some instruction sets include special support for bounds checking. For example, IA-32 includes a BOUND instruction [19], though its performance varies greatly by implementation[4]. ARM also supported a special bounds check instruction in Thumb2-EE mode to support Java bounds checking. While bounds checking instructions can provide some speedup, Chuang *et al.* found that 75% of the cost of bounds checking when using a fast hardware bounds instruction was marshalling the base and top of pointer references [19]. This effect may have contributed to these instructions not finding widespread use.

Hardware bounds checking is an active area of research and we will discuss academic proposals and a coming implementation from Intel in section 2.2.2.

Protected Memories

Protected intelligent memories are another popular proposal to allow limited protected operations in a system. There are proposals from IBM [78, 79] and SP [52] from Princeton, and commercial implementations include TPM from Intel [6] and TrustZone from ARM [3]. These allow operations on protected memories by a trusted processor with a small software

[3]This arrangement comes with the restriction that the sum of the virtual address space used by all processes must be less than the total available in one address space

[4]AMD Athlon and Intel Pentium II based implementations had a latency between 6 and 8 cycles with a good throughput but recent implementations have latancies above 11 cycles, higher than equivalent general-purpose instructions [33].

base which allows interactions from the large untrusted software stack. This approach is useful for safely storing encryption keys or other infrequently accessed secrets but it does not virtualise efficiently and has not been useful for general-purpose application protection.

Intel's Trusted Platform Module (TPM) is a widely deployed industry standard and defines a very limited API for communicating with a simple microcontroller with a private memory. The TPM's chief function is to generate and use cryptographic keys without exposing them to the general-purpose system. The TPM can also create software and hardware signatures for trusted boot.

ARM has implemented a protected processor and memory architecture without using a dedicated processor. ARM uses second hardware context called *TrustZone* with an independent register file which can execute trusted software alongside the operating system. The ARM system bus, AXI, is aware of the security context of requests from the processor and thus a system on chip designer can make security critical memories and peripherals exclusive to the secure context.

While the TrustZone approach is more flexible than the TPM approach, these protected subsystems only work well if security requirements can be structured as a strict dichotomy. For example if a single data structure must be protected, a general-purpose operating system can communicate securely with software running in the secure context to interact with the data structure in protected memory. However, if critical security relationships become complex, the trusted software quickly becomes too complex to be trustworthy.

Capability segmentation can achieve much of the same goal while being more flexible. The capability segmentation approach allows regions of memory to be reserved in early system boot to trusted software routines. Once these regions have been removed from the pool available to the rest of the operating system, it should not be possible to regain access to these regions without the cooperation of the trusted routines. This technique would emulate the protected memory approach with a finer granularity and more flexibility.

2.2.2 Proposed mechanisms for memory protection

In this section I will briefly discuss a few additional categories of hardware protection that researchers have proposed in recent years. Chapter 4 will compare CHERI's memory protection to the first four of these proposals.

Protection table for virtual address space

Witchel *et al.* have proposed that multiple protection domains share an address space by adding a protection table for virtual addresses with Mondrian Memory Protection, or MMP [91, 93]. MMP superimposes a segment structure over virtual memory. The segmentation structure is defined by a descriptor table in memory which is cached in a protection table, similar to the one described in Section 2.2.1. When moving between protection domains in an address space, the processor must duplicate segment descriptors for a common region, but the page table entries for the region can be shared.

This protection table is similar to a page table and incurs similar performance overhead if segments are of equal granularity. Therefore MMP has good performance if the number of segments is much smaller than the number of pages, as the price of looking up the descriptor for a segment will generally only be paid once for translation. For this reason MMP is especially appropriate for coarse-grained sandboxing or for protecting relatively large structures. However, fine-grained object protection would cause excessive pressure on an associative table, as it does with the TLB, causing performance to revert to that of TLB protection, with the advantage that protection is more precise and does not require a sophisticated arrangement of objects in memory.

Automated out-of-band pointer bounds

Devietti *et al.* proposed a hardware accelerated fat-pointer scheme that is transparently compatible with existing software called *Hardbound* [28]. Hardbound stores the bounds information for each pointer in a shadow address space at a fixed offset from the pointers themselves. This allows the format and value of pointers in data structures to be unchanged, allowing transparent sharing with code and libraries that do not use the bounds checking scheme.

The Hardbound model has the disadvantage that the cost of protection does not scale with the granularity of enforcement. A single domain of protection, *i.e.* a sandbox, will pay the price of propagating bounds on every pointer load and store even if all bounds are identical.

Bounds management instructions

Intel has recently announced Memory Protection Extensions (iMPX) to accelerate pointer safety which will ship in future Intel processors [43]. Rather than the automated bounds management and enforcement provided by MMP and Hardbound, iMPX simply provides instructions for creating, moving, and checking bounds. Notably, iMPX includes instructions for inserting bounds into a shadow memory structure similar to Hardbound.

While iMPX allows incremental adoption and is practical in a modern pipeline, its protection mechanisms are optional and bypassable by malicious code. Also, like Hardbound, iMPX does not accelerate sandboxing without checks on every memory reference. Unlike Hardbound, however, iMPX is flexible enough to allow a single set of bounds to be applied to all references.

Compressed RISC capability addressing

The M-Machine was a research machine developed during the 1990s which used efficient, RISC capability addressing [13]. The M-Machine encodes base, bound, current pointer value, and permissions into every 64-bit pointer so that every dereference is guarded. Encoding segment descriptors in the register file meant that the protection domain of the process was entirely defined by the current register set and a multithreaded implementation naturally runs each thread in an independent protection domain. This design allows many standard C constructions to work as expected but due to the change in the binary format of pointers and elevated privilege for some pointer manipulations, the M-Machine model is not transparently compatible with existing systems or programs. While CHERI follows a similar implementation path to the M-Machine, CHERI modifies an existing instruction set to allow experimentation with existing operating systems and programs which were not possible on the M-Machine's radically new architecture.

Hardware types and information flow tracking

Information flow tracking has also attracted proposals for hardware support [80, 77, 24]. Information flow tracking systems attempt to trace the effects of untrusted or classified data through a computer system. A system that supports information flow tracking prevents untrusted data from being used for a trusted purpose (e.g. as a pointer) or prevents confidential data from being used in a public context (e.g. as data in an out-going TCP packet).

Information flow and typing can be closely related since taint or clearance level can be encoded in a data type [85]. A typing system generally provides a type for every operand in a computation and assigns an appropriate type to the result. If each operation tags its result with the appropriate taint level based on the taint level of its operands, the system can certify a trust level for every piece of data in memory.

Hardware information flow proposals generally tag every word of memory with a type, or taint level. Suh *et al.* propose a single bit for every byte of memory to indicate that it is tainted, e.g. that it has come from the network or from a file [80]. TIARA and the SAFE project propose a larger tag to indicate the owner of each word of data as well as a type for each word [77, 24]. A hardware cached rule table specifies the new tag for each instruction result.

The current work on capability segmentation is somewhat orthogonal to information flow control, though they can be used to solve some of the same problems. While information flow control can prevent untrusted data from being used as an address because it knows that it is untrusted, a capability segmentation mechanism can prevent its use as an address because it knows that it is data. Furthermore, the information flow control approach does not have a high-level view of object structure, so it requires tags on every word of memory. The capability segmentation mechanism only requires one reference with typing information for each region of memory, which means that the cost of typed protection scales with the granularity of use.

Expanded watchpoints for instrumentation

Code instrumentation is another active area of research for computer security [35]. Several frameworks for instrumentation have been developed that insert assertion function calls into the program instruction stream at a heavy performance cost [97]. It is possible to use the watchpoint registers in some architectures to instrument a limited number of points in a program without rewriting the program, but this is too limited for a general-purpose instrumentation system. Zhou *et al.* have proposed a more general-purpose watchpoint system that allows an arbitrary number of regions to be watched with assertion routines assigned to each that can be run in parallel [97]. This “iWatcher” system relies on thread-level speculation which is not implemented natively on any current hardware. However, we should note that a general instrumentation system would have a ready community of consumers.

A fine-grained segmentation mechanism could assist instrumentation by allowing watchpoints to be encoded in segment descriptors, which are protected by hardware even in memory. This would allow arbitrary regions to be watched by the system. Indeed, simply giving a descriptor no permissions will cause an exception to be thrown when it is used, emulating a watchpoint. However it could be useful to allow specific user-level functions to run upon use of a segment.

2.3 Conclusion

Capability segmentation has a deep history in computer design and a wide applicability to current memory protection problems. Capability system theory, as first described by Dennis and Van Horn [27], has been tested and is understood in an array of contexts and granularities though it is lacking sufficient segmentation support in modern hardware. However, increasing scrutiny of memory protection is generating demand for stronger memory protection mechanisms. Current mechanisms depend on supervisor privilege and therefore tend to be coarse-grained to avoid excessive overheads. The desire for programs that are both high-performance and memory-safe motivates fine-grained, user-mode memory protection. I would like to demonstrate that capability segmentation can fill this need by lending hardware support to state of the art memory protection techniques including sandboxing, bounds checking and even static analysis.

Chapter 3

A RISC model for segmentation to support capability systems

3.1 Introduction

I propose to include fast, general-purpose segmentation support in modern RISC-style processors to accelerate memory-safe computation. Lack of hardware support for memory safety mechanisms in user space has resulted in a great deal of effort to guarantee memory safety either by static analysis or by run-time enforcement, and these two strategies at various granularities have developed into multiple branches of computer science, some of which were discussed in Section 2.1.3.

These efforts toward memory safety have been particularly fuelled by escalating problems with security do to rising financial rewards and highly-polished hacking tools which allow developers to quickly disseminate exploits [4, 49, 7]. Many exploits, such as the recent OpenSSL "Heartbleed" exploit [46], are due to the lack of strong memory protection in performance sensitive programs as reviewed in Chapter 2.

Segmentation is the view that memory is not a flat space but a collection of objects with properties, or, closer to the implementation, collections of memory ranges with permissions [26, 25]. Whether a processor enforces segmentation or not, predictable computer operation nearly always depends on a segmented view of memory as expressed in a high-level language, or at least in the minds of assembly programmers. Memory safety is the effort to enforce this segmented view of memory, either using software or hardware primitives.

The consensus memory model for general-purpose processors enforces a segmented view of memory using address spaces and paging, neither of which are visible in the program abstraction, that is, to instructions running in user space. Nevertheless user-space programs have segmented views of their own address spaces at many levels. Nearly every programming language above assembly has an object model with an abstract machine that reasons on the level of memory objects. At a coarser granularity, address spaces are populated with multiple executables, libraries, files, and stacks, which are considered distinct objects, or memory segments, by developers and users but which share a flat address space with equal access in hardware. This mismatch between the abstract machine and the actual machine has caused a constant tension between correctness and performance. High-performance segmentation support in hardware would make this trade-off much less precipitous, allowing full correctness with little overhead.

While the need for user-mode segmentation has been escalating, hardware vendors have deprecated what little hardware segmentation support existed in modern processors. The x86-64 instruction set dropped general-purpose segmentation that had been introduced with the Intel 80286 processor, though IA-32 implementations were already too slow to be useful for fine-grained protection. Commercial RISC processors did not attempt fine-grained segmentation support for user space.

In this chapter I will propose a general-purpose capability segmentation model within virtual address space for RISC style execution that maintains the single-cycle-instruction principle and a load-store architecture. I show that it is practical to extend a RISC instruction set to efficiently implement safe segmentation in user space as a practical compiler target, and that is also transparently compatible with the flat address space memory model. I present an example of such an extension to the 64-bit MIPS instruction set and discuss common use cases.

3.2 Motivation for renewed segment support

While hardware segmentation was being deprecated, modern software was going to great lengths to implement segmentation using hardware primitives not intended for the purpose. These efforts have spawned entire branches of computer science. One branch is the field of managed language virtual machines that emulate segmented memory to enforce an object memory model using a blend of complex static analysis and general-purpose instructions to check bounds at run-time [65]. Another branch of software segmentation is the field of software sandboxing, which has sought to use process separation [72, 87], disjoint TLB sets in a shared process, and static analysis of specially compiled general-purpose instructions [96, 86, 55]. Even microkernel efforts were motivated by a lack of tools to segment the kernel into varying domains of trust, leading to movement of kernel functionality into user space processes at a heavy performance cost [1, 74, 54].

Even C, a low-level language, contains an object-like memory model and loses segment information when compiled to run on processors with a flat address space. Below is sample of LLVM intermediate representation generated from the Clang C compiler for an array access:

```
%0 = getelementptr inbounds i32* %array, i64 %idx
%1 = load i32* %1
```

The `inbounds` annotation on the `getelementptr` instruction indicates that behaviour is undefined if the index is not within the object. This information is lost when mapping the C abstract machine onto a flat memory machine. While there are a number of compiler options that attempt to implement these checks in the executable, too much performance is lost, as we will see in Chapter 7, and these are rarely used in production code.

Google's Native Client has also been frustrated by a lack of segmentation in hardware. Google Native Client attempts to execute native code inside a JavaScript framework that can be verified to not reference memory outside of a restricted range. The obvious solution to these requirements is a simple segmentation scheme that would check the bounds of every access. As a result, Google researchers implemented an early version of Native Client using the IA-32 segmentation mechanism. However, they discovered that the Intel Atom processor had a large performance penalty when the code segment was not based at zero, indicating that segmentation was effectively deprecated in recent implementations [42].

When this problem was added to the fact that the newer x86-64 processor mode does not support arbitrary segmentation at all, Google researchers had to abandon the natural segmentation implementation for Native Client and rely on exotic compilation techniques to allow static verification of memory behaviour.

Fast, general-purpose segmentation support would allow the semantics of C and higher level languages to be enforced by hardware and would also allow conceptually-simple sandboxing to be simple in implementation.

3.3 CHERI capability segmentation

I propose a general-purpose capability segmentation model that is managed by user space instructions and that conforms to a prototypical RISC ISA philosophy. As a segmentation mechanism, protection should not be bypassable for any memory access. As a capability model, segment descriptors should be unforgeable. While we could simply accelerate segment manipulation and addressing without capability-style protection, as is done in iMPX from Intel, multi-library applications cannot trust all of their components and need a high-level of assurance on the order of operating system protection mechanisms. Therefore, I propose a capability mechanism that can be verified to be as strong as kernel and user separation, while being flexible enough to allow decentralised protection domain management to match non-hierarchical program structure.

Such a verifiable model of capability segmentation must enforce three properties:

1. All memory accesses are through a segment descriptor.
2. Segment descriptor manipulation is protected.
3. The program must be able to pass control between domains possessing different segments without granting additional segments to either domain.

Indeed, these three properties are guaranteed in most segmented memory architectures. I would add four additional requirements that would make a segmentation model fast enough to be useful from user space:

4. Segment-relative memory access should be as fast as flat memory access.
5. Safe segment manipulation should not depend on kernel level protection.
6. No segment related instruction should require more than one cycle in a classical RISC pipeline.
7. Segment adoption should be optional and incremental.

The fourth, fifth, and sixth properties were proposed for the M-Machine, though the M-Machine implementation only allowed pointer manipulation in the kernel [13]. These three properties make capability segmentation a practical replacement for pointers. It has been observed that memory safety features will be adopted only if performance impact is below 10% (with stronger uptake below 5%) [81], so performance of the segmentation mechanism is crucial if memory safe programs are expected to be the common case. The sixth property, the single cycle property, ensures that a safe segmentation model does not require exotic behaviour and can be implemented with reasonable complexity in any modern architecture. The seventh property, the optional and incremental property, is important for long-term adoption. Approaches that require completely rewriting large amounts of legacy code have rarely succeeded.

While the M-Machine respected all but the last of the properties[1], the design was not available to the community to explore compiler and system ramifications. Only the last principle, the optional and incremental principle, was violated by the M-Machine due to its new ISA and mandatory, exclusive use of the capability protection model.

3.3.1 Our Base Architecture: BERI

I have chosen to implement a RISC capability segmentation model by extending the 64-bit MIPS instruction set. I selected MIPS because it has a well established 64-bit specification and adheres to a prototypical RISC philosophy. The processor is based on a user-mode, multi-threaded 64-bit MIPS implementation in Bluespec SystemVerilog built by Gregory Chadwick [16]. I extended this processor to a general feature parity with the MIPS R4000 (minus floating point support[2]), and some legacy 32-bit memory space support. As with the MIPS R4000, the base processor (called BERI, Bluespec Extensible RISC Implementation) is single issue and in-order with a throughput approaching 1 instruction per cycle. BERI has a branch predictor and uses limited register renaming to allow for arbitrary buffering in its 6-stage pipeline to ease extensibility and avoid end-to-end combinational paths. BERI runs at 125 MHz on an Altera Stratix IV FPGA. I have thus far preferred correctness and extensibility to performance, especially clock speed, as we consider BERI a research platform whose value is in low development time rather than low execution time. Nevertheless, the performance of BERI is sufficient to support software development on the platform itself. The extension of BERI to support RISC capability segmentation is detailed below.

3.3.2 Segment descriptor table

CHERI implements the first segmentation requirement, the universal enforcement property, by adding a 32 entry segment table as a register file through which all virtual addresses are offset before reaching the page table. CHERI segment descriptors are 256 bits wide to allow maximum flexibility for design space exploration. The currently implemented structure is presented in Figure 3.1. The **base** and **length** fields are the two basic fields needed to describe a segment of memory. CHERI allocates a simple 64-bits to each and does not use any sort of compression algorithm at this time to avoid premature optimisation. The **permissions** field is a 31-bit vector with a “1” in each position indicating an allowed privilege for the region. The first 15 bits in this field are hardware defined privileges which include load data, store data, and execute as well as load and store for segment descriptors. The other 10 hardware privileges are experimental at this time, and the top 16 bits of the field preserve “clear-only” semantics but do not have hardware implications.

The **u** bit in Figure 3.1 indicates that a capability is *unsealed* and can therefore be manipulated and derefrenced. This bit is set and cleared according to special rules to allow domain crossing with mutual distrust. The **otype/eaddr** field is a virtual address that

[1]As proposed the M-Machine allowed segment manipulation in user space, though the implementation required a trap to manipulate pointers, violating the fourth, user-mode principle.

[2]A fully featured floating point unit for BERI has been developed by Colin Rothwell.

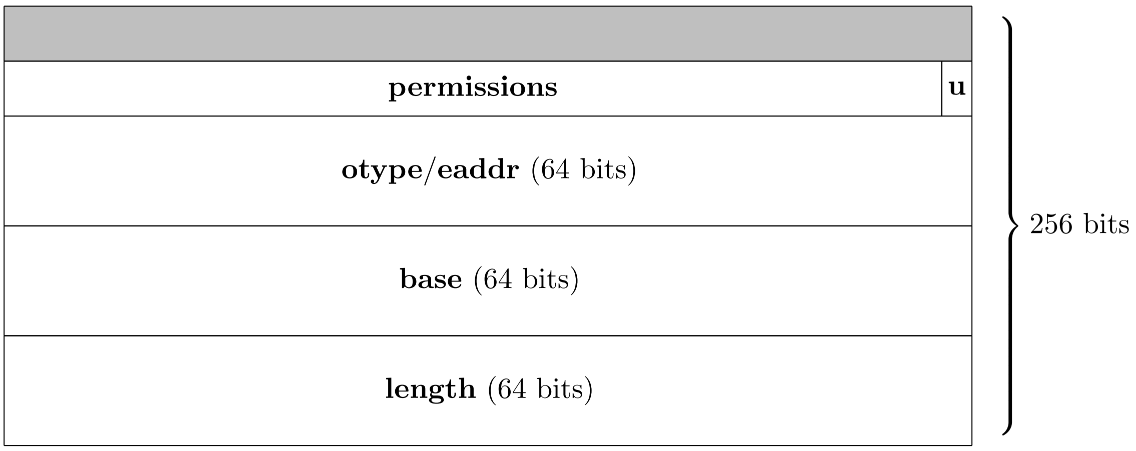


Figure 3.1: Capability Segment Descriptor

acts as both an entry point for an executable capability and a type for a sealed capability. An executable capability may have its **otype/eaddr** set to some address within its span and this capability may then be used to set the type of a data (non-executable) capability. Both the executable and data capabilities can then be sealed (**u**is cleared) and returned to an untrusted party. This party can then invoke the executable capability with the data capability, passing control back to the original domain, with neither capability being manipulable by the untrusted party. Further details of the domain crossing model are discussed in Section 3.4.

Existing MIPS load and store instructions are implicitly offset via segment 0 in the table (*C0*) and instruction fetches are offset via an implied program counter segment (*PCC*, the Program Counter Capability). This arrangement allows legacy code to run unmodified in our architecture to enable incremental adoption. This fixed implicit offset model is somewhat similar to the x86 segment model. In addition I have added load and store instructions that allow explicit use of arbitrary segments in the table as a base register. These new loads and stores custom implementations of the *LC2* and *LDC2* instructions intended for loads and stores to coprocessor 2. As MIPS lacks native register indexed memory instructions, using segment registers can often be faster than those using general-purpose registers for addressing. The virtual address calculation for the three basic cases of instruction fetches, legacy loads and stores, and capability loads and stores are illustrated in Figure 3.3 and the set of capability registers is listed in Figure 3.2.

CHERI descriptor compression — We might note obvious capability compression opportunities here. The capability fields will rarely be observed directly by general-purpose instructions and will never be written by them, so there is little motivation to align addresses in the capability structure. 64-bit MIPS traditionally has a 40-bit virtual address space, not counting the address space segment identified in the top 5 bits so the **base**, **length**, and **otype/eaddr** fields could be shortened to 120-bits together. The **otype/eaddr** field is also a program counter entry point, so might reasonably be required to be 32-bit instruction aligned, freeing up 2 more bits. The 5 bits of the address space segment (e.g. mapped, uncached, etc.) should be easily compressible into 3 bits since not all address space segments are used, freeing up an additional 2 bits. This would leave space for **u** and 6 **permissions** bits in a 128-bit capability. 6 **permissions** bits is likely to be sufficient in a mature implementation, especially with some permission encoding.

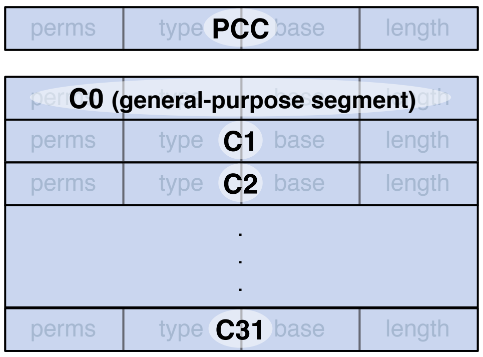


Figure 3.2: Segment descriptor registers

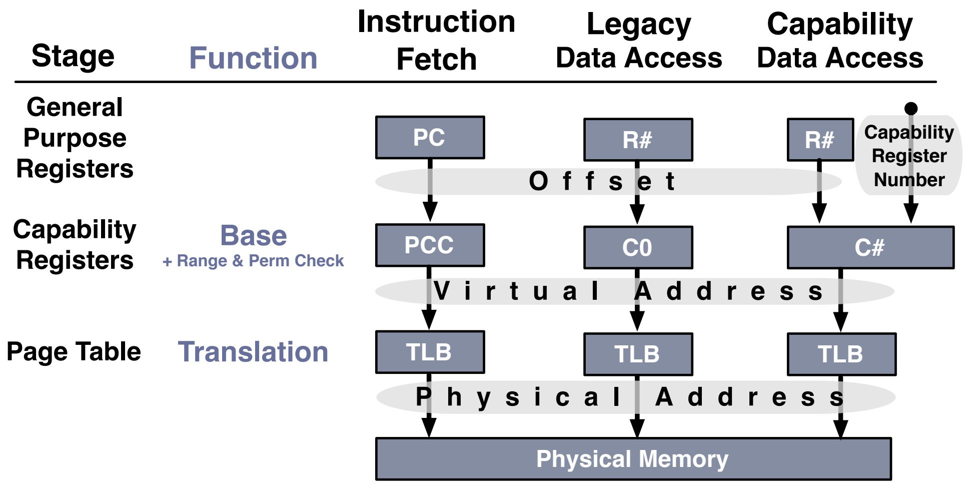


Figure 3.3: CHERI virtual address calculation

Further compression without complex requirements could be attained if two descriptor sizes were supported. For example, a 64-bit descriptor might contain a 43-bit **base**, a 14-bit **length**, as well as **u** and 6 **permissions** with no **otype/eaddr** field. This would allow deriving small capabilities within a protection domain for bounds checking small, fixed-length objects.

3.3.3 Segment descriptor manipulation and protection

Segment descriptors that are protected from arbitrary manipulation become capabilities, unforgeable tokens of authority to a contiguous region of address space. The biggest challenge for RISC capability segmentation support is meeting requirements 2 and 4, that is, protecting segment descriptors from arbitrary manipulation without appealing to the kernel. To meet these requirements CHERI prevents any modification of segment descriptors by general-purpose instructions. To do this we must distinguish not only between general-purpose registers and segment descriptor registers (which is enforced by the ISA structure), but also between memory locations that hold segment descriptors and those that hold other data. To distinguish between segment descriptors and general-purpose data the processor maintains a single bit tag for each 256-bit line of main memory. The tag

Mnemonic	Description
CIncBase	Increase base and decrease length
CSetLen	Set (reduce) length
CClearTag	Invalidate a capability register
CAndPerm	Restrict permissions

Table 3.1: Segment manipulation instructions

bits are stored in a hardware-managed table in memory, as described in Section 5.5.2. Each segment descriptor register also stores the tag bit to allow tag-preserving data movement through capability registers. The tag on a memory location can only be set by a write of a descriptor from the segment descriptor register file whose tag bit is set. A store from a general-purpose register will clear the descriptor tag in memory. An exception will be thrown on a dereference of a segment descriptor whose tag bit is not set. Using disjoint register sets in the ISA and tagged memory, segment descriptors are protected from general-purpose modifications without appealing to kernel mode and we achieve a user-space capability addressing model.

3.3.4 CHERI instruction types

The CHERI register file is modularised as a MIPS coprocessor and occupies the MIPS coprocessor 2 instruction encoding space. MIPS has 4 coprocessors defined in the ISA and coprocessor 2 is traditionally implementation dependant. MIPS coprocessors, which include the system control processor (CP0) and the floating point unit (CP1 and CP1x, i.e. CP3) have 32 primary registers with up to 8 selectable shadow registers for each. CHERI roughly follows this convention by implementing 8 capability registers and exposing the 64-bit fields of the 256-bit registers as shadow registers. Capability manipulation instructions occupy the **CP2** instruction space and generally follow MIPS **MTC** and **MFC** (move to/move from coprocessor) conventions, but add extra fields when necessary. Capability loads and stores use the **LWC2**, **LDC2**, **SWC2**, and **SDC2** instruction encoding space. These are adapted to provide coprocessor base registers as well as general-purpose register offsets and immediate offsets.

