Technical Report

UCAM-CL-TR-936
ISSN 1476-2986

Number 936

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

High-performance memory safety: optimizing the CHERI capability machine

Alexandre J. P. Joannou

May 2019

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

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

This technical report is based on a dissertation submitted September 2017 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Peterhouse College.

ISSN 1476-2986

Abstract

High-performance memory safety Optimizing the CHERI capability machine

Alexandre Jean-Michel Procopi Joannou

This work presents optimizations for modern capability machines and specifically for the CHERI architecture, a 64-bit MIPS instruction set extension for security, supporting fine-grained memory protection through hardware enforced capabilities.

The original CHERI model uses 256-bit capabilities to carry information required for various checks helping to enforce memory safety, leading to increased memory bandwidth requirements and cache pressure when using CHERI capabilities in place of conventional 64-bit pointers. In order to mitigate this cost, I present two new 128-bit CHERI capability formats, using different compression techniques, while preserving C-language compatibility lacking in previous pointer compression schemes. I explore the trade-offs introduced by these new formats over the 256-bit format. I produce an implementation in the L3 ISA modelling language, collaborate on the hardware implementation, and provide an evaluation of the mechanism.

Another cost related to CHERI capabilities is the memory traffic increase due to capability-validity tags: to provide unforgeable capabilities, CHERI uses a tagged memory that preserve validity tags for every 256-bit memory word in a shadowspace inaccessible to software. The CHERI hardware implementation of this shadowspace uses a capability-validity-tag table in memory and caches it at the end of the cache hierarchy. To efficiently implement such a shadowspace and improve on CHERI's current approach, I use sparse data structures in a hierarchical tag-cache that filters unnecessary memory accesses. I present an in-depth study of this technique through a Python implementation of the hierarchical tag-cache, and also provide a hardware implementation and evaluation. I find that validity-tag traffic is reduced for all applications and scales with tag use. For legacy applications that do not use tags, there is near zero overhead.

Removing these costs through the use of the proposed optimizations makes the CHERI architecture more affordable and appealing for industrial adoption.

Acknowledgements

This work was made possible thanks to many people here at the University of Cambridge Computer Laboratory. Specifically, I would like to thank:

- Professor Simon Moore, my supervisor, for his advice which allowed me to set relevant goals in my work, as well as for his patience and technical help for any of the details of my work that I wanted to discuss.
- Robert Watson for his guidance and also for his patience when spending some of his precious time explaining software concepts that I would ask him about.
- Jonathan Woodruff for making CHERI a reality and enabling me and all of us in the CTSRD project to work with this unique platform, and of course for sitting next to me for a significant share of the past four years and still put up with me when debating, both on exciting technical ideas and more existential questions as well.

I am also grateful to Robert Mullins from the University of Cambridge Computer Laboratory and Osman Unsal from the Barcelona Supercomputer Center for spending time reading through my work and providing useful feedback. I greatly enjoyed the discussions we had and believe that their suggested corrections made this work better.

I want to thank Matthew Naylor, Theo Markettos, Hongyan Xia, Robert Kovacsics, Colin Rothwell, Mike Roe, David Chisnall, Khilan Gudka, Brooks Davis and the rest of the CTSRD team who have always been supportive and available, and a source of excellent technical knowledge.

Thank you to Anthony Fox and Kyndylan Nienhuis for our collaborations on the CHERI L3 model which helped improving our confidence in the CHERI ISA.

Thank you to Alan Mujumdar, Niall Murphy, James Snee, Robert Norton-Wright and Alex Horsman, who have been fellow students in the past, and are still great friends who supported my efforts whether from the Computer Laboratory or from further away.

For having triggered in me the excitement I now feel towards the field of computer architecture, I want to thank the ALSOC team at LIP6 back in Paris, specifically Quentin Meunier, Alain Greiner, Pirouz Bazargan Sabet, Franck Wajsbürt and Joël Porquet.

Thank you to my friends Emmanuel Marié and Thomas Dédès who I had the joy to meet again after our time together at LIP6, and with whom I shared many great moments both back home and in the UK.

I also want to thank the people I met during my time in the UK and outside my field of study who where always there to share with me the music, the letter cutting, the bouldering, the TV-shows and movies, the card and board games.