CHERI descriptor manipulation instructions — CHERI descriptor manipulation instructions, which are listed in Table 3.1, are unprivileged and exercised directly by user code. In order for these to be safe, they must strictly reduce privilege. Therefore CHERI does not have an instruction to set an arbitrary base, but rather include the **CIncBase** instruction (capability increase base) which can only increase the existing base with a corresponding decrease in the length field, thus strictly subsetting the segment descriptor. Similarly, CHERI does not have an instruction to set arbitrary permissions but has the **CAndPerm** instruction which performs a logical-and with permission bits and a general-purpose register value, again strictly reducing privilege. The other instructions in Figure 3.1 follow a similar pattern and facilitate a process that begins with a segment descriptor for all privilege to its virtual address space to safely construct arbitrarily restricted domains. These instructions, as well as all other CHERI instructions that do not access memory, are encoded within the **CP2** MIPS instruction subspace.

Mnemonic	Description
CGetBase	Move base to a GPR
CGetLen	Move length to a GPR
CGetTag	Move tag bit to a GPR
CGetPerm	Move permissions field to a GPR
CGetType	Move object type field to a GPR

Table 3.2: Segment observation instructions

CP2 (0x32)	rd	cb	rt	offset	1	0	`CLB rd, rt, offset(cb)`

rd = MEM[cb.base + offset + rt]

Figure 3.4: Instruction encoding for CLB, capability load byte

CHERI descriptor observation instructions — CHERI also provides instructions to read fields of capability descriptors which are listed in Table 3.2. These are not essential for correct operation but increase usability. For example, an operation that is not permitted by the permissions field of a descriptor will throw an exception, but a program may prefer to check permissions directly by reading the **CGetPerm** field of the descriptor. The length of an object can be read using **CGetLen** which may be useful for loops, and **CGetBase** might be used to cast segments to standard pointers for compatibility with legacy code, which would need the general-purpose segment (*C0*) based at virtual address 0.

Segment addressing instructions — CHERI instructions that dereference segment descriptors are listed in Table 3.3. Nearly all combinations of MIPS loads and stores are replicated, including load linked and store conditional of double values. These instructions enable applications to move to a full capability model with no memory accesses through capability 0, the implied segment descriptor. We have not replicated load and store left and right instructions since they are not commonly used by compilers. Capability load and stores have both a register offset and an 8-bit immediate offset as shown in the instruction encoding given in Figure 3.4. This preserves the existing MIPS convenience of immediate offsets for structures while providing the additional flexibility of addressing relative to a larger region described by a capability.

Segment control-flow instructions — CHERI implements two types of branches that depend on the segment descriptors which are listed in Table 3.4. The first type includes **CJR** and **CJALR** which jump into an executable segment descriptor at an offset provided in a general-purpose register. To "jump" in the context of the capability register file is to move a segment descriptor into the program counter capability (**PCC**). **CJALR** additionally stores the existing **PCC** into another capability register. The second control-flow instruction type, proposed by David Chisnall, branches based on the validity of a descriptor. These allow a program to efficiently observe the presence of a reference which can be very useful for garbage collectors. These also allow dynamic languages to replace pointers with the actual data so that, if data is immutable and smaller than the reference, data can be loaded directly instead of the pointer followed by the data.

Domain crossing instructions — CHERI instructions that are used to create and cross between protection domains are listed in Table 3.5. These instructions allow a

Mnemonic	Description
CSC	Store capability register
CLC	Load capability register
CL[BHWD][U]	Load byte, half-word, word or double via capability register, signed or unsigned
CS[BHWD]	Store byte, half-word, word or double via capability register
CLLD	Load linked via capability register
CSCD	Store conditional via capability register

Table 3.3: Segment addressing instructions

Mnemonic	Description
CJR	Jump capability register
CJALR	Jump and link capability register
CBTU	Branch if capability tag is unset
CBTS	Branch if capability tag is set

Table 3.4: Segment control-flow instructions

program to assign a type to segment descriptors and seal them so that enterable segments can be safely distributed to and invoked by untrusted domains. The CHERI domain crossing model is discussed in detail in Section 3.4.

Mnemonic	Description
CSetType	Set the type/entry point of an executable capability
CSealCode	Seal an executable capability
CSealData	Seal a non-executable capability
CCALL	Call protection domain
CRETURN	Return from protection domain

Table 3.5: Domain crossing instructions

Tagged Memory — While we implement tagged memory to protect segment descriptors, it might be noted that it is possible to protect descriptors in memory without tags by simply using the load and store descriptor permissions. Using only the permission bits, one could construct disjoint address spaces for general-purpose data and for segment descriptors. However compilers of modern languages routinely store pointers in data structures and depend on a stack to preserve state, so a model that allows a shared memory space for instructions, data, and descriptors is much preferred to meet our goal of a viable pointer replacement. Therefore CHERI allows intermixing of segment descriptor capabilities and data with the classic solution of tagged memory. Since descriptors are

256-bits wide and must be aligned in memory, tags have an overhead of only 0.4% of physical memory and are stored in a hardware-managed table in DRAM. These tags can be cached both in a tag controller and along with lines in memory, since they are associated with physical addresses.

3.4 CHERI domain crossing

A segmentation mechanism enables multiple protection domains within an address space. For this mechanism to be useful, an instruction set must provide a way to pass safely from one protection domain to another. A CHERI protection domain is the transitive closure of capabilities reachable through its register set, that is, the set of capability registers as well as the capabilities reachable through them. While traditional MMUs implement protection domain crossing with a page table change, CHERI protection domain crossing must only change the current capability register set. Furthermore, to support full mutual distrust, it must be possible to distribute a handle that allows a domain to be entered but does not grant any other authority to the domain.

The M-Machine solved this problem by allowing a capability to be "sealed" such that it could only be entered at offset 0 (i.e. it only authorised jumps to the base of a sealed capability) but could not be otherwise dereferenced, that is, the descriptor could only accessed after flow control had passed to instructions in the protected region. This RISC invocation model assumes that both domains can be trusted to store, invalidate, and restore his own state according to his integrity and confidentiality requirements. While this model provides a workable mechanism, it requires that code be mixed with descriptors and data since any data required for a domain must be available in the same segment as the entry point of that domain.

With CHERI I have designed a more flexible domain crossing model while still allowing domain crossing in a single cycle. CHERI's **CCALL** instruction, diagrammed in Figure 3.5, takes two "sealed capabilities", one executable and one non-executable (sealing is described briefly in Section 3.3.2 and in detail in the next paragraph). The executable capability is placed in PCC and the non-executable is unsealed and placed in an architecturally defined capability register. Writing both the PCC and a general-purpose register in a single cycle takes precedent from the MIPS jump and link instruction.

CHERI dual-capability sealing is more complex than M-Machine single-capability sealing. CHERI requires that pairs of sealed capabilities must be validated as belonging to the same protection domain during invocation. This validation uses the **otype/eaddr** field of the capability. CHERI uses the **otype/eaddr** field as both an entry point in an executable capability and the type. Conflating these two means that the ability to execute from an address also grants the right to create objects of that type. This model follows the object-oriented class philosophy that considers instances of a class belonging to a single class executable. If you can execute a class directly, then you have the authority to create or manipulate instances of that class.

Sealing a pair of capabilities requires three steps:

1. The **CSetType** instruction sets the **otype/eaddr** field to the absolute address of an entry point within that capability.

2. The **CSealData** capability takes as operands the typed code capability which has a valid type and also an unsealed non-executable (data) capability and creates a

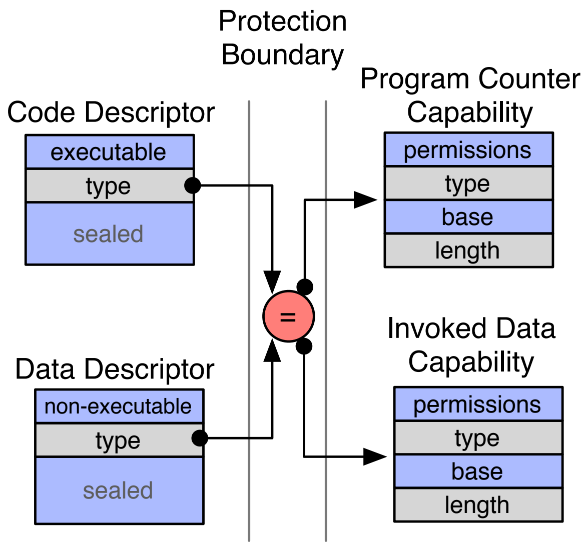


Figure 3.5: CHERI **CCALL**

sealed copy of the data capability with the same type.

3. The **CSealCode** instruction can then seal the typed code capability.

These two sealed capabilities can then be distributed to an untrusted domain which can invoke them as a pair but cannot access them directly. Upon invocation, the executable capability will be placed in **PCC** and the data capability will be placed in an architecturally-defined register in the capability register file. The code in the unsealed capability can then unpack its state from the unsealed data capability, confidant that it was prepared by itself (or someone who had access to its unsealed instructions).

While the M-Machine encapsulated a domain in one sealed capability that must hold both instructions and data for the domain, CHERI decouples these into two capabilities. Decoupling the code and data segments allows a more efficient mapping to the object model which decouples the implementation of a class from instances of that class. That is, a single descriptor for the methods of a class can be used on many objects of that class. However decoupling the code and data segments brings the additional complexity of verifying that the data capability "belongs" to the executable capability. Without this protection, an arbitrary sealed routine could be invoked with sealed data, exposing that data to untrusted code. Conversely, corrupted data could be presented to a sealed executable.

This model allows capability-aware code to cross domains and share references to objects in a fine-grained manner. Code that is unaware of segmentation must use contiguous segments described by capability register 0 and will therefore not be able to share access to memory without using overlapping memory regions, though **CCALL** can still be used in a capability-aware trampoline to move securely between these domains.

While this model was proposed and implemented by myself, conflating the code capability entry point and the type field was suggested by Ben Laurie, and the model was formally analysed by Michael Roe. Robert Watson was heavily consulted to validate practicality for software models.

This model has three levels of implementation, software **CCALL**, hardware **CCALL**, and hardware **CCALL** with **CRETURN** using a stack. The first two are implemented and the last is proposed.

3.4.1 Software CCALL

The first model throws an exception when **CCALL** is executed leaving a unique exception code in a privileged capability cause register to be inspected by the exception handler. This model is slow but allows full software flexibility for experimentation. Furthermore, it allows the operating system to instrument domain crossings and keep a call stack so that control can be returned to the calling domain if the invoked domain violates some policy. This is the model currently used for experimentation in the FreeBSD operating system extended by Robert Watson. The permissions checks described above have been implemented in hardware to reduce the effort of the exception handler.

3.4.2 Hardware CCALL

I have also implemented a simple hardware **CCALL** which behaves as described above, comparing types and entering the new protection domain. This model allows simple invocation and does not enforce higher-level semantics. If a language requires a secure call stack or other special semantics, they must be constructed manually. A call stack, for example, could be constructed by forcing all domain invocations through an intermediary in the language runtime. The sealed capabilities that are to be invoked would be passed as arguments in an invocation of the runtime domain crossing routine.

Furthermore, this model does not have a **CRETURN**. The calling domain must pass sealed return capabilities explicitly as parameters. The invoked routine will then **CCALL** this pair of capabilities to return to the caller.

This simple hardware **CCALL** model is implemented and is evaluated in Section 3.4.

3.4.3 Hardware CCALL with trusted stack

During the ISA design phase, Robert Watson and Ben Laurie considered that a trusted stack might be crucial to domain crossing usability and assurance. In response I developed a proposal for a trusted stack maintained by hardware. One capability register is architecturally designated for the trusted stack and is protected from user-level manipulation. In this model, a **CCALL** would push **PCC** as well as the register that is to be replaced by the unsealed data capability by **CCALL**. These two capabilities would be stored to memory pointed to by the trusted stack capability. This would require two cycles in our pipeline because we have only one port to memory. **CRETURN** would take no arguments but would return to the top entry on the trusted stack. Since only the invoked data capability is preserved upon a return, the calling domain would need to store all the capabilities in his register file into this segment to allow state recovery.

While the hardware trusted stack model gives the highest performance for call/return domain crossing semantics, it embeds much more behaviour in hardware. We do not yet know whether this behaviour will closely match the majority of software use cases for protection domain crossing, so it has not been implemented in hardware (the current software implementation does use a trusted stack).

3.5 Use cases for user-mode segmentation

User managed hardware segmentation, or capability addressing, has a variety of potential use models, most of which are unexplored in the context of modern operating systems. Therefore, we have attempted to design a basic hardware mechanism that is flexible enough to explore a number of models. This section lists some anticipated use cases, though we expect many others will be possible. Cheap segmentation radically changes the memory safety trade-off topography and there is hope that the open-source CHERI hardware platform will enable exploration of this new landscape.

3.5.1 Managed language runtime support

Flexible segment hardware should allow managed language runtimes to be simpler, faster, and more secure. A straight-forward use of segment registers as pointers and of protected calls on invocations would provide strong hardware protection of managed language objects. There is a long history of the segments-as-objects model and this case was one of the basic motivations for iAPX 432 segmentation [67] and in turn for Intel 80286 segments [17].

Java is one of the few languages that attempts to use its abstract machine model to guarantee security and it uses a combination of statically verifiable syntax and run-time checks to enforce security policy. Nevertheless, the Java Virtual Machine (JVM) is chronically vulnerable to security exploits due to the complexity of its implementation in an unsafe language, C++ [23, 66]. While a JVM could be implemented in a verifiable style of C++ [21], JVM performance is critical and inefficient constructions are unlikely to be tolerated. Specifically, pointer arithmetic and casts between data and code are common in virtual machines and are notoriously difficult to formally analyse. If C++ were extended to use capability addressing, the JVM could consistently enforce memory safety for sensitive operations without excessive overhead. Also, the JVM could generate capability addressing instructions for Java memory accesses that must be bounds-checked rather than using general-purpose instructions or risking removal of run-time enforcement. These uses of capability addressing could eliminate much of the complexity of memory safety in the JVM and of enforcement for Java executables.

CHERI capability addressing could also aid formal verification of a JVM. CHERI pointer protection manipulation is explicit in the machine code which allows formal analysis to strongly reason about use of a pointer after it has been restricted. This may allow a more highly optimised JVM to be formally verified.

3.5.2 Sandboxing

The user segment model can conveniently and efficiently support application sandboxing. Current within-address space sandboxing techniques include Google NativeClient [96], SFI [86], and XFI [55], which statically verify that all possible memory accesses are within a sandbox, thereby allowing multiple sandboxes within an address space without run-time checks. User-level hardware segmentation should make these exotic endeavours unnecessary because a few instructions can construct a sub-region of address space that can constrain all memory accesses.

Other approaches, such as DUNE [8], have attempted to use hardware virtualisation support to bring TLB-based sandboxing into user space by running the sandboxing program in kernel mode. This approach gives the application direct access to the TLB to manage its own protection but forces interactions with the operating system through the hypercall mechanism. Use of even a simple sandbox can require many pages and complex management of often implementation-dependant structures. A segmentation primitive is a far simpler and faster solution for sandboxing.

3.5.3 Fat pointers

Segments in our model can be used as general-purpose pointers with the limitation that their range cannot be extended. David Chisnall has extended the LLVM [51] compiler framework and Clang, the C frontend, to accept annotations that implement pointers as segments to ensure they are used according to programmer intent, including bounds checking and read, write, and execute property enforcement. This style of protected pointers are often called **fat pointers** [45]. This fat-pointer implementation is designed to allow simple security enhancement of existing code bases in C that often have poor security properties. Similar extensions to C include Cyclone [45], a variant of C that explicitly allows defining fat pointers that are dynamically checked, and CCured [58], which automates the same process with either static verification or run-time checks. Adding hardware support for fat pointers in the form of segment descriptors removes most of the cost of fat pointer distribution and enforcement, though retaining most of the creation cost. Segmentation also adds the possibility of enforcing some type properties in hardware such as read, write, and execute.

We have chosen fat pointers for our initial evaluation because they exercise our segment descriptor instructions in a fine-grained way and because bounds checking in software is an active area of research.

3.5.4 Unique keys

CHERI segment descriptors are unforgeable capabilities; that is, it is not possible to construct a descriptor including an address to which you do not already have access. In this way descriptors are fundamentally different from binary data currently manipulated by user space. Therefore it is possible for a language runtime or an operating system to authorise access to facilities using descriptors as unique keys. For example, if a file descriptor were represented by a capability, the operating system might be able to grant access to the file based solely on the user code presenting the correct file descriptor without any access control check. This would require that the segment descriptor point to an address that is not ordinarily available in the program's memory space.

3.6 Conclusion

CHERI capability segmentation is strong, scalable, and backwards compatible. Strong and high-performance segment descriptor protection enables a pointer capability model in user space. CHERI segment descriptors are designed to replace pointers if needed, but also can constrain accesses from general-purpose pointers in a fixed sandbox. Furthermore, the CHERI *CCALL* instruction can replace function calls with hardware-enforced protection domain crossing. These new primitives can be adopted incrementally by existing programs. Legacy executables can run unmodified in a sandbox and otherwise unsafe programs can selectively bounds-check arrays. In short, CHERI segmentation is a flexible and widely applicable hardware memory safety primitive. This dissertation will only begin to evaluate its applications.

Chapter 4

Exploratory study

This chapter will compare the CHERI capability segmentation model used for fat pointers with other memory safety mechanisms designed for transparent protection of existing C code. Recent memory safety research has emphasised transparent protection of legacy source code in C. While capability segmentation models have not been considered compatible with modern software organisation, CHERI capability segmentation is designed to fit a general-purpose fat-pointer model.

For the sake of comparison we can divide memory safety mechanisms into *address validation* schemes and *pointer safety* schemes. The TLB is a typical address validation mechanism and fat pointers are a pointer safety mechanism. Address validation ensures that the address being accessed is a valid address for the current domain. Pointer safety ensures that the current pointer being used is allowed to access the referenced address. Address validation maintains state for the address space but pointer safety maintains state for each pointer. Address validation maintains a single permissions record for an address and can therefore strongly and atomically revoke access to a memory location. On the other hand, pointer safety allows more flexibility, with each pointer allowed different bounds and permissions. For example, address validation cannot distinguish between a reference to a structure and a reference to the first element of a structure while pointer safety can offer different bounds and permissions for each. The traditional page table is an address validation technique to support strong memory protection for a multiprocess system. Fat pointers are a pointer safety technique, useful within a program and managed by the compiler.

The shortcoming of address validation versus pointer safety is especially obvious when trying to protect objects on the stack. Since the stack is a contiguous readable and writable region of memory, address validation cannot protect an array that is allocated on the stack and thus cannot prevent buffer overflows that can overwrite return addresses. For the C model, we can say that address validation can protect heap objects at allocation granularity but cannot protect stack objects.

4.1 Recent hardware proposals for memory safety

This chapter will compare various proposed approaches to memory safety in C including Mondrian Memory Protection [92], Hardbound [28], iMPX [43], and the M-Machine [13] with CHERI capability segmentation. These memory safety proposals were introduced in Section 2.2.2 but I will review them in more detail here. We will then compare their memory protection features, their hardware requirements, and their simulated performance characteristics on the Olden benchmark suite [12].

4.1.1 Mondrian memory protection

Mondrian Memory Protection (MMP) was proposed in 2002 by Witchel, Cates, and Asanovic̀ at MIT [92]. MMP is an address validation technique that uses an address-of-data keyed table which holds permissions information for every word of memory. That is, for every address provided by the processor, the MMP system would provide protection information for that address. This behaviour is quite similar to the function of a TLB, but provides word-level granularity rather than page-level and allows multiple protection domains in an address space.

Mondrian memory protection protects objects allocated by the C runtime (*e.g.* by using *malloc*) with permission regions in a multi-level protection table. While the traditional TLB model implements memory protection and memory virtualisation with one mechanism, the Mondrian memory protection authors emphasised the need for protection as distinct from virtualisation [92]. To accomplish this, Mondrian's designers proposed that the processor have a new table for memory protection in addition to the traditional TLB. This multi-level table would hold permissions for every word of addressable memory.

The Mondrian protection table is a multilevel structure similar to a page table. The most practical version of the Mondrian table encodes permissions for every word of memory by using vectors of permission bits in leaf nodes of the table, however the mid-level nodes of the table may directly encode a permission bit vector if granularity is coarse. These permission vectors devote two bits in the leaf nodes to each word for an overhead of 1/16 of used virtual memory for each protection domain for 32-bit granularity protection. These permission vectors can be cached in a Protection Look-aside Buffer (PLB) to reduce table memory accesses.

As with the TLB and other address validation mechanisms, MMP cannot distinguish between contiguous allocations with similar properties. Therefore a safe C allocator for MMP will insert forbidden guard regions between allocations to prevent accidental buffer overruns. These are similar to *guard pages* used in traditional allocators and share the weakness that large offsets can "skip over" the guard region to access other allocations.

4.1.2 Hardbound

Hardbound is a pointer safety proposal published by Devietti *et al.* in 2008 from the University of Pennsylvania [28]. Hardbound is a hardware accelerated fat-pointer scheme that is designed to be transparently compatible with existing C programs. Like MMP, Hardbound automates table look-up and protection to reduce performance cost and increase assurance. Hardbound instruments allocations in the C runtime to infer bounds for every pointer. Hardbound then stores this bounds information in a shadow address space that is at a fixed offset from the pointers themselves. This allows the format and value of pointers in data structures to be unchanged, allowing transparent sharing with code and libraries that do not use the bounds checking scheme. As a result, Hardbound's address-of-pointer keyed table must hold a segment descriptor for every pointer in memory.

To distinguish between pointers and data, Hardbound uses tagged memory with a tag cache, and this tagged memory can also be used to indicate whether a pointer is compressed. A simple implementation of Hardbound would inflate every pointer load with two more words, the base and the bound. However, if the base and current value of the pointer are the same and the length is small and well aligned, Hardbound compresses the fat pointer into the upper bits of the original pointer. The compression state of the pointer is recorded in the tag for that location so that the actual bits in memory are not exposed architecturally.

4.1.3 Intel memory protection extensions (iMPX)

Intel has recently announced Memory Protection Extensions (iMPX) to accelerate pointer safety which will ship in future Intel processors [43]. iMPX extends the x86-64 instruction set with 4 registers to store 128-bit bounds information for pointers along with instructions to load and store from these registers and to check the upper and lower bound of a pointer. Notably, Intel has included special instructions that insert and retrieve the bounds information in a table structure keyed by pointer address. These instructions can accelerate a shadow memory fat-pointer system similar to Hardbound. However, a compiler can also use the ordinary load and store bounds instructions to implement a traditional fat pointer protection scheme, storing the bounds beside the pointer in memory.

iMPX does not tag pointers in memory, but rather stores the current pointer value in the table structure so that when the bounds are loaded, the processor can verify that the bounds belong to the current value of the pointer. This gives some assurance that the program is working as expected, though a mismatch results in bounds being ignored.

While iMPX uses an address-of-pointer keyed table acceleration similar to Hardbound, protection operations are explicit and more easily bypassed than the academic proposals. This gap between proposed safety mechanisms and the implemented mechanism indicates a need for a more practical approach which maintains assurance. I argue that a user-mode capability model is simpler than MMP and Hardbound, maintains high assurance, achieves competitive performance, and provides more flexible primitives for varied protection models.

4.1.4 M-Machine

The M-Machine is a RISC capability proposal from Carter, Keckler, and Dally from MIT in 1995 [13]. The M-Machine uses a highly optimised capability fat-pointer model. The M-Machine encodes base, bound, current pointer value, and permissions into every 64-bit pointer so that every dereference is guarded. As with the previous techniques, the bound of each pointer can be inferred from calls to *malloc*, though a properly implemented compiler would be aware of smaller objects such as fields of a struct. The M-Machine only allowed the supervisor to manipulate a pointer's permissions and bounds so pointer creation requires a system call.

The M-Machine pointer structure is remarkable for its encoding efficiency. Figure 4.1 shows the bit layout. Segment length is limited to a power-of-two and the region must be aligned in memory. This allows the base to be encoded directly in the upper bits of the pointer while the current offset into the region is encoded in the lower bits. The boundary between the two is defined by the *Segment Length* field. The pipeline must ensure that the bits above this boundary are not changed by arithmetic on the pointer, but otherwise arbitrary pointer arithmetic is allowed. The M-Machine design allows many standard C pointer manipulations to work as expected, but requires that protected objects be separated by power-of-two boundaries, inflating memory allocation size.

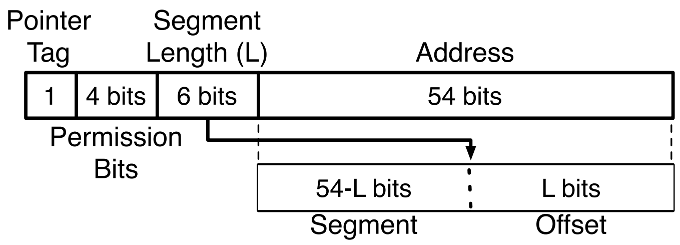


Figure 4.1: Format of the M-Machine Pointer, taken from [13]

Because the M-Machine mandates that all pointers be valid capabilities, it is not possible to transparently compile C programs which assume that integers and pointers are interchangeable. Backward compatibility and incremental adoption were not priorities for the M-Machine and as a result we did not see how its RISC capability model might be used by conventional operating systems and programs.

4.1.5 CHERI

As described in Chapter 3, CHERI accelerates fat-pointer manipulation with a dedicated register set similar to iMPX but adds tagged memory for segment descriptors to protect them from modification by general-purpose instructions, as per the M-Machine. CHERI designates the first segment register as the bounds and permissions to be applied to all general-purpose accesses. The result is a capability machine whose fat pointers are unforgeable tokens of authority but which can run legacy executables unchanged without compromising the protection model.

CHERI has two variations in this exploratory study. "CHERI" is our implemented version with large, 256-bit capabilities. "128b CHERI" uses hypothetical 128-bit descriptors which are possible without losing accuracy in the MIPS 40-bit virtual address space. A hypothetical layout is described at the end of Section 3.3.2.

4.2 Assurance and usability for C

This section will analyse the assurance and security properties of each model when used for fine-grained, automatic spatial memory safety for C. Results are summarised in Table 4.1.

4.2.1 CHERI

CHERI (presented in Chapter 3) provides a non-bypassable, fine-grained permissions check of every memory access. Tagged memory and the dedicated capability register set ensure that pointers are unforgeable even while allowing unprivileged pointer modification. By constraining general-purpose memory operations by the first capability register, CHERI is able to enforce both fine-grained permissions on select memory accesses and coarse-grained bounding of general purpose accesses, providing a clear path for incremental adoption of protection. The CHERI processor implements its pointer safety capability model, allowing the processor to distinguish between references to the same region. However this pointer safety model is in virtual memory space which is further protected by the TLB address validation model, allowing revocation to enforce temporal memory safety.

Protection mechanism	Unprivileged management	Fine-grained	Unforgeable pointers	Pointer safety	Incremental deployment
MMU	-	-	-	-	✓
Mondrian	-	✓	✓	-	✓
Hardbound	✓	✓	-	✓	-
iMPX	✓	✓	-	✓	✓
M-Machine	-	-	✓	✓	-
CHERI	✓	✓	✓	✓	✓

Table 4.1: Comparison of address-validity, pointer-safety (table-based), and pointer-safety (fat pointer based) models.

4.2.2 Mondrian memory protection

MMP provides a non-bypassable, fine-grained permissions check of every memory access. While it is not possible to forge access in the protection table, this table is guarded by supervisor mode so protection management is privileged. However, again like the TLB, MMP is not able to express variable permissions for multiple references to the same region without changing contexts and protection tables, that is, it supports address validation but not pointer safety. MMP is transparent for existing C programs and was one of the first to demonstrate that it is possible to superimpose a mandatory, fine-grained permissions check over existing C programs. This result may not be surprising since MMP is very similar to the traditional TLB mechanism but with word-level granularity. MMP can be deployed incrementally by describing permissions on very large regions and progressively revoking access to add security. Also like the TLB, MMP enables revocation of a region without garbage collecting pointers.

4.2.3 Hardbound

Hardbound also provides a non-bypassable, fine-grained bounds check on every memory access. A pointer is bounded manually with unprivileged operations, but the program can assign arbitrary bounds to a pointer, freely forging references to any memory in its virtual address space. While MMP can be completely transparent to an executable, as is the TLB, Hardbound requires compiler and runtime support to initialise pointers with bounds information, though the body of the program can use generally unmodified instructions. Because every pointer must be given valid bounds there is not a straightforward path to incremental adoption. While every pointer can be assigned generous bounds, a compiler must apply the new instructions everywhere or the program will not work. Furthermore, Hardbound does not provide protection for the program counter. Hardbound associates protection information with the location of a pointer in memory but most program counter values have no associated memory location. Nevertheless, Hardbound is a pointer safety scheme and does provide independent bounds for every pointer, allowing the compiler to distinguish, for example, between references to the stack and references to an array allocated on the stack.

4.2.4 Intel memory protection extensions (iMPX)

Intel's iMPX accelerates fine-grained pointer safety and supports incremental deployment with protection being manual and optional. Indeed, iMPX programs will work as expected if none of the iMPX instructions are executed. Bounds manipulation instructions in iMPX are unprivileged but unfortunately bounds data is easily forgeable. The program can generate arbitrary bounds in bounds registers or is free to modify bounds in memory. The iMPX model is flexible and can either accelerate an out-of-band table model for data-structure compatibility or a standard fat-pointer model for performance. iMPX also does not constrain the program counter during normal execution.

4.2.5 M-Machine

The M-Machine provides pointer safety at low cost with unforgeable pointers. However, the requirement to align all memory regions at powers-of-two implies object padding that prevents precise bounds checking of memory accesses. Modifying pointers are a privileged operation on the implementation of the M-Machine, though its model would allow unprivileged pointer modification instructions in the style of CHERI. Also, due to its strict new pointer model, the M-Machine requires an entire program to be recompiled and likely adapted in order to run, thus incremental adoption is not possible.

4.3 Memory safety limit study results

Protection schemes described in the previous section vary widely in functionality and performance. To better understand performance trade-offs, I performed a simulation-based limit study on pointer-intensive benchmarks, measuring 4 properties: instruction count, memory traffic, page allocation count, and memory-access count. I have adapted each to fit a 64-bit MIPS pointer model so that all of the models could be directly compared. The details of these adaptations are in Appendix A. These adaptions were mainly to extend Hardbound and Mondrian memory protection to protect 64-bit pointers and were advantageous to both.

Collectively, these models span a broad range of fat-pointer and table-based address validity and pointer safety techniques. I used the Olden benchmarks [12], a suite developed for distributed shared-memory research that has become popular in bounds-checking research due to its focus on pointer-based data structures (e.g., Hardbound). The benchmarks use a range of data structures, memory footprints, and workloads to exercise various pointer access patterns and densities.

In order to model performance overheads for each memory protection approach, I recorded complete instruction traces of the Olden benchmarks running on our baseline MIPS implementation in hardware. I then modelled overheads that would be incurred for various protection models. This model took into account all operations relevant to memory safety: C memory-management functions such as `malloc()` and `free()`, stack growth, and all memory loads and stores.

Approaches that track objects, which include CHERI, have some cost for object creation to generate object bounds. Object tracking approaches had three sources of object creation: elf regions, malloc calls, and stack allocation (in 64 KiB chunks). As the benchmarks were statically linked, these regions accounted for all memory use in these benchmarks.

Furthermore each approach can incur an overhead on every load and store of a pointer. Hardbound, for example, may retrieve or insert bounds into a data structure if it is uncompressable. I model the additional instructions and memory accesses for pointer loads and stores for each approach.

Lastly each approach can incur an overhead when a pointer is dereferenced. For example a software approach will require additional instructions to check the bounds and Mondrian Memory Protection may have a protection table miss. These costs for each pointer dereference were also added to the overhead of each approach.

Results are displayed using box plots of the results from 8 of the 9 Olden benchmarks hosted by Trimaran project [83]. The "Veronoi" benchmark, which was originally included in the Olden set, was not present in the set from Trimaran and was not included. The Power benchmark was not used because its minimum runtime was much longer than the others and it was not possible to obtain a trace from hardware and analyse it in a reasonable amount of time. The box plots mark the median, the first and third quartiles, and the most extreme results to display the range of overheads expected in pointer intensive workloads. All performance results are presented as normalised overhead against the baseline with no protection. The 8 benchmarks are listed in Table 4.2.

Name	**Description**
BH	Barnes and Hut force computation (OcTree)
Bisort	Forward and backward sort of integers using disjoint bitonic sequences which are (binary-tree)
Em3D	Electromagnetic wave propagation (linked list)
Health	Health-care simulation (double-linked list)
Mst	Minimum spanning tree of a graph (array of linked lists)
Perimeter	Perimeters of regions in images (quad-tree)
Power	Power pricing system optimisation (N-way tree, linked lists)
TreeAdd	Recursive sum of values (binary-tree)
Tsp	Traveling salesman (balanced binary-tree)

Table 4.2: List of Olden Benchmarks

Instruction count — The dynamic instruction count increase due to added allocation and bounds-checking instructions gives an indication of the load required by instruction issue and reorder logic for a pipeline as well as instruction cache footprint. While it is possible for an instruction heavy approach, such as iMPX, to perform similarly to an approach with implied checks such as CHERI, this would require a power hungry multiple issue pipeline with reorder logic.

Memory traffic — Total memory traffic (in bytes) affects power and gives some indication of cache load. Movement of data is expensive for power consumption, especially if there is an increase in DRAM accesses. If memory traffic is increased but there is a high amount of reuse, or if locality is good, this metric may not directly correlate to power or performance problems.

Memory accesses — The count of individual memory accesses, which ignores the size of accesses, measures the load on the MMU and on cache coherence mechanisms. While *memory traffic* measures the total number of bytes moved by the pipeline, *memory* *accesses* gives the number of individual memory operations. A wide memory access, such as loading a 256-bit capability, will be treated as a single *memory access* by the TLB and by the cache subsystem, though four independent 64-bit accesses will generate an identical amount of *memory traffic*.

Page allocation — The total number of 4 KiB pages used by the program indicates physical memory footprint and TLB pressure and should also give some indication of the locality of the data being accessed. Models with high page overhead may have scattered data access patterns, to access table structures for example, which would adversely affect cache locality and prefetching algorithms. Large pages would improve TLB pressure for the in-band fat-pointer approaches, such as CHERI, but out-of-band table approaches such as MMP, Hardbound, and MPX are likely to require multiple allocations regardless of data-set size.

4.3.1 Instruction count

Figure 4.2 gives both optimistic and pessimistic instruction counts for each model running the Olden benchmarks in Table 4.2. A trace of each benchmark was recorded running on CHERI hardware and a script counted additional instructions required in each benchmark for each memory safety model. Both the optimistic and pessimistic cases include extra instructions for allocation, however the optimistic figure records a bounds check for every load of a pointer, and the pessimistic figure records a bounds check for every dereference. This simulates a memory safe compiler which will might statically analyse memory accesses to see if a bounds check can be lifted out of a loop. In the best case, the compiler will know the greatest extent that will be accessed on a particular load of a pointer and will check the bound once but in the worst case the bounds will have to be checked on every dereference. The Y axis on Figure 4.2 is logarithmic to expose performance differences in the very low overhead cases.

Figure 4.2 includes two types of instruction overhead. The first is pointer creation cost, and the second is pointer use cost. Mondrian, Hardbound, the M-Machine, and CHERI have a pointer creation cost, but no per-reference instruction overhead since bounds checks are implicit. Mondrian and the M-Machine require system calls to create bounded pointers, so their overhead is 4.5% and 1.5% respectively and does not vary between the optimistic and the pessimistic case since they do not incur any per-reference instruction cost.

iMPX and the software fat-pointer case incur a per-reference cost but do not require a system call for pointer creation, so their cost is almost entirely due to the per-reference overhead. For this reason the optimistic and pessimistic overheads vary from 10% to 35%. These results roughly match experimental results for software bounds checking [29, 20, 50].

In the instruction count metric CHERI incurs 0.05% overhead for both cases. From this perspective, CHERI is especially suited to systems with limited instruction issue width or which want to conserve scheduler resources.

4.3.2 Memory traffic

Figure 4.3 gives the total number of bytes accessed during the Olden benchmark runs for each model. As expected, the table walk in iMPX for every pointer load and store requires significantly more memory accesses than any other scheme, especially since bounds information is inflated to 192 bytes to store a redundant copy of the pointer value. The iMPX fat pointer model does better with 128-bit bounds information and without the

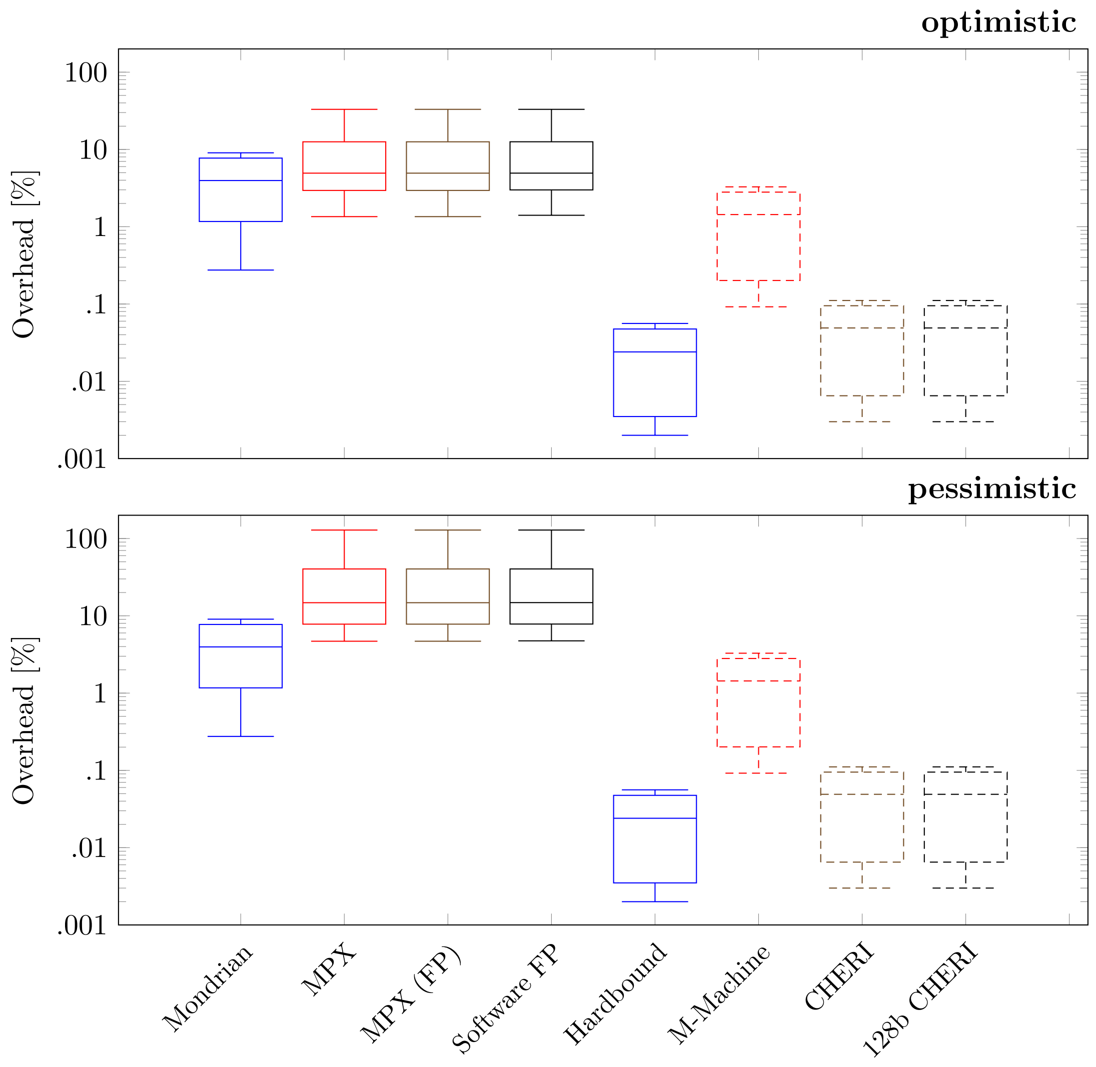


Figure 4.2: Total instructions (count)

extra 64-bit pointer in the table lookup. Furthermore, the bound information for the iMPX fat pointer model will be in-line with the data and exhibit better locality.

Mondrian incurs its overhead not from the loading and storing of pointers but from table lookups on memory addresses that are not cached in the protection table and from system calls for region creation. The total traffic overhead for Mondrian varies widely from negligible in many cases (see the very low average for the MMP case) to higher than software fat pointers for more pathological cases.

The M-Machine has no memory traffic overhead since it fits all protection information into the original 64-bits of the pointer at the cost of legacy compatibility.

CHERI generates more traffic on pointer-heavy benchmarks due to its larger pointers size, however the proposed 128-bit variant is competitive with most of the other models. A compressed 64-bit descriptor format should produce very similar results to the M-Machine for these benchmarks which typically have small segment sizes.

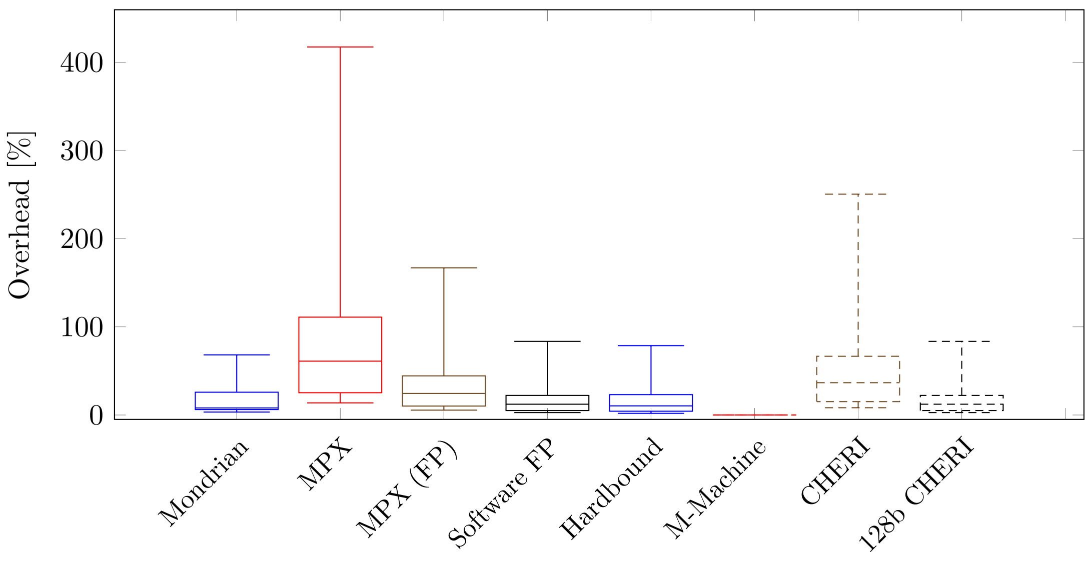


Figure 4.3: Memory I/O (bytes)

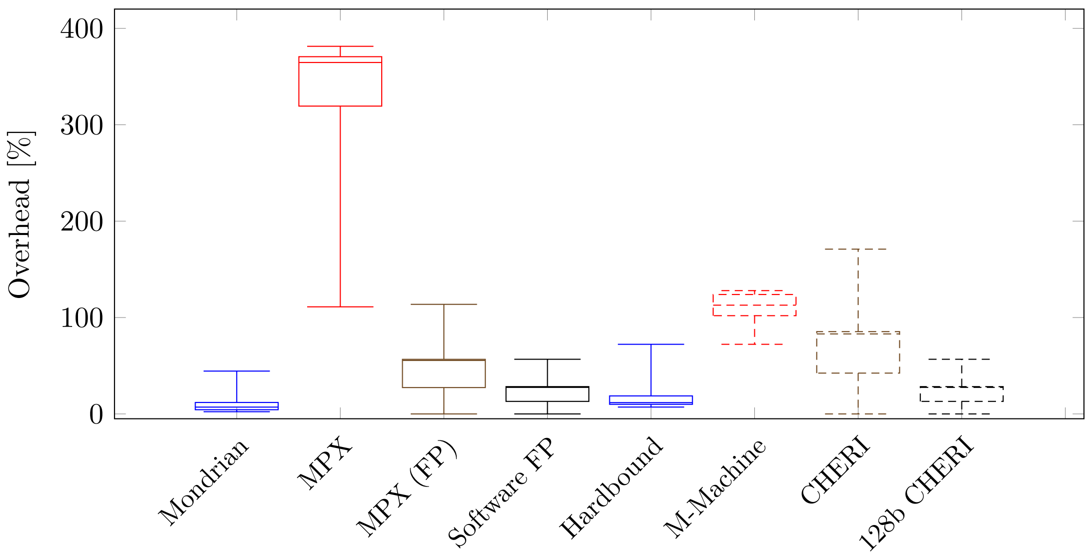


Figure 4.4: Virtual memory footprint overhead (4 KiB pages)

4.3.3 Page allocation

Figure 4.4 gives the 4 KiB page footprint of the memory safety models for the Olden benchmarks.

The iMPX case has the highest page overhead. The iMPX table contains more than 4 pages for each page of memory containing pointers, maintaining 256 bits in the leaf nodes for each 64 bit memory location. The Hardbound model uses a simpler table structure and 128-bits of metadata per non-compressable pointer but must first check an additional tag table. The Olden benchmarks generally use many small objects, so most pointers are compressed in these cases and the page overhead for Hardbound is fairly low as a result with only one result approaching 100% overhead.

Interestingly, the M-Machine behaves poorly by the page metric as every allocation size must be padded to a power of two.

Mondrian uses the least extra memory allocation due to its compact table structure which stores only one 8-byte word for each 256-bytes of allocation (not counting the mid-level nodes). This natural overhead of less than 5% is reflected in the results, though there are some outliers due to the granularity of page size (one allocation in a region of memory requires an entire page of table allocation).

CHERI and the other simple fat-pointer approaches have predictable page overhead as the additional pointer data is packed into existing pages with larger structures only occasionally spilling onto additional pages. While fat pointer overheads can be significant, especially for pointer-based data structions such as in many of the Olden benchmarks, the 128-bit CHERI case has a 24% page overhead on average with only Mondrian consuming less with an average of 12.3%.

4.3.4 Memory references

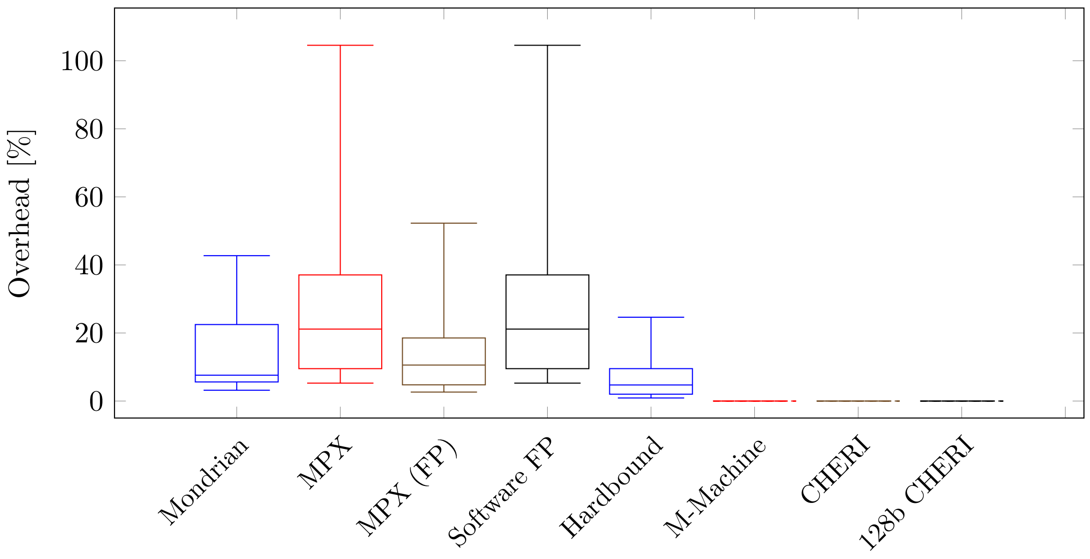


Figure 4.5: Memory references (count)

Figure 4.5 gives the memory-access count overhead for each memory safety model. This table presents the number of individual load and store requests separately from the number of bytes stored in Figure 4.3, as many structures in the CPU scale with the number of independent transactions rather than with the total size of transactions.

The iMPX table case and software fat pointers have identical memory-access overhead. iMPX requires two loads or stores for each table lookup associated with each pointer load or store. Software fat pointers also require two additional words for each pointer case, though in the fat pointer case the two words are the base and bound and are in-line.

The iMPX fat pointer model has half the overhead of both iMPX and software fat pointers because it moves both the base and bound in a single transaction from the bounds register.

The Mondrian model typically has a low overhead, but this can grow large in cases that overflow the protection table since it uses a three-stage table walk.

Hardbound does well due to most pointers being compressible. The hardbound case gives some indication of the benefit that iMPX would gain from a compression scheme to reduce table lookups.

	MMP	MPX	MPX (FP)	Soft FP	HBound	M-Mn	CHERI	128b CHERI
Inst # Opt.	4.44	10.04	10.04	10.07	**0.03**	1.55	0.05	0.05
Inst # Pess.	4.44	34.94	34.94	34.97	**0.03**	1.55	0.05	0.05
Page Alloc.	**12.33**	319.58	48.16	23.84	21.20	109.23	72.78	23.84
Mem. Acc.	15.67	32.02	16.01	32.02	7.62	**0.00**	**0.00**	**0.00**
Mem. Traff.	20.50	109.23	43.69	21.85	20.80	**0.00**	65.54	21.85
Average	11.48	101.16	30.57	24.55	9.93	11.48	27.68	**9.16**

Table 4.3: Comparison of average overheads for Olden Benchmarks.

The CHERI cases and the M-Machine do well on this metric as they have no additional memory transactions as a result of the fat pointer model. This is because they handle the more complex pointer structure as a single unit, as is done in the standard pointer case. That is, every pointer memory operation in the standard case is replaced by a single capability memory operation in the protected case.

4.3.5 Summary of limit study results

Despite being simpler than proposed academic approaches, the CHERI model proves competitive in major metrics (especially in its 128-bit variant), indicating that there are no major performance limitations to the user-level capability segmentation approach. As seen in Table 4.3, CHERI is rarely has the best averages in a category, but is never segnificantly behind the second-best performer in any benchmark. As a result, the 128-bit CHERI attains the over-all best overage overhead for these benchmarks, indicating a balanced design. The main drawback to CHERI as designed is the large capability data structure, and improvements are likely to come from enabling compressed capability formats, though these will need to be traded off with compiler and hardware simplicity.

4.4 Hardware requirements for memory protection schemes

Memory protection schemes also have several classes of hardware requirement for memory protection. This section broadly describes each hardware structure with its cost and then enumerates the requirements for each model of memory protection. The results are summarised in Table 4.4.

4.4.1 New Hardware Components

Below are the classes of additional hardware components required by various memory protection schemes.

Tagged Memory — Several schemes use tagged memory, that is, an extra, implicit bit of data for each memory word that could hold a pointer or a segment descriptor to indicate whether or not it is a valid reference. Tagged memory in this context enforces a simple type safety in hardware with two types, pointer and data. These pointer tags can

Hardware extension	Tagged memory	Table walker	Explicit register file	Side-car register file	Look-aside buffer	Memory-path arithmetic
TLB	-	✓or -	-	-	✓	-
Mondrian	-	✓	-	✓	✓	-
Hardbound	✓	✓	-	✓	-	-
iMPX	-	✓	✓	-	-	-
M-Machine	✓	-	-	-	-	-
CHERI	✓	-	✓	-	-	✓

Table 4.4: Comparison of additional hardware required by memory protection models.

accompany standard tags in the cache hierarchy. A tag controller after the last-level cache can retrieve and store the tags from a table in memory. Tags may be associated only with virtual memory, allowing differing interpretation of stored pointers in different address spaces, or may be associated with physical memory, reducing tag memory to a single table mapping for the entire physical address space.

Memory Table Walker — Several schemes use tables in memory that must be walked by a hardware state machine. These tables both consume sparse out-of-band memory and add complexity to the memory subsystem by adding an additional source of memory requests. Hardware table walkers are most often used for TLBs but some architectures, e.g. MIPS, forgo this feature and walk these tables in software, indicating that hardware table walkers may have a noticeable cost in pipeline complexity.

Register File — Some memory protection schemes require a new register file to hold bounds information. Additional register files may be explicitly referenced or may be implicitly referenced as *side-car registers*. A side-car register file is a set of registers that are each directly associated with a general-purpose register. Side-car registers are quite expensive because every register read requires an additional read to check if the value is a pointer and every register write implies an additional write to propagate bounds and protection information.