Finally, I want to thank my family. Their love and support has always been present and continues to be despite me living in another country.

Merci Thomas, Jean-Max, Tina, Olivier, Louis et Byron. Même si pas techniquement de ma famille, c'est tout comme!

Merci Zaza, Jean-Pierre, Fafa, Didier, Magalie, Éléa, Mathis et Pauline.

Merci Papa, Athena, Fivos, Wendy.

Merci Maman, Victoria, Papi.

Je vous aime.

Merci Mamée et Mamie, je vous aime, vous me manquez.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Memory safety is one of the most important aspects of computer security: memory logically owned by one party being accessed by another is problematic for security sensitive systems. Despite all efforts in writing secure code, memory safety issues such as buffer overflow bugs still exist, and enable exploits leading to OpenSSL Heartbleed [15, 28, 37], Android StageFright [29], or the WannaCry ransomware attacks [30] and many others, with real world consequences directly compromising individual and company financial integrity.

A constantly growing code base only leads to a greater attack surface, and makes it impractical to fix all such bugs. Reducing this attack surface can be done by following the principle of least privilege [75] and using compartmentalization, confining the harm exploits can do. Modern computer systems have been tackling these issues principally with the help of page-based memory management units. The relatively coarse granularity of this approach is today supplemented by software measures, in the form of managed languages and other runtime policy enforcements that are detrimental to the system's performance.

Preventing memory safety issues from occurring is necessary and should be inexpensive. CHERI [91, 95, 99, 100] is a modern computer system that was developed in response to these concerns. It builds into its architecture new memory primitives that enable fine grained memory safety. In this work, I try to reduce CHERI's costs to make it a high performance fine grained memory safe computer system. In order to make the CHERI approach appealing as a real world commercial solution, I hypothesize that it is possible to reduce the capability primitive's memory footprint (in **Chapter 4**), and that an efficient tagged memory system can be implemented using off the shelf memory (in **Chapter 5**).

1.1 Contributions

To perform the measurements and various observations required for completion of this work, I achieve several milestones.

I propose in **Chapter 4** a framework to compress memory region descriptors, using several novel techniques. These techniques enable encoding formats that are more compact than any published hardware fat pointer or region encoding format. I apply these technique to derive a new compressed CHERI format with half the memory footprint of the original CHERI capability.

In **Chapter 5** I propose a method to optimize caching of capability validity tags. It exploits the in memory distribution pattern of these tags[1] through a hierarchical tag table, and caches all levels of that table in a dedicated last-level tag cache at the interface with the DRAM. This method makes CHERI the processor with the most efficient tagged memory implementation compatible with standard memory.

I develop a CHERI architectural model in the L3 formal specification language as an extension to Anthony Fox's existing MIPS model. It is the first formal model of a capability processor capable of booting a full operating system and of running capability code and that can be used for architectural exploration and formal proof of architectural properties.

I design and implement a toolkit for precise monitoring of processor internal events, not generally available to existing hardware, in RTL rather than simulation only. This toolkit enables easy benchmarking on a large scale and with real hardware on FPGA.

I demonstrate the applicability of continuous integration techniques commonly used in software projects to the open source hardware workflow. I implement in the Jenkins framework a suite of jobs that build CHERI hardware using continuous integration to routinely test the processor and run benchmarks on the actual hardware, also leveraging the event monitoring toolkit.

1.2 Publications

- Watson, Robert N. M., Neumann, Peter G., Woodruff, Jonathan, Roe, Michael, Anderson, Jonathan, Baldwin, John, Chisnall, David, Davis, Brooks, Joannou, Alexandre, Laurie, Ben, Moore, Simon W., Murdoch, Steven J., Norton, Robert, Son, Stacey and Xia, Hongyan. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6).* Tech. rep. UCAM-CL-TR-907. University of Cambridge, Computer Laboratory, Apr. 2017. (and previous versions [93, 94])

- Watson, R. N. M., Norton, R. M., Woodruff, J., Moore, S. W., Neumann, P. G., Anderson, J., Chisnall, D., Davis, B., Laurie, B., Roe, M., Dave, N. H.,

[1]Relying on the in memory patterns of the tags make this approach generalizable to kinds of tags other than capability validity tags that also expose sparse in memory patterns