Look-aside Buffer — Like the TLB, memory protection schemes often require an associative look-aside buffer, which is expensive for power and area. An associative table is searched on every memory access to see if the particular memory access is in any valid region.

Memory Pipeline Arithmetic Operations — Memory protection schemes can require extra arithmetic operations in the memory pipeline. This can have a significant effect on integration with existing designs if the operation affects latency or is limited by throughput; that is, if the operation is in the critical path or if the new operation is common enough to saturate available resources.

4.4.2 CHERI

CHERI requires tagged memory, a new register file, and arithmetic operations in the memory pipeline. CHERI uses one tag bit for every 256-bit line of physical memory, a fixed memory overhead of 0.4% for the system. Tagging physical memory not only reduces the amount of tag memory for the system, but allows the operating system to page tags out of memory with physical pages without walking translation tables.

A CHERI segment is accessed on every memory operation so a new register file with independent ports accessed in parallel to the general-purpose register file is required. Because CHERI uses the segment base for address calculation, one add operation is inserted into the path of virtual address calculation, though the bounds check is performed in parallel with the memory request. The high-level effects of adding capability support to the CHERI processor are discussed in Section 5.5.3.

4.4.3 Mondrian Memory Protection (MMP)

MMP requires a complex table walker as well as a look-aside buffer and proposes to use a side-car register file. Existing protection look-aside buffers are only implemented in designs without a TLB, indicating that the expense of an associative search is not likely to be afforded twice, once for fine-grained protection and again for coarse-grained protection and for translation. This would indicate that MMP is most likely to be used in embedded systems to protect a flat, physical address space. However, none of the existing embedded protection look-aside buffer implementations include a hardware table walker as proposed by MMP. This would indicate that a complex hardware table walker to accelerate the use of a large number of protection domains will be considered a difficult, expensive feature for this segment, especially given the rigid use model it imposes. The MMP vector table model used in our comparison expresses all protection domains as a set of power-of-two regions, eliminating additional arithmetic from the memory path so that only the lookup is required, similar to the TLB. Furthermore, MMP does not contribute to the virtual address as it does not provide translation, and thus the MMP table search and permissions check can be performed in parallel with cache lookup. While this may eliminate delay cost for the common case of a table hit, table misses will generate a string of memory accesses that will delay instruction retirement and stall the pipeline. MMP also proposes the use of *side-car registers* to hold permissions for a segment of memory associated with a pointer in a general-purpose register to reduce the use of the associative table. While side-car registers eliminate the need to perform an associative search on every memory reference, they add the requirement to read and write an additional register file on every general-purpose register read and write, doubling the size of the data-path in the pipeline in the optimistic case that bounds information is the same size as pointers.

4.4.4 Hardbound

Hardbound requires tagged memory, a table in memory, a side-car register file, and additional arithmetic in the memory pipeline, though not in the critical path. Hardbound requires two tables in memory, one for tags and one for bounds data. Hardbound proposed tagging virtual memory, which would require a separate tag space for each process and a potentially multilevel table structure, since all of virtual memory must be representable. Hardbound requires a tag for every potential pointer in memory, so a 64-bit architecture with aligned pointers requires less than a 2% memory area overhead for all structures with pointers.

The bounds table in Hardbound is simply a fixed offset from the pointer memory location so a table walker is not required, though there is a 200% memory overhead and the span of used memory must be less than one third of the virtual memory space.

Hardbound also requires a side-car register file to hold bounds. This register file is populated from the bounds table (or from compressed bounds) on every pointer load and is accessed on every use of a pointer, either for arithmetic manipulation or for dereference. Hardbound also adds a bounds check to every data memory access, though these are not in the path of virtual address calculation. These permissions checks will never require memory accesses, and so will be of predictable latency and will never delay writeback.

4.4.5 iMPX

iMPX provides a hardware table walker and a new, explicitly indexed register file. The iMPX table walker is invoked by a dedicated instruction and walks a two-level table, giving software greater flexibility in memory allocation than with Hardbound, though requiring an extra memory reference on every look-up. This table allocates four words for each potential pointer in memory (one for pointer value as a sort of tag, two for bounds, and one reserved) implying an overhead of 400% for structures containing pointers when the iMPX table is in use. This approach essentially combines the tag table and the bounds table and depends on the compiler to load the pointer metadata, which will be ignored if loaded for a pointer value different from that for which it was stored to ensure integrity in place of tagging. The new register file is also accessed explicitly and is not automatically accessed in the common case, so independent ports from the general-purpose register file are not implied, especially since the check of the upper bound is separated from the check of the lower bound. This allows a trivial implementation of iMPX without adding a register file, but by extending the general-purpose register file with new registers mapped to bounds registers in the ISA. Furthermore, the optional independent bounds check instructions allow the general-purpose pipeline to perform the permissions check. Therefore, besides the table walking instruction, iMPX requires only decoder changes for its implementation at a compromise to assurance and performance.

4.4.6 M-Machine

The M-Machine processor requires tagged memory as well as simple masked equality checking in the pointer arithmetic path. A tag is required for every word of memory, an overhead of less than 2% for 64-bit pointers. The M-Machine tags pointers in general-purpose registers but does not use an additional register file. The M-Machine requires a masked equality check when addresses are manipulated by arithmetic to ensure that a pointer does not overflow into another region, but does not add arithmetic to the memory-access pipeline. This means that the tag on a register adds an exception case to arithmetic using that register. This can complicate the pipeline since the operation depends not only on the instruction, but on contents of the register file which is available later in the pipeline.

4.4.7 Summary of hardware requirements

CHERI requires tagged memory, an explicit register file, and an add in the path of address calculation. These requirements fit conveniently into a conventional RISC implementation. The register file and the additional offset are straightforward and tagged memory has also proven to be simple in implementation when merged into the current cache structure, as described in Section 5.5.2.

In comparison, Mondrian Memory Protection and Hardbound both require table walkers, with Mondrian Memory Protection requiring an additional look-aside buffer and Hardbound requiring a side-car register file, all of which are expensive features implemented to make protection implicit. CHERI makes a practical compromise toward more explicit protection to enable both simpler hardware and more flexible software utility.

Chapter 5

CHERI implementation

Memory-safety processor research, and processor architecture in general, has suffered from a lack of accessible implementations. Science works well if experiments can be easily reproduced and extended. However, the few processor designs for memory safety that have been fully implemented in hardware are not open source. Recent memory-safety research has used simulators to estimate performance which should be easier to distribute. However, Mondrian Memory Protection and Hardbound, the two recent major works in simulation, do not appear to be open source or available for download, making it unclear how the work could be reproduced. One might assume that these were not developed to a distributable standard and are therefore unsuitable for further extension and experimentation. Memory protection is a service used by a great body of software and a thorough conclusion about the value of a hardware primitive cannot be reached without a stable and useful development environment to explore the utility of these primitives for operating systems, compilers, and applications.

In contrast, CHERI has been designed for a Field Programmable Gate Array (FPGA) implementation to support full operating system and application development. FPGAs offer the ability to reproduce processor designs inexpensively. Due to working closely with compiler and operating system consumers, I have built CHERI to a high level of quality using a robust language and an extensive regression test suite[1]. This real implementation has prevented us from glossing over difficult implementation details during the course of our design as can be tempting when working in simulation. Nevertheless, CHERI is easily extensible and open-source to allow not only reproduction of our experiments, but easy exploration of alternative designs. The extensibility of CHERI has enabled multiple revisions of the instruction set, compiler, and operating system to arrive at an optimal design.

I will begin this chapter by motivating our choice of development language, Bluespec SystemVerilog (BSV), for an open-source research hardware project, detail the architecture of BERI itself, and then describe our simulation infrastructure, testing strategy, and FreeBSD operating system bring up. Finally I will describe the architecture of the capability coprocessor.

[1]The regression test suite was developed primarily by Robert Watson and Michael Roe with assistance from many others and is described in Section 5.4

5.1 BSV: A language for extensible design

Large open-source hardware projects have not been encouraged to develop because traditional hardware description languages encourage fragile, complex designs. As a result, large hardware designs have only been successfully developed by professional teams of developers with a high level of discipline. CHERI uses a new hardware description language, Bluespec SystemVerilog (BSV), which automates control-flow and enforces type safety in order to make the design easier to extend without breakage to enable a community supported project [68, 69, 38].

Bluespec SystemVerilog is a hardware description language developed from work at Massachusetts Institute of Technology [11] that allows designers to build robust and extensible hardware designs. BSV does this by bringing a rich and strict type system to hardware design which includes interfaces for correct flow control. BSV is a "term rewriting system" [39, 38] and a BSV design is structured as a collection of state that is updated by a set of rules. While this is a common design pattern in traditional hardware description languages, strictly limiting the designer to the rule structure enforces regularity of design as well as abstracting away common control signals.

A simple FIFO example demonstrates the value of BSV syntax and interfaces to illustrate some basic benefits of the language for open-source, easily-extensible hardware. The left column of Figure 5.1 is an implementation of a single element FIFO in BSV, and the right side is a similar FIFO in SystemVerilog. The BSV module exposes three operations, *enq*, *deq*, and *first*. The SystemVerilog module exposes seven signals which may be set or read independently, though only certain combinations and sequences of actions will result in a coherent result. For this reason well-documented SystemVerilog modules are accompanied by a set of timing diagrams to define their interface. This complexity problem grows significantly in large SystemVerilog designs with interoperating modules where each must be understood in order to modify the design in significant ways.

Figure 5.2 gives a similar test bench in each language which exercises the respective modules. Again it should be noted that not only is the code much shorter in the BSV case, but it is easier to understand. Crucially, the flow control signals that might incite incorrect operation are not exposed in the BSV case and correct operation is default and implicit. In contrast, the SystemVerilog case exposes all flow control signals rendering the system fragile and susceptible to misuse.

Both the BSV and the Verilog implementations of the FIFO use a *Pkt_t* type. In the Verilog version this type will be a simple pseudonym for a bit vector, but in the BSV case this type will be distinguished from other types of the same width. This allows the designer to embed his intended use for the FIFO in the design and prevent unintended use without explicit casts.

The BSV rule structure and the implicit clock and reset signals in BSV eliminate other classes of error. A full description of the language is available elsewhere, but I consider the guarded interface structure as the most significant language feature for assisting the manageability of large scale projects.

I believe that the factors mentioned here are key reasons why open-source hardware projects have not seen much success. Successful contribution to an existing system requires an inordinate level of understanding of the system such that it is often easier to reimplement a complex system than to understand an equivalent system built by someone else. I have found that a code base in Bluespec SystemVerilog is relatively approachable and extensible and have had many successful undergraduate, Masters, and Ph.D. level

Bluespec

```
interface FIFO#(type Pkt_t);
 method Action enq(Pkt_t packet);
 method Action deq;
 method Pkt_t first;
endinterface



module mkFIFO(FIFO#(Pkt_t));
 Reg#(Pkt_t) fifomem <- mkReg(0);
 Reg#(Bool) full <- mkReg(False);

 method Action enq(Pkt_t packet)
                          if(!full);
   fifomem <= packet;
   full <= True;
 endmethod

 method Action deq if(full);
   full <= False;
 endmethod

 method Pkt_t first if(full);
   return fifomem;
 endmethod
endmodule
```

SystemVerilog

```
interface FIFO(
 input clk,
 input rstp,
 input Pkt_t din,
 input readp,
 input writep,
 output Pkt_t dout,
 output logic fullp);
endinterface

module fifo(FIFO ifc);
 Pkt_t fifomem;
 assign ifc.dout = fifomem;
 always_ff @(posedge ifc.clk)
   begin
    if(ifc.rstp)
     begin
      ifc.fullp <= 1'b0;
      fifomem <= 0;
     end
    else
     begin
      if(ifc.writep && !ifc.fullp)
       begin
        fifomem <= ifc.din;
        ifc.fullp <= 1'b1;
       end
      if(ifc.readp && ifc.fullp)
       ifc.fullp <= 1'b0;
     end
   end
endmodule
```

Figure 5.1: FIFO in BSV versus SystemVerilog

Bluespec

```
module mkFifoBench();
 Reg#(Pkt_t) counter <- mkRegU;
 FIFO#(Pkt_t) fifo <-
              mkFIFO#(Bit#(32));

 rule enq;
  fifo.enq(counter);
  counter <= counter + 1;
 endrule

 rule deq;
  $display(fifo.first);
  fifo.deq();
 endrule
endmodule
```

SystemVerilog

```
typedef reg [31:0] Pkt_t;

module fifoBench (
 input clk,
 input reset
);
 Pkt_t counter;
 reg readp;
 reg writep;
 wire dout;
 wire fullp;
 FIFO ifc(clk, reset, counter,
         readp, writep, dout,
         fullp);

 fifo myfifo(.ifc(ifc));
 always_ff @(posedge clk)
  if (reset)
   counter <= 0;
  else
   begin
    if (!ifc.fullp)
     begin
      readp <= 1'b0;
      writep <= 1'b1;
      counter <= counter + 1;
     end
    else
     begin
      writep <= 1'b0;
      readp <= 1'b1;
      $display(dout);
     end
   end
endmodule
```

Figure 5.2: FIFO testbench in BSV versus Verilog

projects significantly extend the BERI processor. Projects include a floating point unit, coherent caches, new memory addressing modes, and improved branch predictors. I hope this success will continue more broadly with BERI as a universally available open-source project.

5.2 BERI, a processor in BSV

This section describes the general-purpose 64-bit MIPS core upon which CHERI is based. I have named this base processor BERI (Bluespec Extensible RISC Implementation) and it has been packaged as an open-source project which is available with a Terasic DE4 tablet design. I will give an overview of the BERI design to demonstrate its completeness and maturity as a processor implementation. BERI is currently an in-order core with a 6-stage pipeline which implements the 64-bit MIPS instruction set used in the classic MIPS R4000 [57] but does not implement its 32-bit addressing modes. Floating point support, developed by Colin Rothwell, is fairly mature and executes compiled programs under FreeBSD. Achievable clock speed is above 100 MHz on the Altera Stratix IV and average cycles per instruction is close to 1.5 when booting the FreeBSD operating system. In summary, the high-level design and performance of BERI is comparable to the MIPS R4000 design of 1991, though the design tends toward extensibility and clarity over efficiency in the micro-architecture.

5.2.1 Starting point: Greg Chadwick's multi-threaded user-mode MIPS64

The BERI project started with an early version of the MAMBA multi-threaded usermode 64-bit core in BSV developed by Greg Chadwick for use in many-core research [16, 15]. Chadwick's core was a standard 5-stage pipeline with an extra stage for the context FIFO. This core executed 8 independent threads with associated register files using round-robin scheduling. I extended this implementation into the BERI processor by adding register renaming for full pipelining of a single thread, branch prediction, multiply and divide, level 1 and level 2 caches, a TLB, and all system register support necessary to boot a general-purpose operating system. I removed multi-threading in pursuit of optimal single-threaded performance but hope to reintroduce it in the future.

5.2.2 The BERI pipeline

The BERI pipeline in its current state is 6 stages long. Figure 5.3 gives a detailed high-level overview of its structure.

1. **Instruction Fetch**: submit program counter to memory system
2. **Scheduler**: initial decode for register file and branch predictor
3. **Decode**: fully define instruction behaviour
4. **Execute**: perform arithmetic or assignment
5. **Memory Access**: submit any memory operation

6. **Writeback**: potentially commit updates to architectural state

Each stage of the main BERI pipeline has a **FIFO#(ControlTokenT)** interface. This means that a single data type is used for all stages of the pipeline from *Instruction Fetch* to *Writeback* and that each stage has implicit flow control. If one stage of the pipeline cannot accept a new **ControlTokenT**, for example if *Memory Access* is waiting for a read, the pipeline will naturally stall. I carefully designed the register file to provide register forwarding with full throughput in the presence of arbitrary buffering in the pipeline. As a result a new pipeline stage can be inserted into the pipeline without breaking functionality and with graceful performance degradation. Indeed, in the course of development, students often insert extra stages unintentionally and rarely notice before optimising performance.

5.2.3 Branch prediction (mkBranch)

The BERI branch predictor uses both local and global history to predict branches, as well as a call stack. As seen in Figure 5.3, the BERI branch predictor has three key interfaces that allow correct operation for an arbitrary pipeline. Branch prediction begins with the second interface, the *putTarget* interface, which takes a decoded instruction from the pipeline. This interface generates two tokens, as illustrated in Figure 5.4. Both tokens contain the predicted PC, the first for *Instruction Fetch* and the second to confirm the prediction in *Writeback*. These methods exercise several memories to store prediction history, previous target values, and a call stack.

I produced a simple version of the branch predictor for a masters' level course. Students were able to modify this branch predictor to improve hit rates for a software routine and several students have chosen to do more extensive modifications of the branch predictor as a project. The clear structure of the BSV language has enabled extensive modification by designers with only a cursory understanding of the original design.

5.2.4 Register File (mkMIPSRegFile)

I have designed the BERI register file to enable full register forwarding while exposing a simple interface to the pipeline. The BERI register file, illustrated in Figure 5.5, receives a token in the *Scheduler* stage with all the register numbers that will be read and written. In *Execute* the register file both delivers the operand registers and receives a speculative write. In *Writeback* the pipeline delivers the final write to the register file. This interface is sufficient to implement dependency resolution logic internally without exposing complexity to the main pipeline.

The BERI register file implements register renaming with a result buffer internally. During the *Decode* stage a rule reads and updates a data structure that holds state for each rename register in the result buffer. Every instruction that writes to the register file is allocated a rename register whose value is used in place of the value in the architectural register file as long as its epoch matches that of the instruction. The register file itself limits the number of outstanding instructions in the later stages of the pipeline to avoid overflowing the rename registers. The number of rename registers is parameterisable and is currently 4 to allow full throughput of the pipeline.

The BERI forwarding register file allows implementation of new instructions by modifying *Decode* and *Execute* with full performance without exposing the complexity of dependency resolution. For example, Nadesh Ramanathan (a Masters student) added

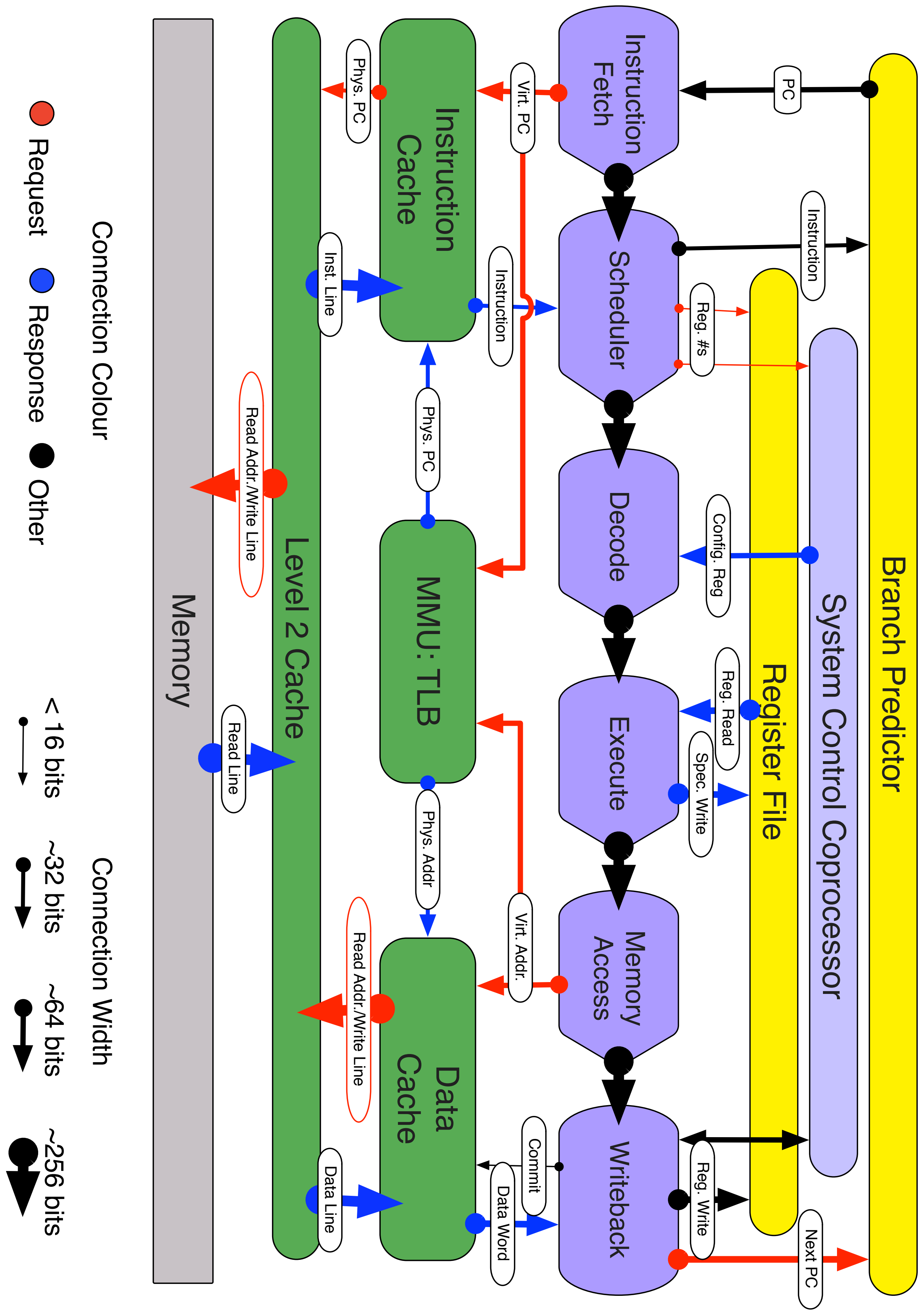


Figure 5.3: BERI Pipeline with Caches

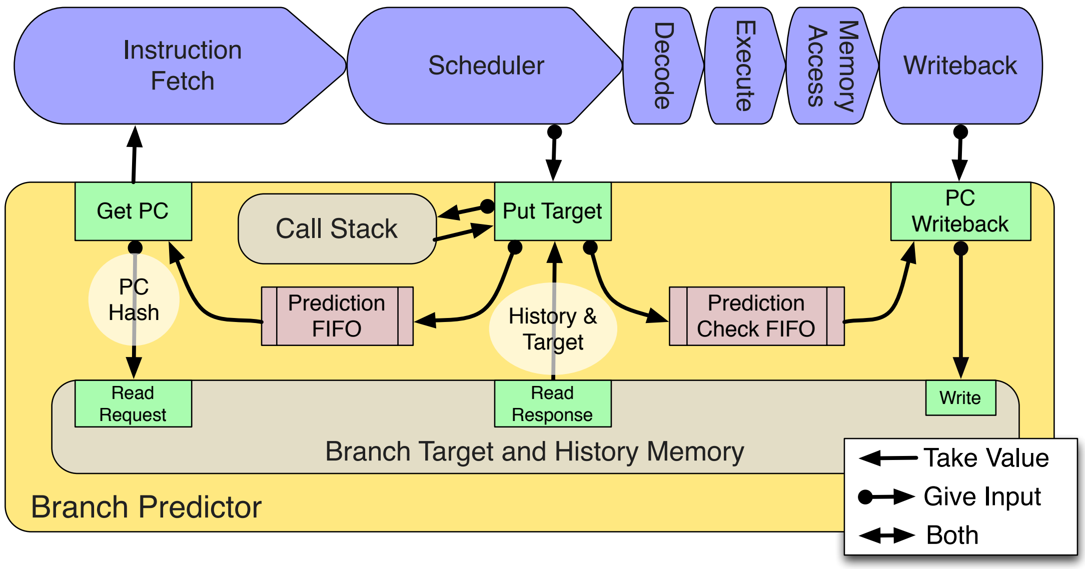


Figure 5.4: BERI branch predictor

working, full performance instructions to BERI for a new addressing mode in less than 3 days despite complete unfamiliarity with the design at the start.

5.2.5 Caches

I have designed three caches for the BERI pipeline: an instruction level 1 cache (mkI-Cache), a data level 1 cache (mkDCache), and a shared level 2 cache (mkL2Cache). The arrangement of these caches is depicted in the bottom half of Figure 5.3. All levels of cache are direct-mapped because memory is plentiful in the FPGA but combinational logic is not. I tried two-way and four-way set associativity in the level 2 and two-way in the level 1, both with way prediction. Timing was a problem in all cases and nearly the same hit-rate was achieved by simply quadrupling the size of the direct-mapped cache. The level 1 caches are both currently 16 KiB in size, are write-through, virtually indexed and physically tagged, and respond on a hit in one cycle. The data cache uses a full 32-byte memory width internally to facilitate wide writes from coprocessors such as the capability coprocessor. The level 2 cache is 64 KiB in size and services the level 1 caches through a request merge module also using a full 32-byte wide interface.

5.2.6 Translation look-aside buffer (mkTLB)

The BERI translation lookaside buffer structure is unique. Large TLBs are usually set associative in modern designs, but the common MIPS specification only defines a fully associative table [37]. The modern MIPS specification defines both a variable-page-sized table (which is fully associative) and a fixed page size table (which is set associative). However FreeBSD, our target operating system, did not yet support this arrangement. I noticed when debugging FreeBSD TLB operations that the operating system never used an indexed write above the WIRED register[2] (which was usually 1) without first

[2]The WIRED register specifies the TLB entry below which entries should not be arbitrarily replaced. The entries below the index in the WIRED register are "wired" and use of those mapped virtual addresses

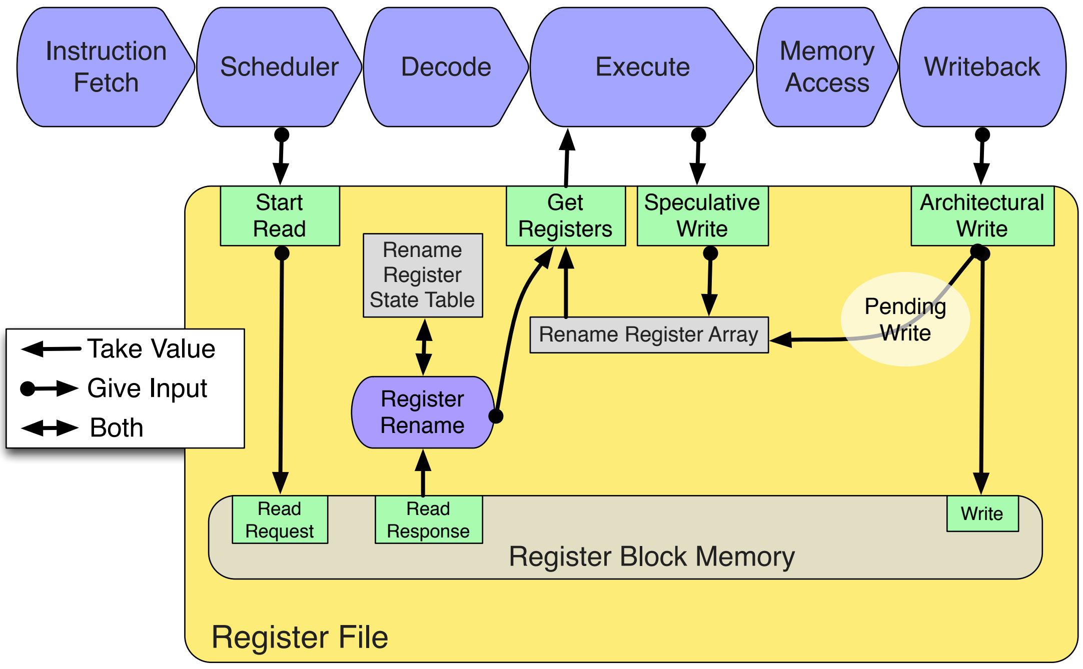


Figure 5.5: BERI register file

doing a probe immediately before. The vast majority of TLB write operations used the *write random* instruction which allowed insertion using an arbitrary algorithm. We found that a TLB that only used fully associative entries at the bottom but that had a large direct-mapped cache at the top worked transparently for FreeBSD.

My design of the BERI TLB is parameterisable, taking advantage of the parameterisation of BSV, but currently has 16 fully associative entries and 128 direct-mapped entries, as depicted in Figure 5.6. BERI conserves space in the associative table because FPGAs do not currently integrate content addressable memories (CAMs), but do integrate standard memories (BRAMs). Most of the entries in the TLB are therefore in a simple memory[3]. The TLB instruction *write random* on BERI maps an entry into the top 128 entries using a simple hash of the lower bits of the virtual page number. A *write indexed* operation will only write to the specified index if it is in the bottom 16 entries. However, a write request to an entry above 16 will simply write to the hash of the virtual page number, ensuring it will be found on a lookup. If this *write indexed* operation immediately follows a *tlb probe* operation and is simply modifying an entry that was inserted using *write random*, then there will be no conflict. In FreeBSD, it seems that this is always the case.

A simple direct-mapped TLB does not work because it is possible that a single instruction uses two virtual pages (one for the instruction and another for data) that map to the same entry in the TLB. In this case the program ceases to make progress as it alternately misses the TLB for its instruction address and its data address. Therefore the entries in the fully associative set above the *wired* register (below which random writes should not occur) are treated as a victim buffer for the 128 direct-mapped entries. Thus the hardware might move entries in the table without notifying the operating system. Since FreeBSD does not maintain a shadow status of the hardware page table but simply

are guaranteed not to throw TLB exceptions.

[3]Commercial ASICs processors also commonly implement large TLBs with memories rather than fully associative arrays due to area and power concerns.

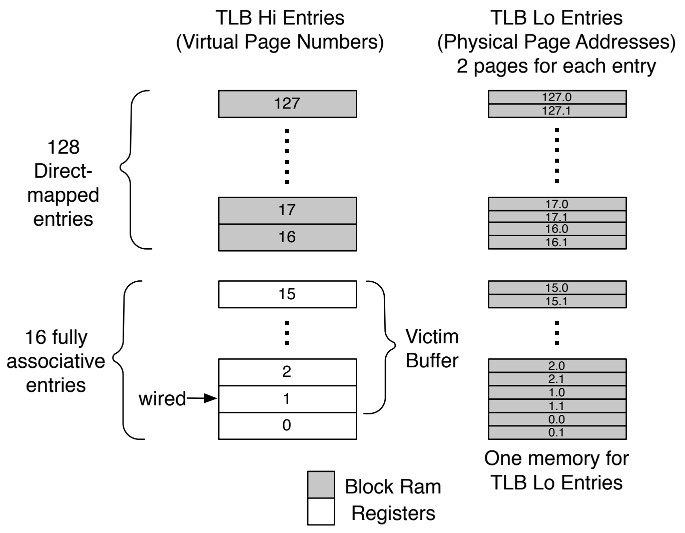


Figure 5.6: BERI TLB structure

probes the hardware table before any modification, this does not cause a problem.

A TLB typically requires multiple interfaces, for example for the instruction and data memory accesses. The mkTLB module contains a single, shared TLB lookup but maintains a small cache of TLB entries for each interface. The number of interfaces and the size of the cache on each interface is variable. These interface caches are also direct-mapped to avoid affecting the critical path. Cached entries are cleared on any TLB write to prevent coherence issues and a TLB cache miss causes a 3 cycle delay for a full TLB lookup. This caching arrangement allows full TLB translation throughput in the common case for all interfaces with very low delay on each interface and without duplicating primary TLB structures.

5.2.7 Debug unit (mkDebug)

I have also developed a debug unit for BERI to support experimental software and hardware development. The BERI debug unit is able to pause the pipeline, insert instructions, and trace execution. The execution trace engine can hold records for the last 4,096 executed instructions in a circular buffer so that recent activity can be inspected at a breakpoint. The state machine for reporting traces also allows continuous tracing of execution which can trace at 60,000 instructions per second over JTAG on the Terasic DE4, about twice the speed of simulated execution. This tracing engine has been useful for debugging experimental features under an operating system. Simulation performs 30,000 instructions per second for both the boot and any executable which needs to be traced, making debugging of the compiled executables intractable. The tracing engine in hardware allows a boot at full speed and a full trace for only the required period.

5.3 Simulation infrastructure

The same BERI design is built for simulation and for synthesis, but a different top-level module is used for each. When compiled for simulation, *mkTopSimulation* passes BERI

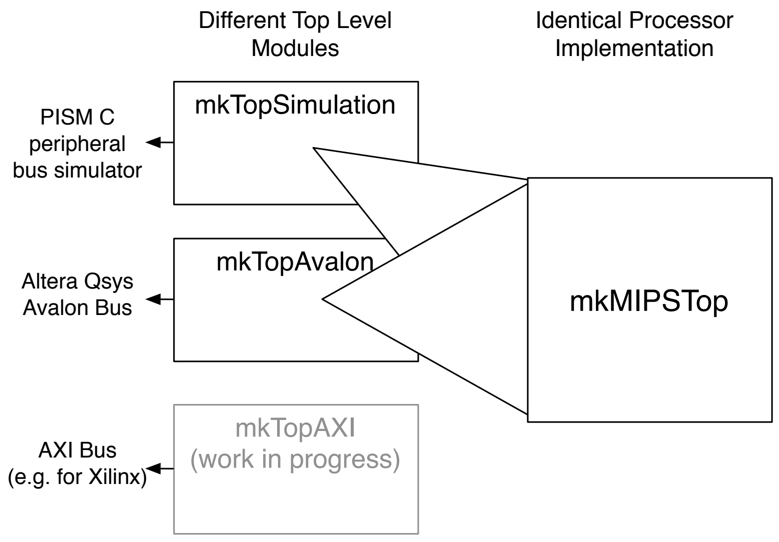

Different Top Level Modules
Identical Processor Implementation
PISM C peripheral bus simulator
mkTopSimulation
Altera Qsys Avalon Bus
mkTopAvalon
mkMIPSTop
AXI Bus (e.g. for Xilinx)
mkTopAXI (work in progress)

Figure 5.7: BERI simulation versus synthesis instantiations

bus requests to a library in C through a BSV "BDPI" interface. The C library, called PISM, emulates the hardware bus. Robert Watson and Philip Paeps have developed PISM peripherals to emulate the Altera JTAG UART, an SD card controller, a touch screen display, and the main DRAM memory. With this arrangement the differences between the simulated system and the synthesised system are relegated to the top level module. This ensures that a simulated design is as identical as possible to the synthesised design so that discrepancies between the two do not introduce bugs in hardware which are difficult to reproduce in simulation. This arrangement is depicted in Figure 5.7.

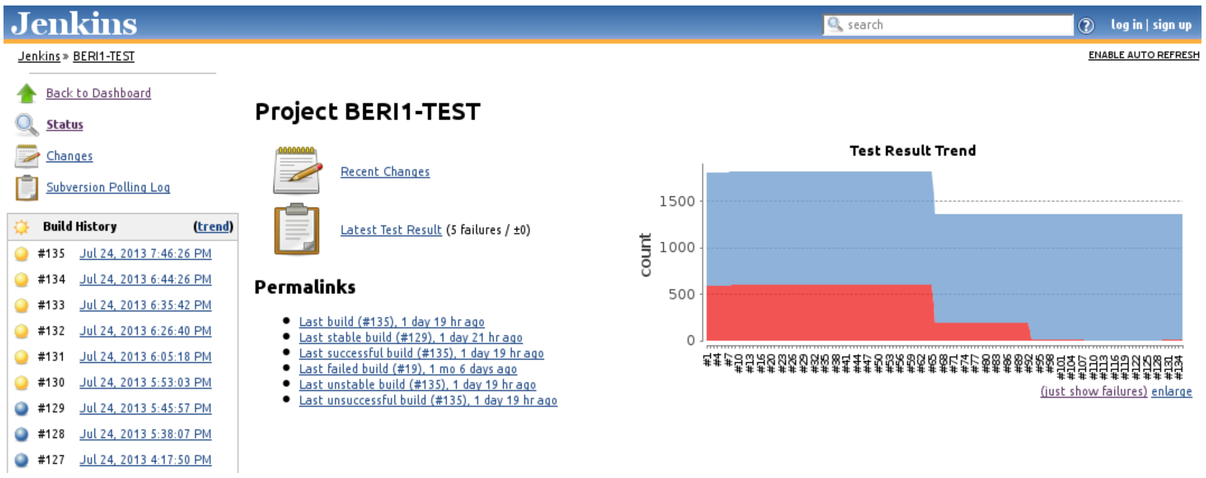
5.4 Test suite

The BERI open-source project also includes a unit test suite which currently includes over 1,400 tests spanning from basic functionality to TLB fills and cache tests. This test suite was constructed by Stephen Murdoch and the tests were contributed by Robert Watson, Michael Roe, Robert Norton, Colin Rothwell, myself, and a few others. The test and category counts are reproduced in Table 5.1.

I run this test suite from the command line before each commit to ensure that changes to the processor do not break functionality in unexpected ways. The test suite has also been harnessed to the Jenkins continuous integration framework (Figure 5.8) so that it is run automatically on every commit to ensure the operational correctness of the hardware design at all times. This type of regression suite is common in open-source software projects but is less known in open-source hardware due to a lack of common infrastructure. I expect that the comprehensive test suite that accompanies BERI will assist its success as an open-source project by allowing contributors to the project to efficiently check their own changes before submission for inclusion in future releases.

Test Class	Number of Test Routines	Number of Tests
ALU	68	334
Branch	71	276
Cache	4	17
Coprocessor 0	66	312
Floating Point	36	93
Framework	18	77
Fuzz Regressions	10	10
Memory	45	307
TLB	16	50
Total	334	1476

Table 5.1: Test Counts



Jenkins

search

log in | sign up

Jenkins » BERI1-TEST

ENABLE AUTO REFRESH

Back to Dashboard

Status

Changes

Subversion Polling Log

Build History	(trend)
#135	Jul 24, 2013 7:46:26 PM
#134	Jul 24, 2013 6:44:26 PM
#133	Jul 24, 2013 6:35:42 PM
#132	Jul 24, 2013 6:26:40 PM
#131	Jul 24, 2013 6:05:18 PM
#130	Jul 24, 2013 5:53:03 PM
#129	Jul 24, 2013 5:45:57 PM
#128	Jul 24, 2013 5:38:07 PM
#127	Jul 24, 2013 4:17:50 PM

Project BERI1-TEST

Recent Changes

Latest Test Result (5 failures / ±0)

Permalinks

- Last build (#135), 1 day 19 hr ago
- Last stable build (#129), 1 day 21 hr ago
- Last successful build (#135), 1 day 19 hr ago
- Last failed build (#19), 1 mo 6 days ago
- Last unstable build (#135), 1 day 19 hr ago
- Last unsuccessful build (#135), 1 day 19 hr ago

Figure 5.8: Sample of BERI tests run by Jenkins on every source commit

5.5 Capability extensions

I have extended the BERI processor with capability addressing for the current work. The capability coprocessor can be conditionally compiled into BERI to make it the CHERI processor. The capability unit and capability decode logic are braced in *ifdefs* to be included or not by the compiler's preprocessor. The other major change in CHERI over BERI is the tag controller between the L2 cache and the external memory interface.

5.5.1 Capability coprocessor

As shown in Figure 5.9, the segment descriptor extensions are modularised as a MIPS coprocessor. All MIPS memory accesses are conceptually offset via segment descriptors. Every instruction interacts with the capability coprocessor from the *Scheduler* stage through the *Writeback* stage.

Instruction fetch optimisation — While the program counter of every instruction is conceptually offset via the program counter capability (PCC) in the capability unit, it is not necessary for the capability unit to translate the address in instruction fetch. The main CHERI pipeline uses the absolute virtual address (i.e. post-capability offset) for branch prediction and relative branch calculations with the capability unit only verifying that this address is within the bounds of PCC at a later stage. This optimisation both removes an *add* from the instruction address generation path and allows PCC to be observed in the same stage as it is modified to simplify forwarding. This lazy bounds check is done in the *Get Capability Response* interface in Execute (see Figure 5.9). The BERI branch predictor is unchanged in the capability case and uses the absolute virtual address of the program counter for prediction both to avoid the extra *add* before address translation and to allow predicting through capability jumps, that is, control-flow instructions that change PCC. If the branch predictor only predicted the "offset" in the current PCC, as was the case in the initial implementation, it would not have enough information to predict the program counter when PCC changes. Furthermore, all operations that change the PC, such as jumps, are calculated with respect to the current architectural PC, which is an offset, and then are finally added to the base of PCC before writing back to the branch predictor. Finally, operations that observe the PC, such as link operations, must do a reverse transformation and subtract the base of PCC. These functions are implemented in the main pipeline using the *Get Architectural PC* interface in the Execute stage. As a result, the use of an absolute virtual address for the PC internally in the pipeline is invisible to software.

As described in Figure 5.9, I have designed the capability unit with a general-purpose request and response interface that is enqueued in the *Scheduler* stage of the pipeline and dequeued in *Execute*. This interface implements both capability manipulation instructions and also the offset of all memory operations via the appropriate capability register. For manipulation instructions such as **CIncBase**, the *Scheduler* may request the descriptor to be modified and *Execute* will pass in the value by which the base must be increased[4]. The capability coprocessor uses the forwarding register file in Figure 5.5 parameterised to hold 257-bit capabilities, which include a 1-bit tag. As described in Section 5.2.4, the

[4]Execute must pass in the general-purpose register value because the forwarded value is not available before this stage in the pipeline.

forwarding register file provides full register forwarding using a result buffer of rename registers.

For standard memory operations, such as a **LB** (load byte), *Decode* will pass in the required capability register number, in this case C0, the descriptor used for all general-purpose accesses. The capability unit uses an *ActionValue* method in BSV for the interface to *Execute* so that, in this case, *Execute* will both pass in the calculated offset from the main pipeline and will receive the final virtual address from the capability coprocessor in preparation for the *Memory Access* stage of the pipeline. Any exception from the capability coprocessor is not reported in *Execute* but is calculated during the *Memory Access* stage and reported in *Writeback* to reduce the critical path.

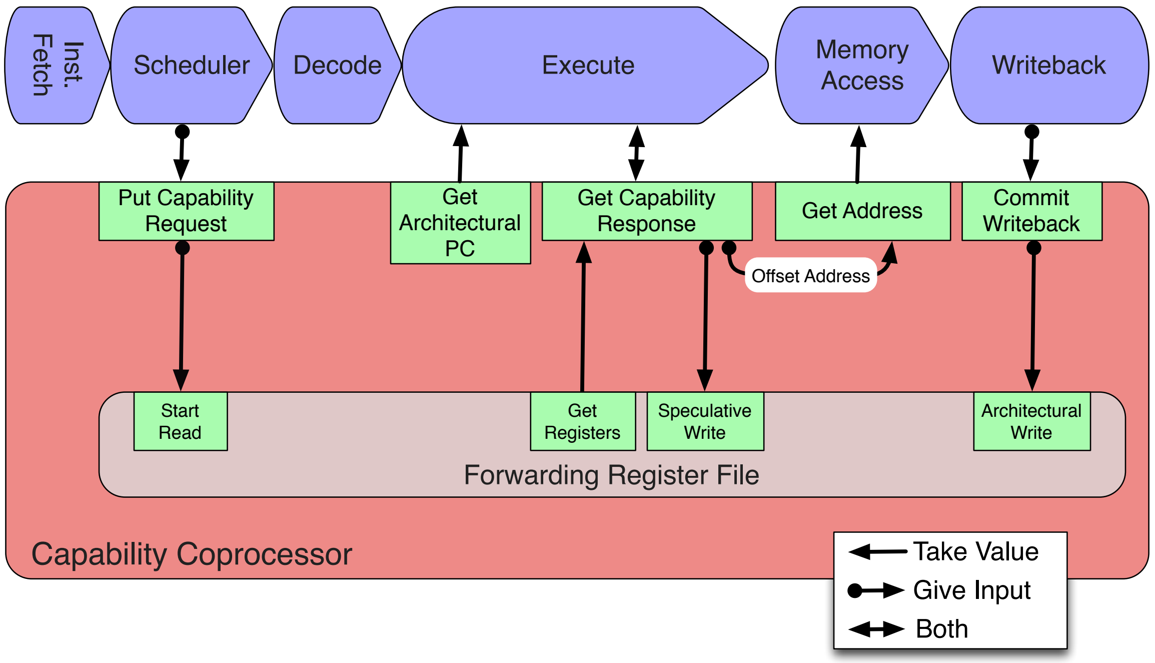


Figure 5.9: Capability coprocessor internal structure

A **CLC** instruction (load capability via capability) will deliver the virtual address in *Execute* as with other memory operations, but will also provide a capability to Execute for stores or receive a capability from Writeback for loads. It might be noted that commercial processors are not likely to require a special memory path expansion for wide writes. Current Intel Haswell processors already have a 256-bit data path to first level caches, for example, and ARM also handles vector loads and stores.

In general, capability manipulation instructions consume a single cycle, the minimum for instructions in BERI. The only exception is that loads that are consumed immediately will cause a delay of two dead cycles. This style of segment descriptor manipulation is orders of magnitude higher performance than protected segment manipulation on IA-32 implementations[5].

[5]Lam *et al.* found that a safe modification of the segment table on x86 required 241 cycles when highly optimised on a 1.1GHz Pentium III [50].

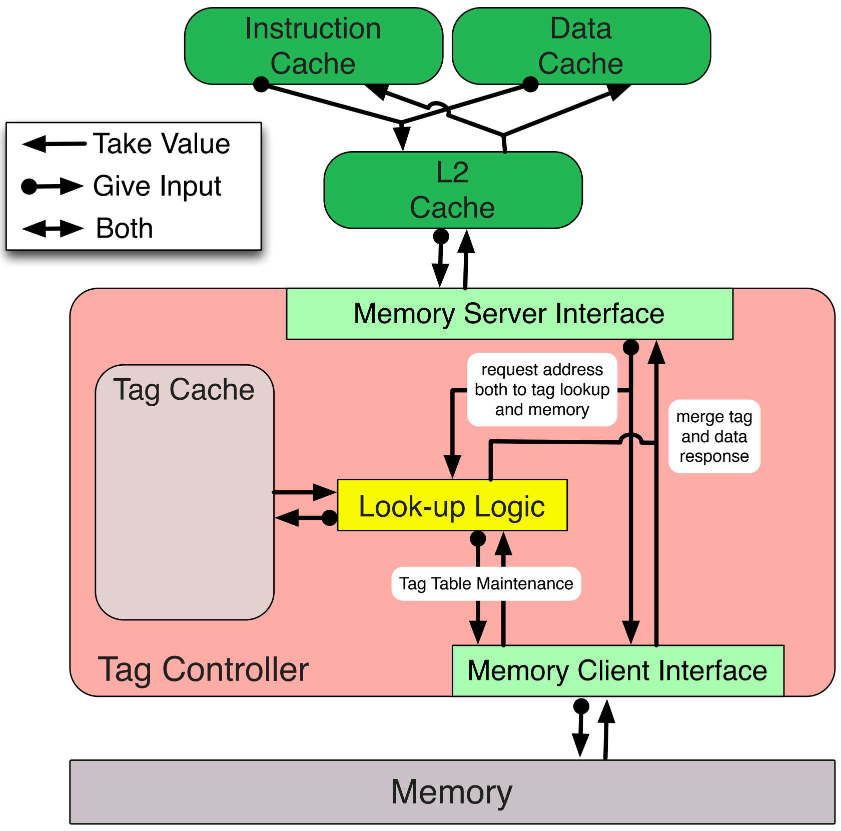


Figure 5.10: Tag Controller Architecture

5.5.2 Tag controller

The tag for each 32 byte line of memory must be kept strictly coherent with the data to ensure the integrity of capability segment descriptors. I have chosen an obvious implementation that ensures coherence, adding the capability tag to the general cache tags in the level 1 data and level 2 cache. The level 1 cache preserves capability tags written from the capability coprocessor interface and clears capability tags upon a write from the data interface.

When memory requests miss in the cache hierarchy, CHERI pipes memory requests from the level 2 cache through a tag controller to ensure that every memory response has a correct tag. This tag controller maintains a 16 MiB table of tags in memory for the bottom 4 GiB of address space. A request to the memory above 4 GiB will return a negative flag and tag writes above this level will not be preserved. This is sufficient because our current system has only 1 GiB of DRAM.

This tag controller, illustrated in Figure 5.10 also has a 2 KiB tag cache in 64 bit lines which can store tags for 512 KiB of address space. Memory requests which hit in this tag cache generate only one memory request for the data, but a miss in the tag cache will generate two memory requests, one for the data and another to fill the tag cache line needed for this access.

A more flexible design of the tag controller would expose a peripheral interface with several regions of memory which may contain tags along with table locations for each. An operating system would allocate regions of memory for the tag controller to maintain tables for all memory which may contain capabilities.

Tags are stored in system accessible memory. While this may impact security if the operating system is not careful to prevent unauthorised access to these tables, making the tags accessible to the system allows paging memory to and from the disk along with the relevant portion of the tag table. Trusting the system to protect the tag table is reasonable because the capability mechanism already depends on the operating system's protection of the page table. Since capability segment descriptors refer to virtual memory, if a process can modify page translation, it can redefine what memory the descriptors describe. While CHERI depends on operating system safety, CHERI capability protection can also be used to strengthen it. If page tables and the tag table are stored outside of general system memory, the operating system can restrict the general kernel capability segment and delegate capabilities to these structures only to the routines that require them.

Similarly, the CHERI capability mechanism depends on the safety of the paging mechanism. If pages are to be safely paged out and into memory, not only the tags, but also the data must be preserved. Since the descriptor itself is contained in data, data corruption can result in expanded privilege. Therefore CHERI must trust operating system disk management to protect swap files from unauthorised modification and an operating system might rely on CHERI segmentation to strengthen this protection.

5.5.3 Area and speed cost of capability extension

The capability coprocessor adds area and speed overheads to the BERI pipeline. The reader should be aware that our pipeline is not representative of a commercial ASIC pipeline or even of a commercial soft-core pipeline as our pipeline has been optimised for modularity and extensibility rather than area and speed.

Figure 5.11 shows CHERI laid out on the Altera Stratix IV EP4SGX230 that we have used for the prototype. The larger components of CHERI are coloured and labelled and the peripherals of the system, including DDR2 and Ethernet, are in shades of blue. Excluding peripherals, synthesis of BERI, the 64-bit MIPS without the capability coprocessor, occupies around 22% of the FPGA. A synthesis of CHERI (BERI with the capability coprocessor) occupies around 28% of the FPGA, so we currently see an area overhead of about 27% for capability support support. This overhead includes not only the capability coprocessor and the tag cache, as seen in Figure 5.11, but also logic in the main pipeline to allow loading and storing 256-bit capability segment descriptors to the data cache. A commercial implementation should optimise descriptor size and would also have a wide vector path to memory that would be shared with the capability coprocessor. I will leave area optimisation of the implementation to future work.

Capability support also affects the timing of our implementation due to wider paths and increased complexity in the pipeline. Capability support in our current implementation reduces clock speed by 11.2%, with a synthesis of BERI achieving a maximum frequency of 124.19 MHz, while a synthesis of CHERI (including the capability coprocessor) achieves a maximum frequency of 110.36 MHz. Little attempt has been made to improve the timing of either case beyond 100 MHz. It might be noted that we have prioritised the performance of capability operations to achieve single cycle latencies for all instructions. An implementation that expects less frequent capability segment descriptor updates could decouple capability modifying operations from the main pipeline while still achieving an order of magnitude better performance than kernel managed segmentation arrangements.

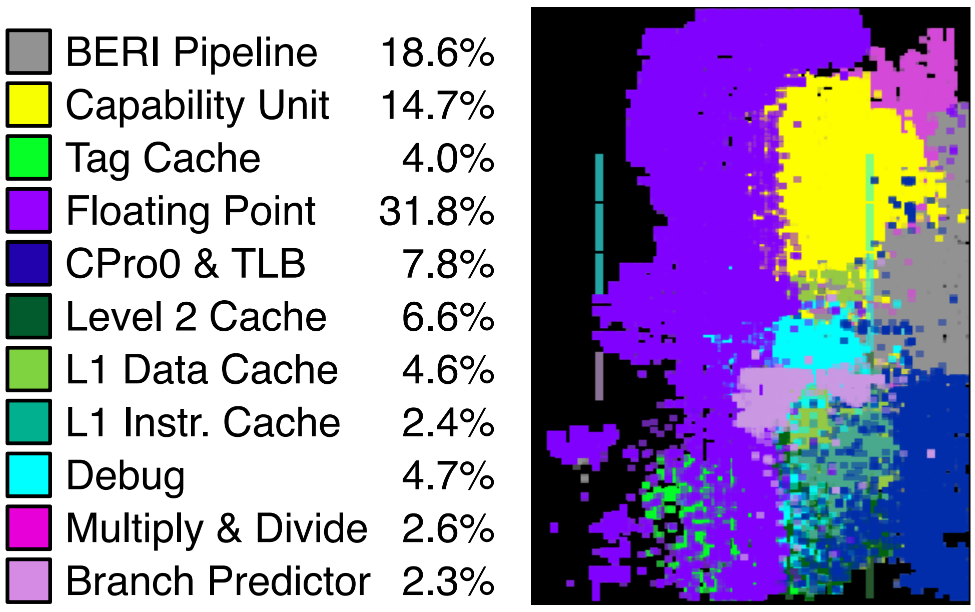


Figure 5.11: Layout of CHERI on FPGA

5.6 Conclusion

The CHERI processor is a robust and complete implementation, able to serve as a development platform to allow full exploration of its memory-safety extensions. The parameterisable renaming register file and branch predictor enable full, scalar pipeline throughput. The parameterisable TLB enables high performance for large workloads, as well as design-space exploration. Furthermore, the capability coprocessor has been designed for full performance with segment descriptors with several key optimisations that should be applicable to commercial implementations.