Gudka, K., Joannou, A., Markettos, A. T., Maste, E., Murdoch, S. J., Rothwell, C., Son, S. D. and Vadera, M. 'Fast Protection-Domain Crossing in the CHERI Capability-System Architecture'. In: *IEEE Micro* 36.5 (Sept. 2016), pp. 38–49

- Chisnall, David, Davis, Brooks, Gudka, Khilan, Brazdil, David, Joannou, Alexandre, Woodruff, Jonathan, Markettos, A. Theodore, Maste, J. Edward, Norton, Robert, Son, Stacey, Roe, Michael, Moore, Simon W., Neumann, Peter G., Laurie, Ben and Watson, Robert N.M. 'CHERI JNI: Sinking the Java Security Model into the C'. in: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 569–583.

- Joannou, Alexandre, Woodruff, Jonathan, Kovacsics, Robert, Moore, Simon W., Bradbury, Alex, Xia, Hongyan, Watson, Robert N. M., Chisnall, David, Roe, Michael, Davis, Brooks, Napierala, Edward, Baldwin, John, Gudka, Khilan, Neumann, Peter G., Mazzinghi, Alfredo, Richardson, Alex, Son, Stacey and Markettos, A. Theodore. 'Efficient Tagged Memory'. In: *35th IEEE International Conference on Computer Design.* Nov. 2017

1.3 Dissertation overview

The rest of this document focuses on CHERI and the improvements that can be made to it to make it a high performance memory safe computer.

In **Chapter 2** I present the context in which the CHERI processor exists, and give more information on the motivations behind this project through an overview of existing previous work on security in computer systems, and in the field of capability machines.

In **Chapter 3** I present the CHERI project itself. I explain in more details the project's framework and relevant tools, and present the current version of the CHERI processor. I evaluate this current version in order to characterise a base case against which to compare later optimizations.

In **Chapter 4** I propose a compressed 128-bit version of the 256-bit CHERI capability format that aims at minimizing its memory footprint and reducing cache pressure. I first present a naive approach to implementing such a format and then enrich it with ideas borrowed from other pointer compression schemes. I conduct an evaluation of the compressed mechanism and present the results. I also propose a generalisation of the "posits" mechanism for even further compression.

In **Chapter 5** I propose to improve CHERI's tagged memory by using a hierarchical structure to improve CHERI's tag cache utilisation, exploiting the sparsity of capability validity tags. I present a static study of pointer distribution for which I develop a tool to identify pointers in x86 application coredumps, a dynamic study for which I implement a tag cache simulator and replay gem5 traces generated by Robert Kovacsics and Jonathan Woodruff. I implement a hardware hierarchical tag cache and evaluate it on FPGA.

Finally, in **Chapter 6** I summarize the work that has been accomplished. I present possible research avenues to further improve CHERI and make it a commercially appealing solution to better memory safety for computer systems.

Chapter 2

Background

This chapter explores the notion of memory corruption as a computer security issue, and the "eternal war in memory" (as defined in [85, 86]) that stems from those issues. It explains existing protection mechanisms with their strength and limitations, introducing modern computer systems' approach to security. It then presents the notion of a *capability* as a secure reference to memory.

2.1 The persistent challenges of memory safety

Software memory bugs can trigger unintended system behaviours. When exploited, these bugs can enable attackers to take control over the execution flow of the system and run malicious code, steal sensible data through memory leaks, or possibly falsify data. OpenSSL Heartbleed [15, 28, 37], Android StageFright [29], or the WannaCry ransomware attacks [30] are famous recent examples of such memory safety bugs with a significant impact.

The most common kind of such exploits are buffer overflows. Let us consider a piece of C code using a pointer to a buffer of allocated size 10 bytes. When reading or writing from or to a buffer in memory, an untrusted user can ask for an arbitrary offset in that buffer. The requested offset could be referencing an address past the end of that buffer (bytes 11 and onwards), which could result in accessing potentially sensitive information stored after the buffer in memory. Similarly, letting someone write these memory locations can lead to important data being overwritten. Moreover, if the data overwritten is relevant to the control-flow of the machine, it can lead to undesired execution of malicious code. Typically, if the buffer is on the program stack and the overflow corrupts the return address stored on the stack as well, the location of the code jumped back to can be controlled. The following paragraphs present examples of exploits built on these principles, and some possible mitigations.

2.1.1 Common vulnerabilities

Szekeres et al. argue that a considerable amount of commonly used software is written in programming languages such as C/C++ that "lack safety", allowing an attacker to "take full control" over the execution of the program. This is generally made possible to the attacker through memory corruption bugs which are prone to appear in low level programming languages. Their papers on "the eternal war on memory" [85, 86] point out that it is infeasible to fix all memory corruption bugs given the quantity of code that it would involve (billions of lines). They identify classes of vulnerabilities and attacks, and automated techniques that can mitigate these vulnerabilities.

Memory corruption bugs leading to potential vulnerabilities are described as one of two things: a *spatial memory safety violation* or a *temporal memory safety violation*. Spatial memory safety violations come from bugs such as buffer overflows where a pointer is made to point outside the object it is supposed to reference, e.g. wrong type conversion, lack of bounds checking, etc. Temporal memory safety violations come from bugs where pointers become *dangling*, that is they point to a deallocated region of memory, e.g. *use-after-free* bugs.

From an out-of-bounds or dangling pointer, it is possible to leak or corrupt information that was not meant to be accessed through this pointer in the first place, opening a path to a variety of attacks. One possible exploit of this is called a *code injection* attack i.e. some arbitrary code is written in the now accessible memory and gets executed [83]. *Return-into-libc* attacks [76] can use buffer overflows of unprotected local stack variables to overwrite the return address of the current stack frame with the address of an arbitrary function in libc (or any other piece of code present in the process's memory at this point), hijacking the control-flow of the program. *ROP* (return-oriented programming) attacks [16] overwrite return addresses with the location of an arbitrary point in the existing code where there is a useful gadget, i.e. a small piece of code performing a specific action and ending with a return instruction. Chaining gadgets can allow for arbitrary computation. *JOP* (jump-oriented programming) attacks [10] work in a similar way to ROP attack but generalising the requirement that gadgets end in return instructions to jump instructions.

On top of attacks aimed at the control-flow of the system, some attacks can directly target data in order to alter program logic and gain privileges or leak information [19].

2.1.2 Existing mitigation techniques

Some mitigation techniques exist for the various presented classes of attacks. For example, code injection attacks are made impractical by the W$\oplus$X policy enforced by modern hardware (AMD's NX no execute bit [36] and ARM's XNX execute never bit [7] for example). This forbids a memory page from being both writeable and executable (paging is discussed in more details in **Section 2.2**), effectively preventing code to be written on a page and also executed. Attacks relying on overwriting addresses on the stack can be somewhat mitigated with the help of stack protectors or stack canaries [27, 38] which either detect or prevent the change of a return address on an active stack frame. DynIMA [32] tries to detect ROP attacks by observing at runtime whether suspicious sequences of potential gadgets (instructions sequences terminated with a return instruction) ever execute. Similarly, DROP [17] can detect ROP attacks by identifying continuous returns to the same specific presumed gadget addresses.

Operating system randomization [20], and particularly ASLR (Address Space Layout Randomization) is an approach that aims at making the crafting of an attack over an existing exploit, if not impractical, at least much harder. The general idea is to make the information required by the attack to work (system call number, library entry point addresses, and other addresses in general) not statically allocated, but rather randomly determined. This forces extra steps to be taken in the attacks to first discover this information, which may not be a trivial task.

Some more general security policies mitigating larger classes of attacks also exist, with the ideal being to enforce complete *memory safety* i.e. the absence of spatial and temporal memory violations.

Knowledge of pointer bounds is necessary to enforce memory safety. Techniques that try to capture intended pointer behaviours such as Cyclone [51] that annotate sources or CCured [65] that perform some static analysis can replace conventional pointers with *fat-pointers* that keep track of bounds informations and use it to perform various checks to enforce spatial and temporal memory safety.