CHERI has also been designed for extension by the community to allow direct comparison with future memory safety research. We hope that the BERI and CHERI processors will facilitate high quality research with its accessible and extensible design.

Chapter 6

CHERI capability utility

In this chapter I will examine three uses for the CHERI capability mechanism in a modern software stack:

- Kernel access control
- Sandboxing
- Pointer safety

CHERI capabilities have been used in our current software stack for these three applications and I will briefly compare these implementations with alternative implementations using general-purpose instructions.

6.1 CHERI capabilities for access control

The fundamental definition of a *capability* in computer science is an unforgeable token of authority. Current computer systems have no notion of an unforgeable value; rather, any binary number can be created in a register or in memory. Therefore most protection is probabilistic, using long keys which are very unlikely to be fabricated without prior knowledge of the key, but which are trivial to fabricate if the key is known. Since arbitrary behaviour can be synthesised in user space, an operating system must depend on its own protected structures and memory to validate permissions on any request from user space.

CHERI capabilities introduce the concept of unforgeable values in user space. Due to tagged memory and constrained manipulation of capability segment descriptors, a process cannot create a capability segment descriptor that describes memory for which it does not already posses a descriptor. *CheriBSD*, FreeBSD supporting CHERI capabilities, an extension mainly by Robert Watson, begins each process with a capability for user addressable memory. It is possible for user space to hold capabilities to addresses outside user addressable memory, though these would fail if dereferenced due to rules in the TLB and are effectively *non-memory* capabilities. Since these *non-memory* capabilities can only be fabricated by the kernel, they can be used as keys to kernel resources.

If, for example, the kernel were to use capabilities as file descriptors, a file descriptor passed from user space as part of a system call could be accessed directly without an access control check because it would not be possible for the process to create a valid file descriptor capability that was not delegated by the kernel. Furthermore, if a process has sandboxes in its own address space, the file descriptors delegated to the process and held in its own memory will respect the sandboxing policy of the process. As with other data, the file descriptor capability must be stored in memory so an effective sandboxing mechanism will prevent not only the unauthorised spread of data but also the spread of file descriptor access within the address space, allowing a complex non-hierarchical privilege policy for kernel resources within the process. Without this mechanism the kernel would have to store access permissions not only for the process but also for each sandbox within the process, severely limiting the scalability of a sandbox mechanism.

In the future we might remove the TLB restrictions to the user accessible address space and depend on the capability mechanism to restrict user space to non-kernel addresses. This optimisation would allow the kernel to delegate portions of unmapped kernel memory directly to user space. This might be used for network buffers, for example, to eliminate the need to either copy user space buffers or insert shadow mappings in the page table of the process.

6.2 CHERI capabilities for sandboxing

CHERI capabilities can also be used for sandboxing a process into sub-address spaces, essentially allowing a process to have its own memory protection policy for program portions. Current memory protection units reserve memory protection management for the kernel and equally trust (or distrust) the entirety of a process. However, programs usually have their own trust model, having portions that are expected to interact with the kernel and one another using pre-defined functions. CHERI facilitates this use case with the default capability segment model. The first capability segment register is the default capability through which general-purpose memory accesses are addressed. In the simple sandboxing model, the running code only has access to a single contiguous segment. This segment may contain a series of sealed capability segments representing targets which it is allowed to invoke outside the sandbox.

CheriBSD currently implements this model and several applications have been sandboxed using this facility, including **tcpdump**, a slide viewer application, and test programs. CheriBSD uses the hardware **CCALL** and **CRETURN** instructions to enter and pass between sandboxes, but, for the sake of flexibility for experimentation, implements the actual sandbox transition in a software trap handler. This handler maintains a stack of return capabilities to facilitate both the eventual **CRETURN** and also stack unwinding so that sandbox violations can be reported properly to the caller.

FreeBSD Capsicum or Google NativeClient might be considered alternatives to this sandboxing model. FreeBSD Capsicum creates a separate process and address space for each sandbox of a program, with the kernel managing independent memory allocations and access control lists for each sandbox process. This model requires a full context switch to pass from one sandbox to another with independent TLB entries for each sandbox, even for shared objects. While CHERI sandboxing allows sandboxes to share an address space, it is not possible for non-overlapping sandboxes to actually share data unless they use explicit capability addressing instructions. This prevents transparent ports of non-sandboxed programs that pass-by-reference across the sandbox boundary. Nevertheless, CHERI capability addressing for memory sharing is still more straightforward than the multi-process solution of shadow page table entries with inter-process synchronisation.

Google NativeClient allows a sandbox to run within the address space of the process but requires that the sandboxed binary be statically verifiable and therefore greatly constrains the flexibility of the compiler to produce optimal code. This approach trades general-purpose performance for finer grained protection domains, which is acceptable for small portions of untrusted code but would be unacceptable for performance critical functions, such as encryption and image-rendering libraries. Furthermore, NativeClient allows only one layer of sandboxing within one supervisor. CHERI sandboxing allows an arbitrary number of sandboxes with invocations directly between them, allowing a path to more seamless integration with existing language structures for function invocation.

Simple CHERI sandboxes have the major usability drawback that the implicit segment for general-purpose references must be a single contiguous region. Therefore, in the simple model, each sandbox must have its own stack and heap mapped into its own sandbox region, contiguous with its executable program memory. This currently requires that each sandboxed portion of a program be compiled and linked independently and contain copies of libc (or at least stubs of used functions) in each sandbox. However, this model can at least use unmodified code. A more flexible model would require all references to shared code and data to use capability instructions explicitly with functions being invoked with **CJALR** to change the program counter capability and data being addressed through the correct capability register. While capability addressing for out-of-sandbox objects might be somewhat automated, changing the size of these pointers in memory will affect program-visible memory layout which is likely to break most C programs. This means that either some amount of manual program modifications would be needed for a C program to use CHERI protection or an out-of-band capability storage model should be used. Managed languages, which do not expose the binary format of pointers to the program, should be able to adopt CHERI protection transparently.

6.3 CHERI capabilities for pointer safety

CHERI capabilities can also be used for pointer safety by automated bounds checking. Bounds checking is the most fine-grained use of segmentation hardware. If a segmentation mechanism can efficiently support fine-grained bounds checking operations, it should be sufficiently fast for all coarser grained applications. When inspecting MIPS assembly in this section, one should note that all instructions are single cycle without cache misses and so instruction counts directly translate to performance in a scalar pipeline.

When implementing simple fat-pointer-type bounds checking, either the programmer or the compiler is responsible for several things related to bounds management:

- Assigning bounds at object creation.
- Passing the bounds along with the object.
- Inserting bounds checks before every memory access.

In the implementation of David Chisnall's extension to LLVM, the compiler is responsible for the first of these; that is, the compiler must set the bounds of a capability pointer when the object is first allocated. The second two responsibilities, passing bounds with pointers and checking bounds on an access, are automated by CHERI hardware and only require the compiler to use appropriate capability instructions for moving pointers and referencing memory. While the CHERI approach does require the compiler to be intimately complicit in memory protection, it does allow source code to be nearly unchanged.

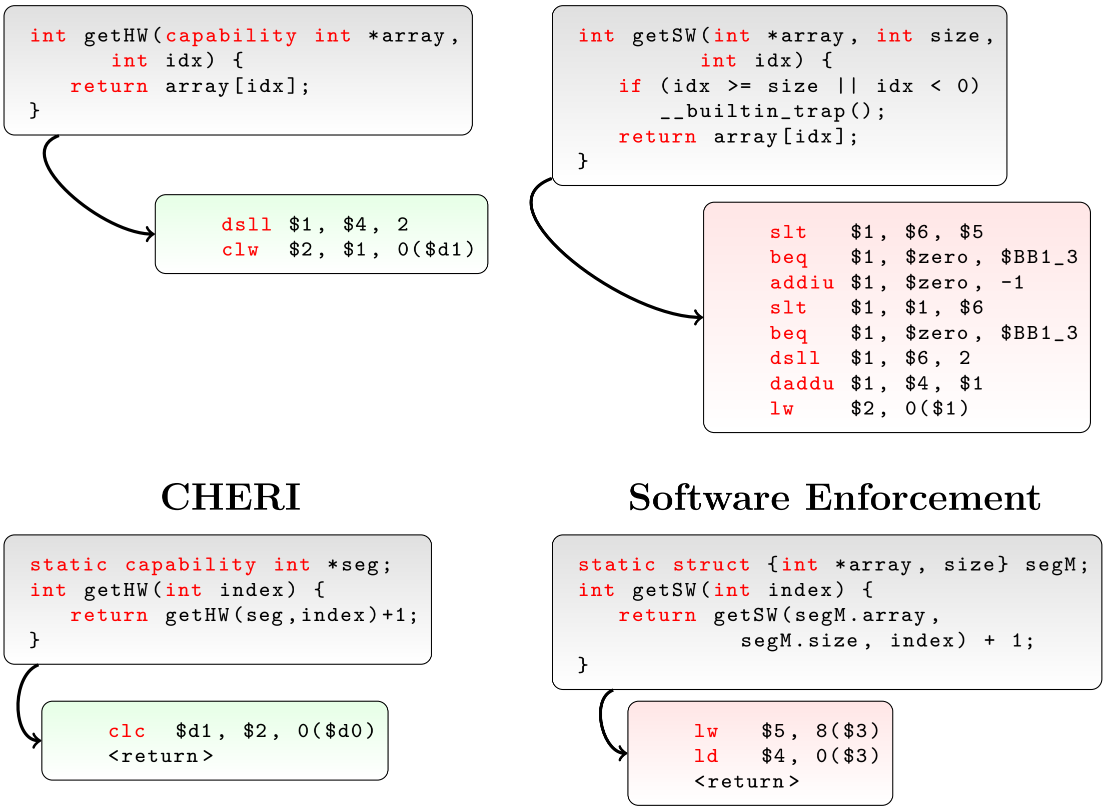
The Cyclone compiler explored modified types for memory safety [45], requiring only new types for pointers that should be automatically bounds checked. Cyclone demonstrated that a mildly invasive change to source code could effectively imply bounds checking for certain structures, while still giving the programmer control over where to spend the cost of bounds checking.

Another memory safe compiler, CCured, demonstrated that C could be automatically compiled for memory safety, with bounds checking inferred where necessary to preserve spatial safety [58]. CCured showed that bounds checking could be inserted by the compiler with no changes to the source code at all.

David Chisnall has extended the LLVM compiler and Clang frontend with a `capability` qualifier for pointers that are to be capability segment descriptors in hardware. This implements the Cyclone protected pointer type approach in LLVM for CHERI. A compiler for the CCured automated safety approach is under development.

6.3.1 CHERI LLVM bounds checking example

Two implementations of bound-checked array access are shown in Figure 6.1. In these two cases, constructed by David Chisnall, a `get` function uses a `getElement`function to perform a bound-checked array access. The case on the right is manually checked in software and the case on the left is automatically checked using the CHERI segmentation mechanism. To save space, only the differences in the assembly produced is shown.



```
int getHW(capability int *array,
      int idx) {
    return array[idx];
}
```

```
    dsll $1, $4, 2
    clw  $2, $1, 0($d1)
```

```
int getSW(int *array, int size,
      int idx) {
    if (idx >= size || idx < 0)
        __builtin_trap();
    return array[idx];
}
```

```
    slt   $1, $6, $5
    beq   $1, $zero, $BB1_3
    addiu $1, $zero, -1
    slt   $1, $1, $6
    beq   $1, $zero, $BB1_3
    dsll  $1, $6, 2
    daddu $1, $4, $1
    lw    $2, 0($1)
```

```
static capability int *seg;
int getHW(int index) {
    return getHW(seg,index)+1;
}
```

```
    clc  $d1, $2, 0($d0)
    <return>
```

```
static struct {int *array, size} segM;
int getSW(int index) {
    return getSW(segM.array,
              segM.size, index) + 1;
}
```

```
    lw   $5, 8($3)
    ld   $4, 0($3)
    <return>
```

Figure 6.1: CHERI capability segmentation versus Software Enforcement in C and Abridged Assembly

The software case is similar to the code generated by Cyclone [45] and other mechanisms that implement bounds checking via software fat pointers and the CHERI case uses our extension of LLVM to implement bounds checking using CHERI capability segmentation. The CHERI case has less than 1/4 the instructions for the array function which will translate into higher performance in simple pipelines.

An invocation of `getHW` and `getSW` is presented in `getHW` and `getSW` respectively. The extra load in the `getSW` case exposes a fundamental cost of software fat pointers, namely that the two components of the fat pointer must be loaded into separate registers with independent instructions. In more complex examples, this would be more significant because MIPS (in common with most modern architectures) permits passing only two fat pointers in registers, as it reserves only four integer registers for parameters with additional parameters being passed on the stack. In contrast, our architecture can pass four fat pointers in capability registers in addition to arguments in general-purpose registers.

We might note that, while `getHW` and `getSW` are generally equivalent, `getHW` will also throw an exception if the capability pointer is not, in fact, a valid pointer and if the permissions of the capability pointer do not allow reading. A software type check for pointers would require a shadow memory with a bit for each word of usable memory, though a hashing algorithm that uses upper bits of the pointer might be used to increase confidence that pointers are, in fact, pointers. Pointer permissions could also be packed into the upper bits of the length. Either of these additions would greatly increase the complexity of the software safety check.

Furthermore, we might note the loss of atomicity due to splitting the load of the fat pointer. The base and length are stored in adjacent words and, without a locking scheme, loading both into registers can not be atomic with respect to writes from another thread. This means that the bounds passed here, in addition to being more costly, are not safe in the presence of concurrency. This shortcoming is explicitly noted by the authors of Cyclone [45] which was not considered thread-safe due to lack of atomicity in pointer handling. Since they did not attempt to solve this problem, we may conclude that atomicity for fat pointers is not possible in software with reasonable performance. Restricting memory-safe programs to a single-thread is a serious limitation in the modern context. Intel partially solves this problem in iMPX by storing the value of the pointer atomically with the value of the bounds registers in a table structure. This stored value of the pointer is then validated against a future use of the pointer to ensure it was not modified apart from the bounds. Using iMPX extensions as simple fat pointers does nothing to guarantee atomicity between the pointer and the bounds, though the bounds are atomic with respect to each other. The CHERI solution of storing the pointer and the bounds as a single structure is both simpler and stronger than the Intel table approach, though less transparent to software.

6.3.2 Bounds checking memory safety analysis

Fine-grained memory protection, which our hardware extensions accelerate, enhances memory safety and reduces data misuse errors. These errors are often of concern because they lead to security vulnerabilities. Szekeres, Payer, Wei, and Song recently published a paper titled "Eternal War in Memory" [81] with an excellent categorisation and decomposition of security exploits that follow from memory safety violations; we use their analysis as a framework to consider the potential effectiveness of a system using CHERI capability segmentation.

In this section we analyse the value of a hypothetical language that uses CHERI capability segmentation features to enforce the properties of its abstract machine model. This shows the value of our hardware mechanisms in the limit and exposes shortcomings in the current mechanisms that prevent the acceleration of full memory safety. The properties we enforce using our hardware model are:

1. An array or structure cannot be accessed beyond its bounds
2. An array or structure cannot be used inappropriately as defined by read, write, and execute permissions
3. A pointer cannot be modified to point to something that it does not already encompass

In the parlance of the Szekeres *et al.* paper, we ensure spatial memory safety and pointer integrity, as well as pointer use to some degree. Spatial memory safety is enforced with the segmentation mechanism, pointer integrity is enforced with the capability tagging mechanism, and pointer use is enforced with some granularity using the capability segment permissions mechanism. Virtual machines that enforce these properties and more are common, such as the JVM and the Microsoft .NET virtual machine, but our extensions enable enforcement of these properties natively in hardware with nearly no slowdown versus unprotected code. We consider a hypothetical language that uses only these hardware supported features.

Figure 6.2 is reproduced from the Szekeres *et al.* paper, and gives a complete categorisation and decomposition of software exploits in six stages with the mitigation strategies each step. We have coloured red the steps that are prevented by a system that makes full use of fine-grained segmentation to enforce its abstract machine model. We have coloured yellow the steps that can be mitigated by use of appropriate permissions but which will not be prevented in all cases. We have coloured grey the stages that are not reachable in our system. We discuss each of the interesting cases in the first three stages below.

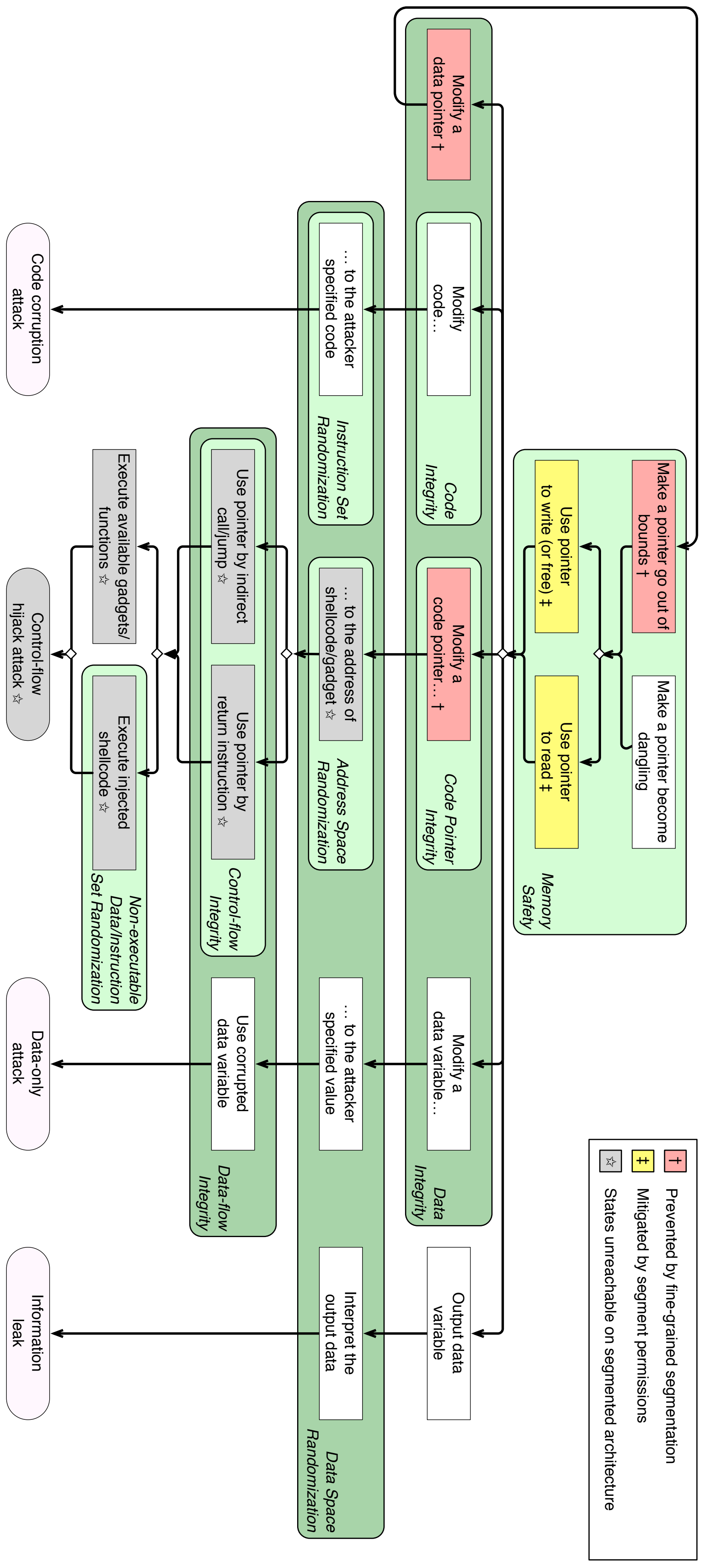

Make a pointer go out of bounds †
Make a pointer become dangling
Use pointer to write (or free) ‡
Use pointer to read ‡
Memory Safety
Modify a data pointer †
Modify code…
Code Integrity
Modify a code pointer… †
Code Pointer Integrity
Modify a data variable…
Data Integrity
Output data variable
… to the attacker specified code
Instruction Set Randomization
… to the address of shellcode/gadget ✫
Address Space Randomization
… to the attacker specified value
Interpret the output data
Data Space Randomization
Use pointer by indirect call/jump ✫
Use pointer by return instruction ✫
Control-flow Integrity
Use corrupted data variable
Data-flow Integrity
Execute available gadgets/ functions ✫
Execute injected shellcode ✫
Non-executable Data/Instruction Set Randomization
Code corruption attack
Control-flow hijack attack ✫
Data-only attack
Information leak
† Prevented by fine-grained segmentation
‡ Mitigated by segment permissions
✫ States unreachable on segmented architecture

Figure 6.2: Path from memory safety violation to exploit flow chart [81]

Make a pointer go out of bounds — This initial vector for attack is prevented trivially by using a bounded segment for every array or structure. Our mechanism strongly enforces spatial memory safety. This is one of only two starting points for a software exploit of a program and preventing it with low performance overhead is a significant improvement over the current memory safety/performance trade-off.

Make a pointer become dangling — This initial vector for attack is a matter of temporal memory safety; that is, the possession of a pointer to memory that has changed its use. This attack is not prevented by fine-grained segmentation because segment descriptors in our model are as persistent as simple pointers. Existing software techniques that do not allow careless reuse of memory would be necessary to prevent this exploit in the current implementation. These include safe stack allocation strategies, non-reuse of virtual address space for the heap [2] (reasonable in a 64-bit address space) as well as garbage collection, which may become more efficient with access to the tag table to identify valid references. Runtime compilers should be especially careful since they would have both writable and executable references in the same program and must not be careless with the life-time of the writable reference. All of the following stages of an exploit are considered in the context of a dangling segment descriptor because it is not possible for an out of bounds attack to proceed any further using fine-grained segment enforcement.

Use pointer to write (or free) — This step is prevented in some cases where the permissions of the dangling reference do not allow writing. The free problem would need to be solved in the language runtime.

Use pointer to read — This step is prevented in the rare case that the permissions of the dangling reference are write or execute only.

Modify a data pointer or Modify a code pointer — These steps are prevented by tagging capabilities in memory. The tagging mechanism is orthogonal to the segment reference mechanism so it is not possible to corrupt a segment descriptor in memory regardless of the permissions of the descriptor through which it is accessed.

Modify code or Modify a data variable — Assuming that a dangling segment descriptor is writable, our simple hypothetical language cannot prevent code or data from being modified. However, we can prevent the modification of code if a few rules are observed in the language, namely that executable code is never allocated on the stack or in the heap and that executables are never writable. These rules are quite reasonable and enforceable using fine-grained segmentation and are, in fact, observed by most language runtimes today.

Output data variable — Assuming that a dangling segment descriptor referred to readable data, we cannot natively prevent data leakage.

The result of the memory safety acceleration is that one of the four exploits is entirely unreachable and that all paths from red boxes are also unreachable. Blocking significant paths in this chart increases the assurance that bugs in software will not be exploitable, especially when common exploits require many such bugs to succeed. While hardware accelerated capability segmentation is a valuable tool in allowing the construction of full speed memory safe languages, it does not natively solve all problems with memory safety. Nevertheless we expect that these new mechanisms will accelerate not only the natively supported protections, but will also reduce the cost of software enforcement of the remaining safety properties.

6.4 Managed Languages

A managed language, such as Java or Microsoft C#, could combine the above uses of CHERI capabilities to accelerate enforcement of the language model. Language objects which are considered as encapsulated with inaccessible contents by the language model could become actually isolated protection domains in the hardware. Domain-crossing instructions would be used to invoke the "methods" of a protection domain. Furthermore, all language objects could be referenced through capabilities with precise bounds and access permissions, as defined by the language model. Such an application of CHERI capabilities would be much more natural than the application to C since these managed languages strongly assume a virtual machine with complex memory safety properties. While a managed language might see performance improvements, C would see a greater security benefit because of its unsafe memory model. Memory safety improvements to the C source of managed language compilers and virtual machines is likely to be one of the greatest benefits to managed languages on CHERI.

6.5 Conclusion

CHERI capability segmentation mechanisms have been used by CheriBSD and David Chisnall's extension to the LLVM compiler for access control, sandboxing, and for bounds checking. We also hope to extend these applications in the near future to implement automated bounds checking (both visible to the executable and invisible using out-of-band capability descriptors) as well as sandboxing with native hardware support. These applications demonstrate the value and flexibility of the CHERI memory safety extensions. In contrast, other memory safety proposals which have emphasised transparency for C programs have sacrificed general-purpose flexibility. For example, Mondrian Memory Protection cannot provide very fine-grained bounds checking and cannot enable access control using unique keys. Hardbound does not allow selective protection and is not optimal for coarse-grained sandboxing. Even iMPX, while generally a flexible approach, does not accelerate sandboxing or unforgeable keys. While the current use cases are only beginning to be explored, we anticipate many other creative use cases for CHERI capability segmentation as the CHERI open-source project attracts researchers with new ideas.

Chapter 7

Evaluation of CHERI performance

Performance is critical for memory safety. Szekeres *et al.* observed that memory safety techniques that introduce an overhead larger than roughly 10% do not tend to gain wide adoption in production environments, and that the rules of the Microsoft BlueHat Prize Contest specify an overhead of less than 5% [81, 56]. The importance of performance efficiency is becoming even more critical in the mobile computing environment. While desktop systems might have absorbed the impact of inefficient software with their plentiful spare cycles, mobile applications are being tightly scrutinised to preserve power even when ample computing resources are available. If memory safety is to become the common case, the cost of memory safety must be a barely discernible overhead.

This chapter will characterise the performance of our existing implementation under FreeBSD for several benchmarks and will present an initial exploration of domain crossing performance. These results will demonstrate that CHERI capability segmentation works in practice and is approaching the overheads necessary for widespread adoption with obvious optimisations still available.

7.1 Performance evaluation for bounds checking

Segment descriptor manipulation mechanisms have traditionally had performance overheads on the order of TLB manipulation routines. However, CHERI segments are intended to replace pointers so CHERI segment descriptor manipulation is designed to be comparable to pointer arithmetic. Therefore I have chosen to initially evaluate CHERI segment descriptor performance in the context of fat pointer bounds checking. Bounds checking using software fat pointers is functionally equivalent to bounds checking using CHERI segment descriptors, which can be considered as hardware-supported fat pointers. Bounds-checking operations will include copying descriptors, restricting descriptors, loading and storing descriptors to the stack, and of course accessing general-purpose memory through descriptors.