Some partial memory safety can be provided by identifying *unsafe* pointers that have runtime computed values (i.e. may go out of bounds) and specifying the set of locations that are legitimately reachable through these pointers using static analysis, and then instrumenting the program to tag legitimate locations in a shadow space and add checks on writes through unsafe pointers such that only legitimate locations can be written. This is the technique presented by Yong et al. [102]. WIT [4] refines this approach by marking memory locations not simply as legitimate or not, but with an ID referring to a specific unsafe object, which further restricts the set of locations to which writes are possible. Similar ideas were already used by Abadi et al.'s CFI [1] (control-flow integrity) which statically determines the valid execution flow of a program and checks this information on jumps and returns. The information is stored inlined with the code rather than in a shadow space, the argument being that it can be protected by the read-only property of code memory pages (i.e. "code integrity" policy).

Other compile time transformation tools such as SoftBound [64] try to provide spatial memory safety by keeping track of pointer metadata in a shadow memory space, keeping compatibility with standard binaries. This is a *pointer based* approach, suffering from the caveat that the information attached to a pointer is only updated by instrumented program modules. *Object based* approaches also exist as an alternative, and focus on pointer manipulation checks rather than dereference, such as for example the approach presented by Richard W M Jones and Paul H J Kelly [54], or the CRED approach [71] that adds the ability to support out-of-bounds pointer manipulations. On top of these techniques enforcing spatial memory safety, there exist approaches that try to address temporal memory safety issues. Valgrind Memcheck [67] is a binary rewritting approach that keeps track of the *definedness* in a shadow space in order to detect accesses to recently deallocated memory, but does not cope with reallocated memory. CETS [63] is a compiler enforced technique that gives memory allocations a unique identifier and associates it with the pointers, performs runtime checks when dereferencing the pointer, and enables full memory safety if spatial memory safety is guaranteed.

Szekeres et al. [85, 86] consider the performance overheads of a mitigation technique, its compatibility with other existing features, its robustness (i.e. whether the offered protection is complete), and whether it depends on changes in compiler toolchain or source-code. They observe that none of the proposed techniques that enforce spatial or temporal memory safety (from which all other classes of attacks emerge) are deployed in practice.

Very few of the mitigation techniques providing partial memory safety are in fact being used at all. Only 4 techniques are actually reported as regularly deployed: code integrity and non-executable data (both hardware enforced), and ASLR and the stack guards/cookies (software techniques which offer very low performance overheads). SoftBound, CETS, WIT, CFI and a the rest of the techniques listed in Szekeres et al. [86]'s summary table are not deployed. The success of hardware enforced techniques motivates the search for more hardware approaches to security.

More recent techniques also exist. For example, the pointer authentication instructions introduced in Armv8.3-A [72] enable an application to tag pointers with a Pointer Authentication Code (PAC) computed as a cryptographic hash based on a set of input values including secret keys inaccessible to user code. This PAC can later be verified by the application before dereferencing the pointer. Another technique is the Intel Memory Protection Extensions (MPX) [48] which provides a way to check pointer references. It store the bounds of a pointer (either in memory next to the pointer, or in a dedicated shadow space), and accesses those bounds that are placed in special registers and checked with dedicated instructions. Note that these mechanisms rely on explicit use of the extension features in the code. If untrusted code is ran in a sandbox with access to a limited region of memory, neither ARM's pointer authentication technique nor Intel's memory protection extensions would prevent that code from crafting a pointer to outside the sandbox and accessing it. The problem here comes from the fact that both mechanisms can easily be bypassed, which enables untrusted code to forge memory references[1].

2.2 The current hardware approach to security: conflation of virtualisation and security

The current hardware approach to security opportunistically exploits the memory translation mechanisms already in place to enable memory virtualisation [34]. Here, "virtualisation" is the concept of creating an abstract logical representation of a resource, abstracting the access to the resource from the resource itself. Virtual memory as proposed on the Atlas computer [39] provides a level of abstraction for memory accesses giving the illusion of a linear virtual address space to applications while mapping pages (e.g. 4KiB blocks) of virtual memory to physical memory or backing store (swap). It gives to the virtual memory programmer a vision where the logical (virtual) memory has a logical size and can be accessed with logical addresses that can differ from the actual (physical) size and actual addresses in the implemented memory. A simpler virtual interface to the actual memory is then possible, where the programmer manipulates "virtual addresses" from a "virtual address space" without knowledge of the "physical addresses" from the "physical address space" that these map to. This allows for the hiding away of implementation details, i.e. in the case of Atlas, the fact that physical memory is actually backed by a "core store" and "drum" combination [56]. This creates the need for a layer of virtual to physical address translation in the system. The following paragraphs explain the basics of memory virtualisation in order to understand how security features can be added to those mechanisms, and the shortcomings of this approach.

2.2.1 Paged virtual memory

Denning [33] explains that in a paged virtual memory system, the main physical memory is conceptually organized in fixed sized blocks called "physical pages" or "page frame". These physical pages can store the content of "virtual pages" which are blocks of memory manipulated by the programmer in the virtual address space, and of size matching the block size of physical pages. A physical page number is used to identify a specific physical page, and a virtual page number identifies a specific virtual page. A virtual address is constituted of a virtual page number together with

[1]We later introduce capabilities (see **Section 2.3**) which are unforgeable tokens of authority, and extensively discuss the CHERI architecture (see **Chapter 3**), which addresses this problem.

a word index or offset within the virtual page. For power-of-two block sizes s, the virtual page number and the offset are simply the upper $n - log2(s)$ bits and lower $log2(s)$ bits of the n-bit virtual address respectively.

To get a physical address from a virtual address, virtual pages are mapped to physical pages, and the mapping information is kept in a page table. The virtual page number of the virtual address is used to index the page table. A page table entry contains a physical page number which can be substituted to the virtual page number in the virtual address to produce a physical address, preserving the offset within the same-sized block of memory, but making the position of the block of memory easily changeable.

Note that modern page tables are not simple linear tables, but hierarchical tables, and often allow for a predefined set of page sizes which can coexist at the same time. Maintaining active mappings is usually performed by the operating system, but the translation from virtual address to physical address itself is implemented in hardware. The page table is stored in main memory, but page table entries can be cached once looked up to avoid frequent memory access because of virtualisation (which would double memory traffic). The caching structures used to store page table entries are usually associative memories called table lookaside buffers (TLBs). A TLB together with the circuitry performing the address translations is often referred to as a memory management unit (MMU).

2.2.2 Hardware security in paged virtual memory systems

From a virtualisation of memory perspective, the only information required in a page table entry is the physical page number that should be used to store the associated virtual page. But it is possible to store some additional metadata to describe properties of a page. For example, it is possible to have some attributes telling whether memory accesses to this page should be cached or not, or a dirty bit telling whether some data has been written in the page. Similarly, some bits can be dedicated to making the page readable, writeable, executable, or some combination of these. Modern x86 [47] and x86_64 [36], ARM [7], MIPS [46], PowerPC [45] computers all implement page "protection" bits that aim at controlling the kind of memory accesses allowed for that page. With some small variants, those bits essentially control the rights to read data, write data, or read (execute) instructions. The W$\oplus$X policy previously mentioned rely on these bits to be implemented. It effectively ensures that no page that could have been written by an attacker can simultaneously be executed, preventing code injections.

Bundling these bits in a page table entry allows easy enforcement of various policies at the time of address translation which always has to be done on the path to memory. However, this approach also ties the protection bit to the same granularity as that of the pages. This means that in order to have a software object only readable among other writeable objects, it is necessary to map the read only object to its own page. If two writeable objects coexist in the same writeable page, it is not possible to detect that a write access meant for one object reaches memory locations of the other. This coarse granularity can enable such things as buffer overflows.

A page table effectively defines a process's vision of the memory. Any piece of code executing with a given page table, as part of a given process, can access any part of the memory that is mapped in this page table. This means that an application with compartments that do not wish to allow other compartments to access their data cannot be implemented within a process with a single page table. The manipulation of page table entries is generally performed through privileged instructions. Setting up a page table entry is therefore time consuming, as it necessarily involves context switching into the kernel. This means that an application with mutually distrusting compartments is constrained to trade performance for security features, spawning separate processes and page tables for separate security domains.

2.2.3 Segmented memory

Segmented memory conceptually addresses a memory location with a pair {segment number, offset within the segment}. The segment number is used as an index into a segment descriptor table to obtain a base address. The offset is added to the base to form the effective address. This allows for easy relocation of blocks of memory, as addressing done within a segment is done relatively to its base.

Additionally, a segment descriptor contains information on the segment's size, which is a property of each individual segment, as opposed to pages that all have the same fixed size. Knowing the segment's bounds through its base and size fields enable bounds checking of a memory access. Segment descriptors can also embed other protection bits to further filter memory accesses.