Our benchmarks were run on the CHERI CPU described in Section 5.5.1, which was implemented in the Altera Stratix IV GX EP4SGX230 FPGA on a Terasic DE4 board. All results were collected running userspace programs on top of the open-source FreeBSD operating system to demonstrate the maturity of our platform and its ability to virtualise processes that use segment descriptors.

7.1.1 Array microbenchmark

Our Array microbenchmark is designed to measure the per-access cost of bounds checking. The same source code was used for both the unsafe and hardware bounds checked cases to demonstrate that our compiler extensions are transparently backward compatible. When compiled for a pure MIPS target, the C preprocessor simply removes the `capability` qualifier.

Listing 7.1: Relevant source from array benchmark.

```
int * indices = initialise(SIZE);
capability char * array = initialise(SIZE);
for (i=0; i<SIZE; i++) {
  int index = indices[i];
  sum += array[index];
}
```

As seen in code listing 7.1, the array benchmark repeatedly indexes a byte array with elements from an integer array. This arrangement prevents static analysis from removing the bounds check.

Figure 7.1 presents the overhead for the inner loop introduced by bounds checking against three different baselines. The first is an Intel i7 930 at 2.8GHz running Linux, the second is on BERI at 100MHz as compiled by LLVM, and the third is on BERI at 100MHz with hand optimisations. The Intel baseline completes each iteration of the loop in 0.71ns, or about one iteration through the loop every two clock cycles. The unoptimised BERI baseline requires over 11 cycles to complete the loop, and the optimised BERI case requires almost 9 cycles. CHERI numbers were gathered in simulation for convenience and to remove uncertainty, though correct operation of the benchmark was confirmed in hardware. All cases were run at four optimisation levels and the fastest was selected in each case.

Each base case is compared against a manual check inserted into the source as well as bounds inserted by the Clang *-fsanitize=bounds* option. Because CHERI allows registers to be used as offsets to capabilities, it effectively supports register-relative addressing and thus gains an orthogonal advantage over the base MIPS case which must perform an extra add. Therefore, CHERI includes both a case with register-relative addressing removed (i.e. manually adding the offset to the base of the capability before every dereference) and a case with full register-relative addressing.

While Intel sees a 37% overhead for both manual and automatic bounds checking, the unoptimised BERI case has a 77% overhead for manual bounds checking and a 99% overhead for automatic bounds checking. CHERI bounds checking without relative addressing has 0% overhead to three decimal places due to an equivalent instruction count. CHERI using its segment-relative addressing gains a 7.5% advantage. Hand-optimising the loop makes the overhead for the manual check less dramatic, though CHERI with segment-relative addressing maintains a similar advantage.

For reference, a 2004 study on scientific computing in Java versus C++ found an overhead of 137% for the fastest JVM with just-in-time compilation support on a nearly identical bounds checking benchmark [84].[1]

[1]Another study using very highly optimised assembly and the best implementation of the IA-32 BOUND instruction found an overhead of 20% in the best cases and averaging 40% for high-level benchmarks that will have many fewer bounds checks relative to our microbenchmark [19]. Also, the CASH system

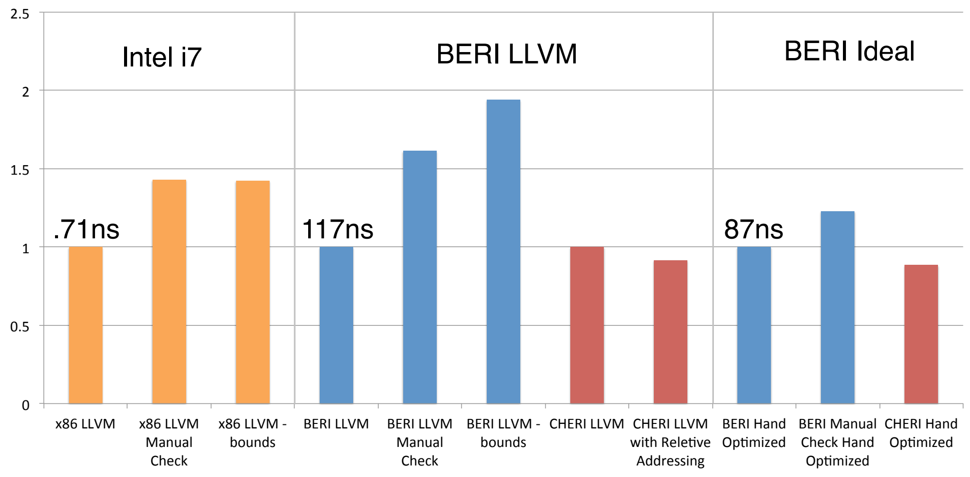


Figure 7.1: Bounds checking overheads for the array benchmark

The results of the array benchmark indicate that applications that check the bounds of repeated accesses to a large structure can adopt CHERI memory protection for these structures at nearly no cost.

7.1.2 Quicksort microbenchmark

Our Quicksort microbenchmark is designed to expose the cost of subroutine calls for bounds checking schemes. This benchmark implements a simple in-place quicksort and uses our `capability` annotation to provide fine-grained bounds checking on the subset of the array passed to each recursive call when built with hardware bounds checking support.

Passing the fat pointer data structure has been found to be the largest source of overhead for fat pointer bounds checking when the actual bounds check is highly optimised [50]. In our implementation, we can pass a segment descriptor in a single register and spill or restore a segment descriptor from the stack with a single (large) memory access. These larger stores to the stack do not increase instruction count but do increase cache pressure. Figure 7.2 shows the results of this benchmark on different array sizes. As before, these results show the average of 10 runs, with the error bars extending to one standard deviation.

The differences in performance are very small for most array sizes and are shown as percentage slowdown for the hardware bounds checked case in Figure 7.3. The overall slowdown is under 7% in every case. The general performance delta is due to the extra instructions that set the length of the array at each recursive invocation. The timing variations between runs are largely due to different layouts in physical memory, causing subtly different caching behaviours. The slight pattern of growing performance disparity as sort sizes increase should be due to the increased cache pressure caused by large segment descriptors being used in place of pointers on the stack.

Strangely, an array size of 16 KiB saw a 27% speedup. This is likely due to pathological cache aliasing and illustrates that CHERI capability segmentation reduces bounds checking overhead to the point that it is likely to not be the performance bottleneck.

used the IA-32 segmentation primitive for bounds checking a limited number of arrays on higher level microbenchmarks and found an overhead of 28.6% [50].

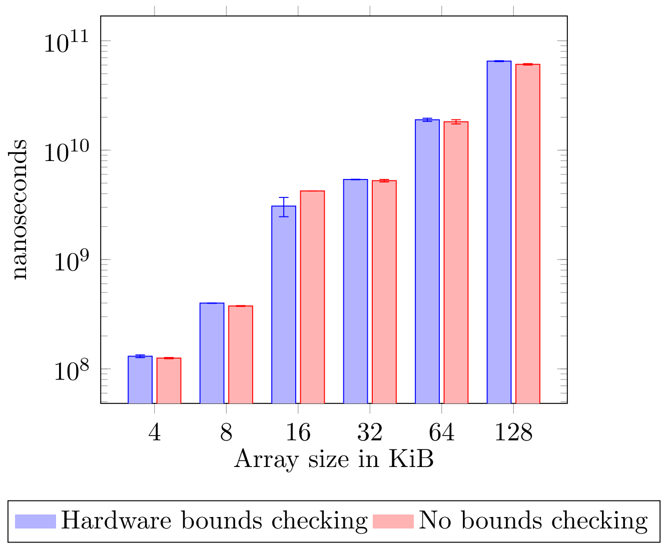


Figure 7.2: Results of the quicksort benchmark

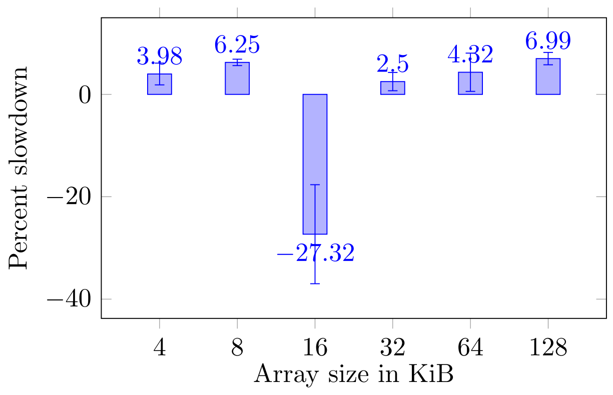


Figure 7.3: Slowdown from hardware bounds checking versus no protection in the quicksort benchmark

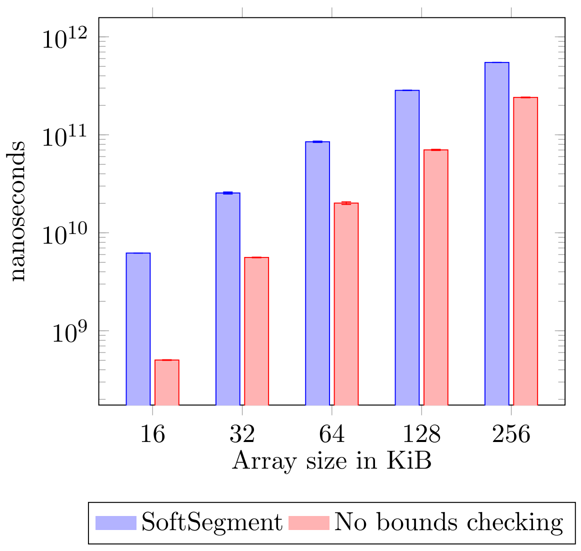


Figure 7.4: Results of the quicksort benchmark on an Intel CPU

I implemented the same algorithm in Cyclone on x86, using fat pointers in the same way, and saw a 131% slowdown versus the unprotected case. Cyclone stores the base and length in adjacent words in memory and does not perform any synchronisation so is vulnerable to the same race conditions that were described in Section 6.3. This is not an entirely fair comparison, as the Cyclone compiler is fairly old and does not propagate bounds checks between functions, causing a lot of overhead in the swap function, which is inlined in the C versions.

LLVM AddressSanitizer [73], which I found to have 23% overhead in this benchmark, is a software address validation scheme which consults a global table of allocated memory at each access. This approach is user-friendly because it does not require a change in calling convention and thus allows arbitrary use of library code without recompilation or rewriting. It gives weaker protection, however, as it only knows whether an address is valid allocated memory, not whether is belongs to the current namespace, a common weakness with other address validation schemes. Our implementation is a pointer safety scheme which not only verifies that the reference is within the original array, but even that a reference is within the correct sub-array deep in the call stack.

To determine the approximate cost of implementing our own segmentation scheme purely in software on a modern superscalar CPU, David Chisnall also ran a modified version of this benchmark with "SoftSegment", which aims to provide a weaker form of the guarantees of our RISC segmentation in software. SoftSegment approximates tagged memory by protecting fat pointers against accidental modification via a simple hash check which is stored in unused space in the pointer. The results of this are shown in Figure 7.4, which compares the SoftSegment implementation against the unprotected version on a modern Intel Xeon E5-2680 CPU. With the out-of-order superscalar CPU running checks in parallel with the loads where possible, we still see a typical 4x slowdown in the SoftSegment implementation, despite having weaker safety properties than CHERI's tagged capability segments. Type checking pointers in software is not only slow, but also very difficult to do correctly in the presence of concurrency. Using the existing atomicity of the processor cache mechanism is a far more reliable solution for pointer type checking.

These Quicksort results demonstrate that CHERI memory safety with moderate pointer manipulation and movement still incurs an overhead that is generally less than 5% and has less than 1/3 the overhead of leading software memory safety techniques.

7.2 Olden benchmarks

In addition to the array and quicksort microbenchmarks, Mike Roe, David Chisnall, and I adapted two of the Olden benchmarks, mst and bisort, to use CHERI protection[2]. We added a `capability` qualifier to all pointer declarations in these benchmarks so that the compiler would use CHERI capabilities instead of pointers everywhere for the CHERI case. The benchmarks were compiled with the modified version of LLVM and run as userspace programs on FreeBSD 10. Mst and bisort use an array of single-linked lists and a binary tree, respectively. Both data structures use small allocations containing pointers, which become capabilities on CHERI, and so provide a stress test of a memory protection system. Each virtex in the mst list and each node in the binary tree is allocated independently and each capability in the CHERI case points to a single allocation, so these benchmarks use many, many capabilities.

To ease comparison, we ran the benchmarks with the same parameters as used in the evaluation of Hardbound: `bisort 25000 0` and `mst 1024 1`. With these parameters, bisort will sort a binary tree of 25000 nodes, and mst will compute the minimum spanning tree of a graph with 1024 vertices. The three versions of the benchmarks were: unmodified MIPS code; MIPS code with software bounds checks inserted by CCured [58]; and MIPS code with manually inserted hardware bounds checks.

CCured was chosen as a comparison point because it was an important design point for an automated bounds checking compiler. CCured attempts to elide bounds checks where it can statically prove that they are not required, whereas CHERI hardware enforcement uses them everywhere. Additionally, the CCured version is not thread safe, as the checking introduces potential races, as described by the CCured documentation, whereas the CHERI hardware bounds checks are safe because all descriptor updates are atomic.

Figure 7.5 shows the time for the benchmarks to run. For each benchmark, the time is split into the time taken for each of the major computational steps. The error bars show one standard deviation in the timing results.

In the bisort benchmark, CHERI experiences a very small overhead while constructing the tree. This process involves walking the tree in a linear fashion and allocating memory for each node. In CHERI, this requires one extra instruction in the allocation to track set bounds, but in the software-enforcement case it is significantly more complex.

The sorting phase involves walking the tree to determine the order. This is mostly dominated by cache miss time. In the cases where the nodes are in the cache, MIPS and CHERI times are almost identical, but the CHERI case increases the size of each node and so increases the probability of cache misses. Bisort nodes contain two pointers and a 64-bit integer and are 24 bytes in the unprotected case and 96 bytes in the CHERI case, but consume 128 bytes due to alignment constraints. While the working set is larger than the available cache in both cases, an unprotected node is smaller than a 32 byte cache line but a node in the CHERI case consumes two cache lines. While extremely large linked lists are rare, this case demonstrates the need for compressed segment descriptors.

[2]The maturity of our modified LLVM compiler did not enable further conversions at the time. Notably, LLVM does not yet correctly support floating point for 64-bit MIPS.

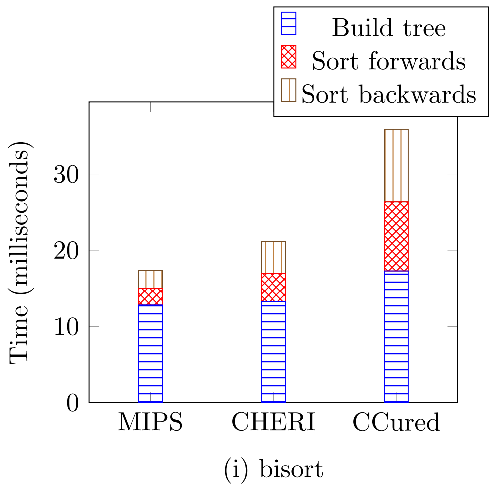


Figure 7.5: Benchmark results comparing unmodified MIPS code to software and hardware bounds checking.

The mst benchmark shows a different access pattern. Here, there are only two (contiguous) allocations and so there is little distinction between the access patterns involved in creating the data structure and in using it. When building the graph, the total run time is dominated by the hash calculations, which are arithmetic and not memory accesses, and so are the same in both cases. When computing the minimum spanning tree, as with bisort, the dominant factor is cache misses. In this case, the access pattern is a linear read which can benefit from prefetching as well as from compressed descriptors.

Figure 7.6 shows the percentage slowdown of CHERI for different input sizes on the bisort benchmark. Note that the slowdown increases as the data size increases, because we get many more cache misses. Three distinct tiers can be observed reflecting the two levels of cache in CHERI. Below 128 nodes in the linked list, both the CHERI and unprotected cases fit into the 16 kilobyte level one cache and CHERI has very little overhead. When the linked list size is between 128 and 512, the list fits into the 64 kilobyte level two cache and CHERI incurs a 15% overhead. Between 1k and 16k nodes we settle at an overhead of 26%. Peaks can be observed at 128 and 1k where where capability list overflows one level of cache but the simple pointer list does not.

7.3 Domain crossing performance study

Domain crossing performance is crucial if we are to divide programs into closely communicating domains of varying trust. The current tools used to do this are the TLB and process mechanism with inter-process communication in the operating system. Watson measured the cap_enter system call on FreeBSD at 1.22us on a 1.86 GHz Intel Xeon E5320, or 2270 cycles [87]. Takahashi *et al.* found that a highly optimised page table switch without process change on SPARC required 78 instructions [82], and that on an Intel Pentium II a base segment change (without touching the TLB) required 267 cycles without appealing to kernel mode and 700 cycles if the segment change code runs in kernel mode [76]. Considering that we would hope to move from simple function invocations within an address space to these protected procedure calls, this overhead is often prohibitive even without considering the complexity of subdividing the application.

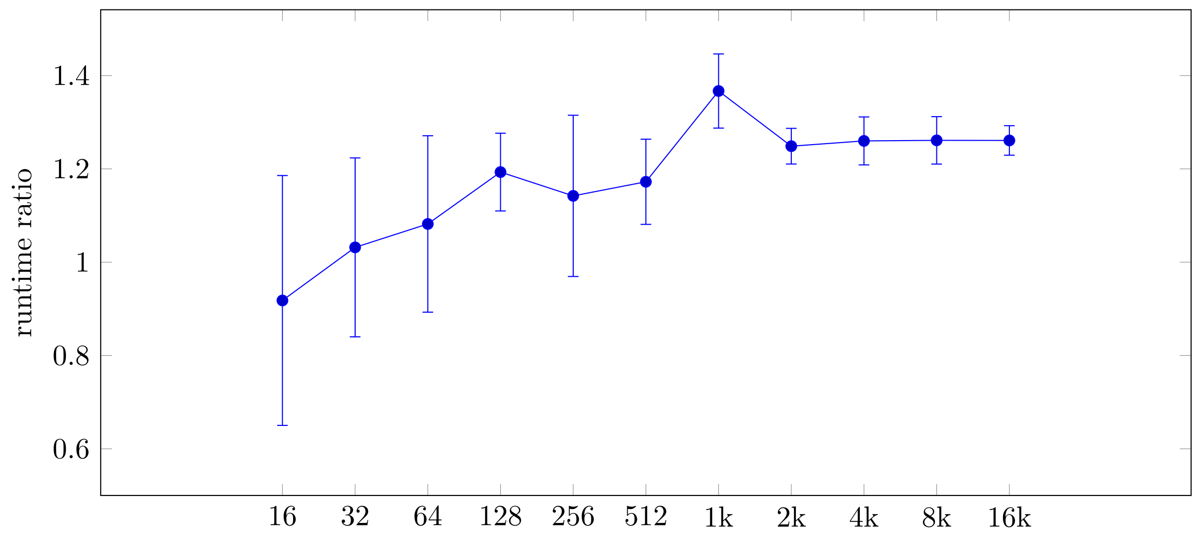


Figure 7.6: CHERI/unprotected runtime ratio for increasing list sizes in bisort.

A protection domain in CHERI is defined by the capability register file, so to change protection domains is to change register sets. This would imply that a protection domain crossing would have a similar function to the setjump/longjump mechanism in the C runtime. However, because the register file in the CHERI model is compiler managed, the compiler may use a simple capability ABI which uses fewer capability segment registers and thus reduces the cost of protection domain crossing.

7.3.1 Domain crossing example

I have constructed a minimal domain crossing example. An ambient domain calls into a very limited domain that performs an operation (an XORI with the operand) and returns the result. The ambient domain does not trust the sandboxed domain, though the models enforce this distrust to varying degrees. The four domain crossing strategies are listed below and example code is given in Figure 7.7:

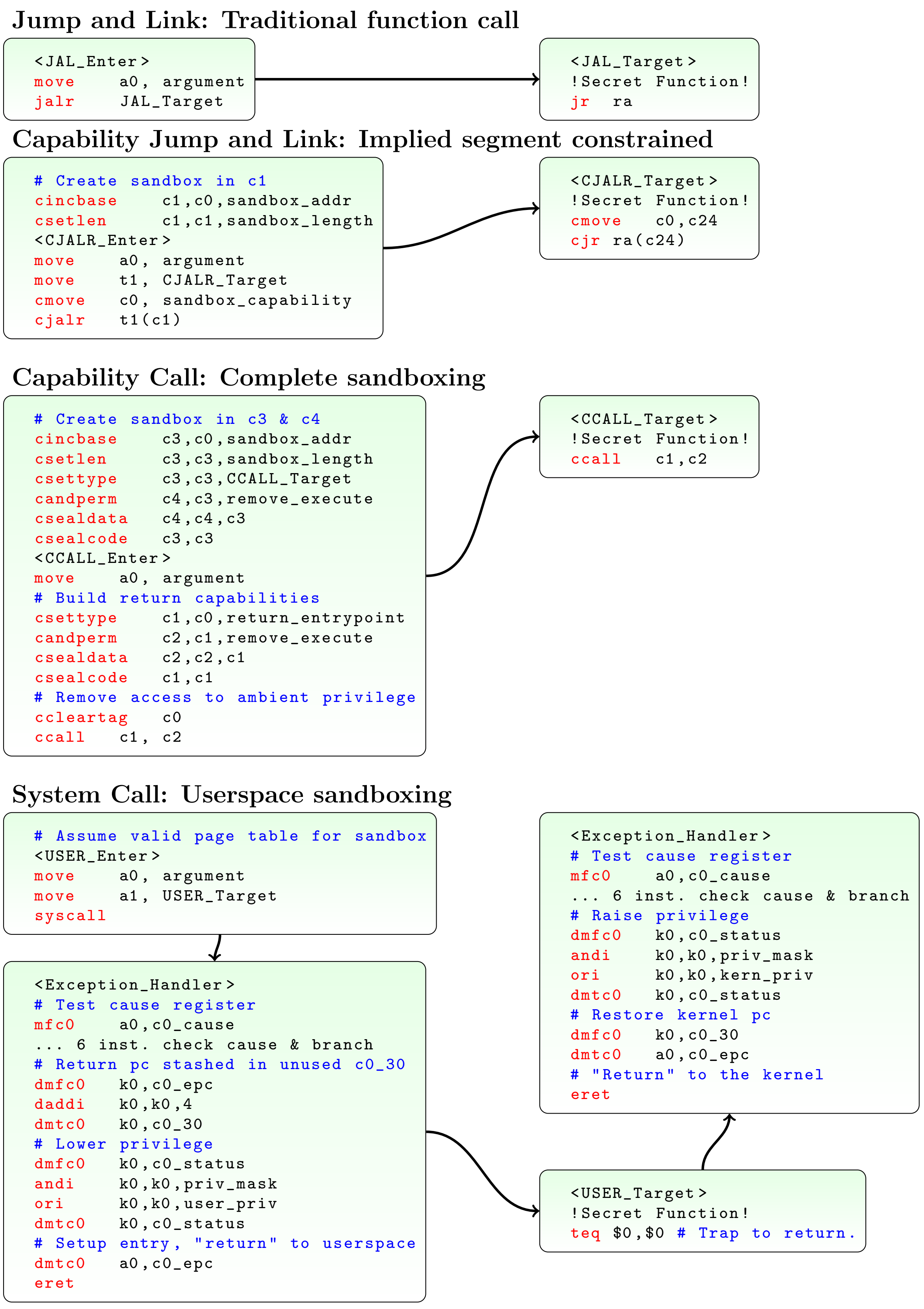
JAL: Jump and Link — This case is a traditional function call with no protection.

CJALR: C0 Constrained — This case uses the capability jump and link instruction to change PCC and C0, the capability segment register through which all general-purpose accesses are constrained. Capability jumps can't make a new protection domain available, that is, it does not grant access to capabilities that were not previously available to the domain. As a result, the calling domain must not revoke access to any capabilities in its domain when performing the jump if it expects to retain access upon return. Nevertheless, a binary might be statically proven to have no capability instructions besides the capability jump for the return. If the sandboxed executable is free of extraneous coprocessor 2 instructions, the executable cannot touch addresses outside of its sandbox.

CCALL: Capability Call — This case uses *CCALL*, the safe capability domain transition instruction. An executable and a data capability are sealed with the same type and unsealed only on a *CCALL* which takes the matching pair as operands. The caller must seal and pass the return capabilities as arguments to the invoked domain. This model allows clearance of all capabilities that belong to the callers' domain while still allowing a return. This facilitates arbitrary capability operations in either protection domain without compromising the protection model.

SYSCALL: Userspace Sandbox — This case puts the protected domain in userspace. The caller runs in kernel space but uses an custom exception handler to transition into the sandbox and back out to the caller. This case has been highly optimised with the return address being stashed in an unused control register and all unused functions removed from the exception handler.

These four scenarios have been tested to three memory safety standards, memory safety, integrity, and confidentiality. For the **memory safety** standard, we only insure that no memory references can be made to addresses outside of the protection domain (in the cases that support this). For the **integrity** standard, we additionally save the state of our used registers and restore them on return to ensure that the untrusted domain cannot corrupt the execution in the trusted domain. This is essentially a *callee-save* policy for safety. For the **confidentiality** standard, we additionally require that all register values produced in the privileged protection domain be zeroed before the domain crossing, preventing the unintended leakage of information.