The segment descriptor table is typically managed by the OS, so subsetting a segment is expensive. For example, it might be desirable to use a segment to describe a stack frame, thereby limiting memory accesses to the current stack frame through bounds checking. This would however be far too expensive since stack frames change frequently and each change would require an expensive OS call.

As opposed to the paged approach, protection over segmented memory can be performed on a finer granularity, though as pointed out by Denning in [33]: "Loading a segment requires finding an unallocated region large enough to contain the new segment, whereas loading a page requires finding an unallocated page frame. The latter problem is much less difficult than the former: whereas every unallocated page frame is exactly the right size, not every unallocated region may be large enough, even though the sum of several such regions may well be enough".

Denning explained how segmented memory was a means of implementing virtual memory in 1970, but the fragmentation of memory from this approach has effectively made paged virtual memory the method used to provide virtual memory in today's computer systems. Segmented memory is sometimes still used on top of the paging virtual memory system.

Note that the conflation of memory protection with memory translation as occurs with paged virtual memory leads to poor security properties. The CHERI model (see **Section 3.1**) distinguishes memory translation and memory protection. It can work with a conventional virtual memory system (using paging). It uses objects similar to segments, "capabilities", not to translate addresses, but to perform memory protection checks. Capabilities will now be explained in more detail.

2.2.4 Mondrian Memory protection: an attempt at separating security and virtualisation

The "MMP" Mondrian Memory Protection technique [97] from 2002 tries to separate memory protection mechanisms from memory virtualisation mechanisms. It uses a table of permissions for each word of the memory. This table is similar to a page table but with much finer granularity, and is in practice implemented as a sparse multi-level table, with its entries cached in a "PLB" (Protection Lookaside Buffer).

The MMP protection mechanism cannot be bypassed (every memory access is checked against the permission table) and the protection table cannot be manipulated by user code. MMP allows several protection domains in a single virtual address space. However, providing protection via a table keyed by word addresses, it is not possible to have several references to a same region with different rights associated with them. It is also not possible to distinguish between different contiguous regions with the same permissions[2]. MMP's protection table maintenance requires privileged manipulations. The multi-level implementation of the protection table also implies the need for complex software or hardware table walking with extra memory accesses.

2.3 Capabilities

Capabilities are "unforgeable tokens of authority" that can be delegated. They are abstract handles on specific resources granting a set of rights and abilities to perform some operations on those resources. They are not forgeable, meaning that no fraudulent copy or imitation of a capability can be crafted. Using a capability to represent the authority held by a "computation" or software component is a notion first introduced by Jack B. Dennis and Earl C. Van Horn [35].

In the context of computer systems, capabilities can represent the authority of a component of the system to complete specific tasks using other components of the system, as explained in the first chapter of "Capability-Based Computer Systems" by Henry M. Levy [58]. Software applications can be viewed as a set of components, each requiring a set of resources and abilities to fulfil their assigned task. Objects that will be manipulated and the actions that can be performed on each of these objects can be referenced using a set of capabilities. Different parts of an application will be granted a different set of capabilities as each is assigned a specific task, necessitating a different subset of objects and rights on these objects. The individual capabilities represent unforgeable references to these objects, carrying both information on how to get to them and what set of operations can be performed with them when used through those reference.

In this thesis, capabilities will describe a set of memory locations accessible to a piece of software and the access rights that this piece of software can have on those memory locations (read, write, execute etc.). A software component being granted a

[2]Accesses to different contiguous regions could be detected by inserting "guard regions" (similar to guard pages for paged memory), however this would not cope with accesses offset beyond the guard region.

specific capability to some object can also delegate this capability to another component of the system, effectively granting that other component access to a very specific subset of the memory and rights that it has access to. The receiving component still does not have access to any of the memory that has not been explicitly delegated to it.

A "C-list" or list of capabilities effectively defines a set of privileges over some resources. This defines a protection domain. A component of a system will only be able to interact with the subset of the whole system within its protection domain (described by the C-list it is granted), and only in ways that are explicitly specified in the capabilities of its C-list. Providing a component of the system only with the resources necessary for its legitimate purpose is known as the principle of least privilege [75], that is "every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job".