Jump and Link: Traditional function call

```
<JAL_Enter>
move     a0, argument
jalr     JAL_Target
```

```
<JAL_Target>
!Secret Function!
jr  ra
```

Capability Jump and Link: Implied segment constrained

```
# Create sandbox in c1
cincbase     c1,c0,sandbox_addr
csetlen      c1,c1,sandbox_length
<CJALR_Enter>
move     a0, argument
move     t1, CJALR_Target
cmove    c0, sandbox_capability
cjalr    t1(c1)
```

```
<CJALR_Target>
!Secret Function!
cmove    c0,c24
cjr ra(c24)
```

Capability Call: Complete sandboxing

```
# Create sandbox in c3 & c4
cincbase     c3,c0,sandbox_addr
csetlen      c3,c3,sandbox_length
csettype     c3,c3,CCALL_Target
candperm     c4,c3,remove_execute
csealdata    c4,c4,c3
csealcode    c3,c3
<CCALL_Enter>
move     a0, argument
# Build return capabilities
csettype     c1,c0,return_entrypoint
candperm     c2,c1,remove_execute
csealdata    c2,c2,c1
csealcode    c1,c1
# Remove access to ambient privilege
ccleartag    c0
ccall    c1, c2
```

```
<CCALL_Target>
!Secret Function!
ccall    c1,c2
```

System Call: Userspace sandboxing

```
# Assume valid page table for sandbox
<USER_Enter>
move     a0, argument
move     a1, USER_Target
syscall
```

```
<Exception_Handler>
# Test cause register
mfc0     a0,c0_cause
... 6 inst. check cause & branch
# Return pc stashed in unused c0_30
dmfc0    k0,c0_epc
daddi    k0,k0,4
dmtc0    k0,c0_30
# Lower privilege
dmfc0    k0,c0_status
andi     k0,k0,priv_mask
ori      k0,k0,user_priv
dmtc0    k0,c0_status
# Setup entry, "return" to userspace
dmtc0    a0,c0_epc
eret
```

```
<USER_Target>
!Secret Function!
teq $0,$0 # Trap to return.
```

```
<Exception_Handler>
# Test cause register
mfc0     a0,c0_cause
... 6 inst. check cause & branch
# Raise privilege
dmfc0    k0,c0_status
andi     k0,k0,priv_mask
ori      k0,k0,kern_priv
dmtc0    k0,c0_status
# Restore kernel pc
dmfc0    k0,c0_30
dmtc0    a0,c0_epc
# "Return" to the kernel
eret
```

Figure 7.7: Sandbox invocation loop in abridged MIPS pseudo-assembly

To benchmark these domain crossing strategies, I ran each sandboxed function 100 times on the simulated processor without an operating system. I report the average cycles for a single domain crossing. The sandbox is created outside of the inner loop so sandbox creation is amortised over the 100 invocations.

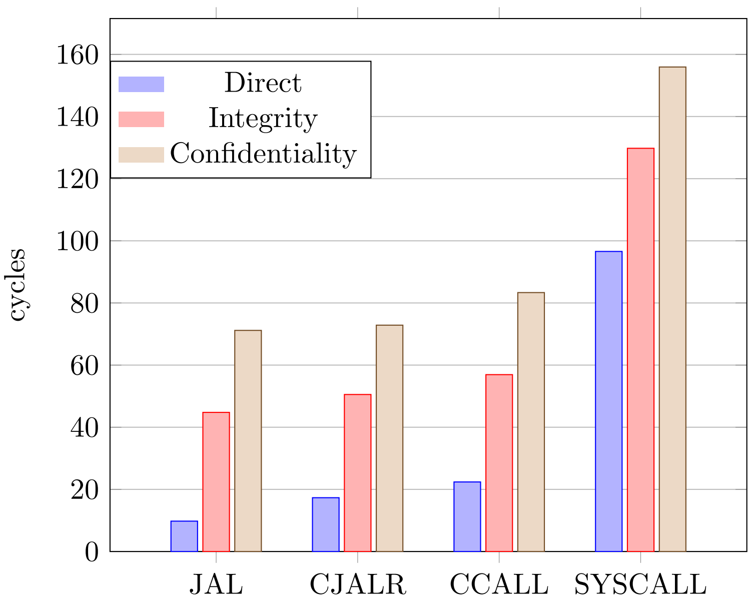


Figure 7.8: Domain crossing performance

Figure 7.8 shows the results of this benchmark. The capability protected cases are more expensive than a simple jump and link (JAL), with CJALR requiring 17.32 cycles and CCALL requiring 22.39 cycles versus 9.78 cycles for JAL. The worst of these, however, is 4× faster than the SYSCALL direct invocation which requires 96.58 cycles in the best case, roughly agreeing with Takahashi *et al.* that a RISC instruction set requires 78 instructions for a minimal domain crossing [82]. There are two key reasons for this performance gap, the generality and “exceptionality” of the exception mechanism.

Generality — The exception mechanism is a general mechanism used for many purposes and the code paths in the exception handler must disambiguate between exception causes before taking specific action. In this case, we only have two possible actions, call and return, but even these require 7 instructions to disambiguate in the handler, as seen in Figure 7.7. This set of instructions runs twice for each sandbox invocation, once to enter and once to leave the sandbox. This consumes at least 14 instructions of the loop.

Exceptionality — The exception mechanism is by nature exceptional, meaning that it should not have to be highly optimised. As a result, an exception flushes our BERI pipeline resulting in 8 dead cycles. There are actually two pipeline flushes for each use of the exception handler, one for the trap or system call and the other for the exception return (ERET). Since the exception handler is called twice for each sandbox invocation, once to enter and once to return, there is a total flush cost of 32 cycles, significantly more than for the entire secure call and return with full safety using CHERI capabilities. Furthermore, system control coprocessor register reads and writes are not fully pipelined, though BERI implements interlocks to guarantee safety with optimal performance without NOPs. Nevertheless, after careful ordering optimisations in the exception handler, 14 cycles are spent waiting for cp0 updates in every loop. Commercial exception handlers have at least 5 NOPs after every cp0 manipulation which reduces performance even further.

These problems are endemic to the fact that the exception mechanism is not the common case and must be allowed to be special and relatively unoptimised. Therefore if secure invocation is expected to be the common case, it should not use the exception mechanism.

If we require integrity and do not trust the sandbox to restore register state, the overhead for a fully protected CCALL over a simple JAL shrinks to less than 30% while the SYSCALL still has 290% overhead, though it would likely be much worse in a full operating system. If a security policy does not trust a sandbox to access the callers' memory, it is not likely that it would trust the sandbox to return registers in an uncorrupted state. Therefore the integrity case might be considered the most common case.

7.4 Conclusion

Microbenchmarks demonstrate that CHERI capability segmentation can make fine-grained bounds checking competitive with unprotected execution for a limited number of pointers. The Olden benchmarks expose the cost of larger pointers for pointer-heavy structures, though performance is still much better than software protection. Linked lists and trees which protect each node with an independent descriptor will require descriptor compression to achieve competitive performance with unprotected code. Finally, the domain crossing study demonstrates the efficiency of CHERI domain crossing instructions for memory safety while reminding us that full integrity and confidentiality imposes significant overhead for all cases.

Chapter 8

Conclusions and future work

CHERI demonstrates that capability addressing can fit well into an existing RISC instruction set and is practical for use in modern software. Capability systems, which have often stressed formal analysis over practicality, have been considered inflexible and impractical for high-performance general-purpose computing systems. Security concerns have heightened interest in high-assurance computing but proposals such as Mondrian Memory Protection and Hardbound have valued transparent backwards compatibility over practicality. As a result, industry adoption of these models has compromised security properties for practicality [43]. CHERI demonstrates that a capability segmentation model can offer strong memory safety in a prototypical RISC instruction set without exotic extensions.

8.1 Contributions

As stated in the introduction, CHERI makes four major contributions which are reviewed below.

8.1.1 Backwards compatible

The M-Machine demonstrated that capability addressing is practical in a RISC architecture. However the M-Machine did not constrain itself to compatibility with an existing architecture or memory models with the result that existing compilers and operating systems could not easily run on the machine. CHERI uses a backward-compatible 64-bit MIPS instruction set and runs the FreeBSD operating system unmodified with the thousands of applications available for this platform. CHERI allows applications to selectively create protected objects and nested protection domains in their own address spaces according to a traditional capability model. This provides applications with an incremental path to capability memory safety which is new for capability addressing machines.

8.1.2 Capabilities as a practical compiler target

CHERI also demonstrates that capabilities can be a practical compiler target. Classical capability machines used large allocation sized capabilities for addressing, including the M-Machine which did not allow segment descriptor modification by the program. The Cyclone compiler, and indeed Java and C#, have shown that fat pointer implementations are a natural compiler target. CHERI adopts a fat pointer model for its capability segment descriptors and inherits this compiler work, allowing software fat pointer implementations to directly translate to CHERI capability segment descriptors. I have presented benchmarks compiled with a modified version of CLANG with LLVM which use capability segment descriptors in place of pointers with only annotations added to the source code. This work demonstrates that capability descriptors (*i.e.* unforgeable pointers) are useful and natural in modern program and compiler structure.

8.1.3 Capabilities with scalable performance

CHERI implements a scalable mechanism that can enforce both bounds on program objects and sandboxes for program components. Mondrian Memory Protection enforces sandboxes efficiently and Hardbound enforces object bounds, but neither scales to the other extreme. CHERI uses a special segment register to constrain general-purpose references as well as a segment register file for explicit object references. This arrangement enables a broad range of memory safety models.

8.1.4 Uncompromising safety properties

Despite providing a flexible low-level primitive, CHERI does not compromise capability integrity. CHERI capabilities are unforgeable and are protected in registers and in memory. As a result, a process cannot reference memory for which it does not posses a capability. These strongly enforced memory safety properties provide a foundation for compiling formally verifiable programs. The strong capability mechanism also enables non-addressing uses of unforgeable tokens such as file descriptor capabilities.

8.1.5 Result of Contributions

As a result of these contributions, CHERI is practical in hardware and useful to modern software. The CHERI model is suitable for adoption by modern RISC architectures with only a few optimisations.

8.2 Future Optimisations

The development of CHERI has also uncovered bottlenecks which should be addressed for further revisions and for commercial implementations.

8.2.1 Compressed descriptors

The large size of CHERI descriptors is the main overhead for fine-grained capability protection. While the CHERI capability model has a very small overhead in terms of instruction count, memory overhead can be significant. Quadrupling pointer size and requiring 32-byte alignment increases cache misses for pointer-heavy data structures.

I would propose to introduce a compressed capability format. While CHERI capabilities could be compressed into 128-bits losing very little accuracy due to the 40-bit MIPS virtual address space, the Hardbound results demonstrate that pointers to small objects can be used in a large proportion of cases. This would suggest that a 64-bit segment descriptor with, for example, 8 bits of length in addition to a full base and a subset of permissions could greatly reduce overhead in the common case. The instruction set would need only to be extended with new instructions to load and store the smaller descriptor which would throw an exception if the capability cannot be perfectly represented in the restricted format. Because these compressed capabilities would have a limited size, the compiler could only use compressed capabilities for objects allocated with a static size, though this is a common case.

Compressed capabilities would also require additional tags. For example, 64-bit capabilities would require five bits for each 32-byte line: the original tag that identifies a 256-bit capability as well as four tags that indicate valid 64-bit capabilities. It would not be possible to have both a valid 256-bit capability as well as any 64-bit capabilities, so 16 states would be left undefined for possible future type-checking extensions.

8.2.2 Register validity tags

The largest cost of confidential domain crossings is register invalidation[1]. Validity tags for capability and general-purpose registers would accelerate register invalidation. A single 32-bit structure could determine whether each register should have a null value or the current state of the register. An instruction with an immediate mask could invalidate most of the registers in a register file in a single cycle without actually writing to the storage for each register.

8.2.3 Domain crossing stack

Domain crossing may require a trusted stack to be useful to software. If this proves true, full user-space domain crossing will require pushing return capabilities on the trusted stack in hardware. This model is possible in two cycles on our MIPS implementation, only slightly stretching the definition of a RISC instruction set.

8.3 Conclusion

Approximate memory safety is no longer sufficient for computer systems which we entrust with our possessions, privacy, and physical safety. While arithmetic in microprocessors has not tolerated approximate correctness, memory protection has been considered good enough with rough page granularity and with no discernment between references within an address space. This open-door policy may have been acceptable when networked computers were a “small-town” community, but its ramifications are becoming ever more serious as the global community of networked computers swells with valuable information and with obfuscating complexity that provides anonymity to unscrupulous interests.

CHERI capability segmentation provides a path from existing processors and systems to a new landscape of fine-grained, principled memory protection enforcing computer security. While it is possible to redesign computer systems from the ground up for security, such an endeavour is likely to sacrifice many hard-learned lessons in performance, usability, and even security. CHERI respects precedent in processor architecture design by using only constructs that are well understood and easy to optimise. The CHERI ISA is fully

[1]Integrity has a lower cost because only a subset of registers must be preserved and restored

backward compatible to ensure that programs and operating systems can inherit the most effective solutions for performance and usability while incrementally adopting strong and principled security where it is required. In 20 years, something very similar to the CHERI memory safety model is likely to be prevalent, indeed iMPX from Intel has begun the transition, and the CHERI prototype and open-source platform will be crucial for understanding exactly what that future should look like and how we will get there.

Bibliography

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. Technical report, Computer Science Department, Carnegie Mellon University, August 1986.

[2] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX conference on Security*, pages 12–12. USENIX Association, 2010.

[3] T Alves and D Felton. ARM TrustZone: Integrated hardware and software security. *Information Quarterly*, 3(4), July 2004.

[4] Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Michael van Eeten, Michael Levi, Tyler Moore, and Stefan Savage. Measuring the cost of cybercrime. In *WEIS*, 2012.

[5] ARM Limited, ARM Limited, Cambridge, UK. *Cortex-R4 and Cortex-R4F Technical Reference Manual*, r1p3 edition, November 2009.

[6] Sundeep Bajikar. Trusted platform module (TPM) based security on notebook PCs - white paper. *Mobile Platforms Group, Intel Corporation (June 20, 2002)*, 2002.

[7] Richard Barber. Hacking techniques: The tools that hackers use, and how they are evolving to become more sophisticated. *Computer Fraud & Security*, 2001(3):9–12, 2001.

[8] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[9] A. Bensoussan, C.T. Clingen, and R.C. Daley. The Multics virtual memory: concepts and design. *Communications of the ACM*, 15(5):308–318, 1972.

[10] V. Berstis. Security and protection of data in the IBM System/38. In *Proceedings of the 7th annual symposium on Computer Architecture*, pages 245–252. ACM, 1980.

[11] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Version 3.8 Reference Guide*, November 2004.

[12] Martin Christopher Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines.* PhD thesis, Princeton University, June 1996.

[13] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. *SIGPLAN*, 29(11):319–327, November 1994.

[14] M. Castro, M. Costa, J.P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58. ACM, 2009.

[15] Gregory A. Chadwick. Communication centric, multi-core, fine-grained processor architecture. Technical Report UCAM-CL-TR-832, University of Cambridge, Computer Laboratory, April 2013.

[16] Gregory A Chadwick and Simon W Moore. Mamba: A scalable communication centric multi-threaded processor architecture. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 277–283. IEEE, 2012.

[17] R.E. Childs Jr, J. Crawford, D.L. House, and R.N. Noyce. A processor family for personal computers. *Proceedings of the IEEE*, 72(3):363–376, 1984.

[18] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *ACM SIGOPS Operating Systems Review*, volume 33, pages 140–153. ACM, 1999.

[19] W. Chuang, S. Narayanasamy, B. Calder, and R. Jhala. Bounds checking with taint-based analysis. *High Performance Embedded Architectures and Compilers*, pages 71–86, 2007.

[20] M. Cierniak, G.Y. Lueh, and J.M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *ACM SIGPLAN Notices*, volume 35, pages 13–26. ACM, 2000.

[21] A. Coglio, A. Goldberg, and Zhenyu Qian. Toward a provably-correct implementation of the JVM bytecode verifier. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 403–410 vol.2, 2000.

[22] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming.* Academic Press Ltd., 1972.

[23] D. Dean, E.W. Felten, and D.S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 Symposium on Security and Privacy*, pages 190–200, Oakland, California, May 1996. IEEE Computer Society.

[24] Andre DeHon, Ben Karel, Jr Thomas F. Knight, Gregory Malecha, Benoit Montagu, Robin Morisset, Greg Morrisett, Benjamin C. Pierce, Randy Pollack, Sumit Ray, Olin Shivers, Jonathan M. Smith, and Gregory Sullivan. Preliminary design of the SAFE platform. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS 2011)*, October 2011.

[25] P.J. Denning. Virtual Memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[26] J.B. Dennis. Segmentation and the Design of Multiprogrammed Computer Systems. *Journal of the ACM (JACM)*, 12(4):589–602, 1965.

[27] J.B. Dennis and E.C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, 1966.

[28] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. *SIGARCH Computer Architecture News*, 36(1):103–114, March 2008.

[29] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. ACM.

[30] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA, 2006. ACM.

[31] DM England. Capability concept mechanism and structure in system 250. In *Proceedings of the International Workshop on Protection in Operating Systems*, 1974.

[32] R.J. Feiertag and P.G. Neumann. The foundations of a Provably Secure Operating System (PSOS). In *Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press, 1979. `http://www.csl.sri.com/neumann/psos.pdf`.

[33] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering*, 2011.

[34] Freescale Semiconductor. *MPC5744P Product Brief*, 2nd edition, March 2012.

[35] S. Hangal and M.S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM, 2002.

[36] P.M. Hansen, M.A. Linton, R.N. Mayo, M. Murphy, and D.A. Patterson. A performance evaluation of the intel iAPX 432. *ACM SIGARCH Computer Architecture News*, 10(4):17–26, 1982.

[37] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual.* MIPS Technologies, Inc.

[38] James C. Hoe and Arvind. Operation-Centric Hardware Description and Synthesis. *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, 23(9), September 2004.

[39] James C Hoe and A Arvind. Hardware synthesis from term rewriting systems. In *VLSI*, pages 595–619. Citeseer, 1999.

[40] M.E. Houdek, F.G. Soltis, and R.L. Hoffman. IBM System/38 Support for Capability-based Addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 341–348. IEEE Computer Society Press, 1981.

[41] C.B. Hunter, J.F. Ready, and E. Farquhar. *Introduction to the Intel iAPX 432 architecture.* Reston Pub. Co.(Reston, Va.), 1985.

[42] Intel Corporation. *IA-32 Architectures Optimization Reference Manual*, April 2012.

[43] Intel Plc. Introduction to Intel Memory Protection Extensions. `http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`, July 2013.

[44] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *Micro, IEEE*, 18(4):60–75, 1998.

[45] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *ATEC '02: Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[46] Poul-Henning Kamp. Please put openssl out of its misery. *Queue*, 12(3):20, 2014.

[47] P.A. Karger. *Improving Security and Performance for Capability Systems.* PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, England, October 1988. Technical Report No. 149.

[48] Tom Kilburn, David J. Howarth, R.B. Payne, and Frank H. Sumner. The Manchester University Atlas operating system part I: internal organization. *The Computer Journal*, 4(3):222–225, 1961.

[49] Nir Kshetri. Positive externality, increasing returns, and the rise in cybercrimes. *Commun. ACM*, 52(12):141–144, December 2009.

[50] L. Lam and T. Chiueh. Checking array bound violation using segmentation hardware. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 388–397. IEEE, 2005.

[51] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[52] Ruby B Lee, Peter CS Kwan, John P McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings 32nd International Symposium on Computer Architecture, 2005*, pages 2–13. IEEE, 2005.

[53] Henry M. Levy. *Capability-Based Computer Systems.* Butterworth-Heinemann, Newton, MA, USA, 1984.

[54] Jochen Liedtke. On microkernel construction. In *SOSP'95: Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, December 1995.

[55] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *SOSP 2011: Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[56] Microsoft. The BlueHat prize contest offical rules. `http://www.microsoft.com/security/bluehatprize/rules.aspx`, 2014.

[57] Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. The MIPS R4000 processor. *IEEE Micro*, 12(2):10–22, 1992.

[58] George C Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128–139. ACM, 2002.

[59] R.M. Needham and R.D.H. Walker. The Cambridge CAP Computer and its Protection System. *Operating Systems Review*, pages 1–10, 1977.

[60] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980. 2nd edition, Report CSL-116.

[61] P.G. Neumann, R.S. Fabry, K.N. Levitt, L. Robinson, and J.H. Wensley. On the design of a provably secure operating system. In *Proceedings of the International Workshop On Protection in Operating Systems*, pages 161–175, August 1974.

[62] P.G. Neumann and R.J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, pages 208–216, Las Vegas, Nevada, December 2003. IEEE Computer Society.

[63] P.G. Neumann, L. Robinson, K.N. Levitt, R.S. Boyer, and A.R. Saxena. A provably secure operating system. Technical report, Computer Science Laboratory SRI International, Menlo Park, California, 13 June 1975.

[64] E.I. Organick. *The Multics system: An examination of its structure.* MIT Press, Cambridge, Massachusetts, 1972.

[65] P. Parrend and S. Frénot. Classification of Component Vulnerabilities in Java Service Oriented Programming (SOP) Platforms. *Component-Based Software Engineering*, pages 80–96, 2008.

[66] Nathanael Paul and David Evans. Comparing Java and .NET security: Lessons learned and missed. *computers & security*, 25(5):338–350, 2006.

[67] Fred J Pollack, George W Cox, Dan W Hammerstrom, Kevin C Kahn, Konrad K Lai, and Justin R Rattner. Supporting ADA memory management in the iAPX-432. In *ACM SIGARCH Computer Architecture News*, volume 10, pages 117–131. ACM, 1982.

[68] Daniel L. Rosenband and Arvind. Modular Scheduling of Guarded Atomic Actions. In *Proceedings of DAC'04*, San Diego, CA, 2004.

[69] Daniel L. Rosenband and Arvind. Hardware Synthesis from Guarded Atomic Actions with Performance Specifications. In *Proceedings of ICCAD'05*, San Jose, CA, 2005.

[70] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[71] M.D. Schroeder. Engineering a security kernel for Multics. In *SOSP '75: Proceedings of the fifth ACM Symposium on Operating Systems Principles*, pages 25–32, New York, NY, USA, 1975. ACM.

[72] M.D. Schroeder and J.H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3), March 1972.

[73] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, volume 12, 2012.

[74] J.S. Shapiro, Jonathan Smith, and David Farber. EROS: A fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM Symposium on Operating Systems Principles*, Dec 1999.

[75] J. Shepherd. Principal design features of the multi-computer. *ICR Quarterly Report*, 19, November 1968.

[76] T. Shinagawa, K. Kono, and T. Masuda. Fine-grained protection domain based on segmentation mechanism. *Japan Society for Software Science and Technology*, 2000.

[77] Howard Shrobe, Thomas Knight, and Andre de Hon. TIARA: trust management, intrusion tolerance, accountability, and reconstitution architecture. Technical Report MIT-CSAIL-TR-2007-028, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Lab, May 2007.

[78] Sean W Smith, Elaine R Palmer, and Steve Weingart. Using a high-performance, programmable secure coprocessor. In *Financial Cryptography*, pages 73–89. Springer, 1998.

[79] Sean W Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.

[80] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGPLAN Notices*, 39(11):85–96, October 2004.

[81] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, 2013.

[82] M. Takahashi, K. Kono, and T. Masuda. Efficient kernel support of fine-grained protection domains for mobile code. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pages 64–73. IEEE, 1999.

[83] Trimaran project website. `http://www.trimaran.org/`, 2014.

[84] Rodrigo A. Vivanco and Nicolino J. Pizzi. Scientific computing with Java and C++: a case study using functional magnetic resonance neuroimages. *Software: Practice and Experience*, 35(3):237–254, 2005.

[85] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2):167–187, 1996.

[86] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, 1993. ACM.

[87] R.N.M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for Unix. In *Proceedings of the 19th USENIX Security Symposium*. USENIX, August 2010.

[88] R.N.M. Watson, P.G. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S.W. Moore, S.J. Murdoch, P. Paeps, et al. CHERI: A research platform deconflating hardware virtualization and protection. In *Workshop paper, Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, 2012.

[89] N.J. Wheatley and M.P. Andrews. Memory protection system using capability registers, October 4 1983. US Patent 4,408,274.

[90] M.V. Wilkes and R.M. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier North Holland, New York, 1979.

[91] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. *SIGARCH Computer Architecture News*, 30(5):304–316, October 2002.

[92] Emmett Witchel, Josh Cates, and Krste Asanović. *Mondrian memory protection*, volume 37. ACM, 2002.

[93] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[94] J. Woodruff, R.N.M. Watson, D. Chisnall, S.W. Moore, Anderson J., B. Davis, B. Laurie, R. Neumann, P.G. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings 41st International Symposium on Computer Architecture, 2014*, 2014.

[95] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974.

[96] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[97] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *Proceedings 31st International Symposium on Computer Architecture, 2004*, pages 224–235. IEEE, 2004.

Appendix A

64-bit MIPS adaptations for limit study

A.1 Mondrian MIPS64

The Mondrian memory protection (MMP) model creates a segment for each allocated object and populates a table with the protection information for each word in memory. The table is hierarchical, so a large region of uniform permissions can be recorded directly in a higher level in the table while regions with complex permissions must populate leaf nodes.

Our MMP model is extended to 64-bits from the original 32-bit implementation. This short description assumes a detailed knowledge of the original MMP model. I used the vector table model. Our processor has only 40-bit addresses, so I extended the first two table indices to 14 bits from 10 to preserve the granularity of the crucial mid-level and leaf nodes. Records have been extended to 64-bits from 32 since pointers are now 64-bits. Intermediate nodes can now record permissions for 16 nodes rather than 8, so a directly encoded mid-level node describes a 4096-byte entry in 256-byte regions. We protect 64-bit words rather than 32-bit to match the 64-bit architecture. We assume a hardware read of the table with the entire node being cached in the memory protection table, 4096 bytes for a mid-level node and 256-bytes for leaf-level nodes. Due to the hardware read of the table, we do not implement *side-car registers* which require knowledge of the entirety of the originally allocated region which would require a more complex table scanner than is typically implemented for a critical path lookup. Without the more complex table scan, the side-car registers do not affect performance.

My MMP table fill algorithm is modelled as software. I have implemented a minimal table fill algorithm in C and compiled with optimisation. The simulator uses instruction counts from this implementation to estimate instruction cost of each stage of the table fill algorithm.

A.2 Hardbound MIPS64

Our 64-bit MIPS implementation of hardbound extends base and bound information to 64-bits rather than 32-bits used in the original simulated IA-32 model. We continue to use a direct offset for the table and model a 128-bit load or store of base and bound information for every load or store of a pointer. We also model the memory use of the tag table.

A.3 iMPX Table MIPS64

The iMPX table approach translates directly to a 64-bit MIPS model.

A.4 iMPX Fat-pointer MIPS64

Fat pointer protection using an iMPX-style approach also translates directly to 64-bit MIPS.

A.5 Soft Fat-pointers MIPS64

Our soft fat-pointer model uses the same model as the iMPX fat-pointer model but performs the load and store of the bounds information as two loads and stores as well as a bounds check using standard instructions.

A.6 CHERI

CHERI uses a simple fat-pointer model, loading and storing a full capability for every pointer load or store. CHERI requires two extra instructions for pointer creation, one to set the base and another for the length, but pointer use does not incur instruction overhead. We present two models of CHERI, one with 256-bit capabilities, which is the model we have implemented in this work, and another with 128-bit capabilities. 128 bit capabilities are possible with nearly no information loss due to the 40-bit virtual address space in MIPS, so I would expect commercial adaptations to use at least this level of optimisation. The 256-bit case incurs a 24-byte overhead for loads and stores of pointers and the 128-bit case incurs an 8-byte overhead.