This is a desirable feature of a computer system where multiple tasks can be executing simultaneously, as it provides a means to separate the resources of the system that are necessary to each individual task, only granting access to the required resources, avoiding undesired interactions with other tasks. In practice, in computer systems, this is done by separating the subsets of the memory each piece of software will be able to access.

2.4 Software implementations of capability systems

Several tasks executing concurrently and on the same memory system can potentially read or write locations accessed by one another. On one hand, this can be used to implement communication between tasks (shared memory regions). On the other hand, this raises the question of task separation. It might be that some data manipulated by a specific program should not be seen by others, or simply that a task expects its data not to be overwritten by a third party before its next read. The isolation of an application's memory accesses can prevent undesired accesses from taking place.

To filter memory accesses, several techniques can be used. Paging (see **Section 2.2**) conflates the notions of protection and virtualisation by associating with each virtual addresses of a program a physical address within a page that has been tagged with specific access rights. This enables operating systems to allow applications to use virtual addresses that could conflict, and at the same time be sure that they will not interact with each other by mapping those virtual addresses from different applications to different physical addresses.

The paging mechanism ensures that an application running within a given virtual address space can only access physical pages allocated by the OS. Additionally, the per page permissions or rights can, to some extent, express the kind of operations that are allowed on the data stored in the page (a program can for example be able to read but not modify a specific page). This can notionally be regarded as an ad hoc way of implementing a capability in the sense that mapping a set of pages is effectively equivalent to granting specific accesses to a memory region. The memory cannot be accessed unless a mapping exists, only authorised accesses can be performed, and a user application cannot forge it as only the operating system is allowed to register a mapping. Pages are usually on the order of several KiB or more, which means that the granularity at which undesired accesses can be captured is relatively coarse.

An application that wants to use several references to the same data, each allowing for different operations on the data e.g. a producer only writing and a consumer only reading, cannot be trivially implemented with this approach. Using two virtual pages in the same address space pointing at the same physical page to craft several references to the shared data with different rights is possible. However, both virtual pages are accessible from within the process which is a fundamental flaw. This approach still captures some of the programmers' intent and makes attacks harder to implement. The most widely used approach consists of using two different address spaces, one for the producer and one for the consumer, each mapping the shared data physical page with a virtual pages only providing the relevant rights. Overall, this indirect way of crafting customized references to data comes from the fact that paging actually associates access rights with the data's location rather than with the reference.

Memory segmentation mechanisms (see **Section 2.2.3**) are in that sense a much better fit for the concept of a capability. These involve segments with a base and a length describing memory regions of arbitrary size down to a byte granularity, and a set of permission bits associated with the segment. A segment can be used to represent a memory reference. As previously pointed out, segments can in theory be used to implement virtual memory but this approach suffers from fragmentation, making paging mechanisms a much better fit for memory virtualisation.

2.4.1 Software capability systems leveraging the process model abstraction

Computer systems that actually implement capability mechanisms usually allow controlling them through privileged instructions only, which gives a way to implement the unforgeability of capabilities by only allowing trusted code to create and manipulate these references.

Several attempts at implementing capability-like protection systems in software have been done, mostly through exploiting the hardware primitives previously described. For example, the Extremely Reliable Operating System EROS [77] used several virtual address spaces to separate different protection domains. The API provided by EROS would wrap system calls to actually interact with the hardware paging mechanism. This meant that capabilities had to be implemented at a coarse page granularity. More generally, capability operations requiring interaction with the paging mechanism such as inter protection domain communication would require a system call, leading to context switch costs in and out of the kernel, making the whole mechanism costly.

The Capsicum [92, 96] approach uses the paging mechanism to implement capabilities in a similar way to the EROS microkernel. It is designed as an extension to the UNIX file descriptor API. It can be used for application compartmentalisation and sandboxing, breaking up software in smaller components to ultimately mitigate potential vulnerabilities. This enables it to be deployed alongside conventional applications in an existing UNIX like operating system, which explains its adoption in the FreeBSD kernel.

Google's Native Client [101] is a system implementing sandboxing of browser based applications. It aims to allow untrusted code access to features such as threads, SSE instructions, compiler intrinsics and hand-coded assembler while still preventing potential system-wide corruption. At its core, this system leverages the segmentation mechanism previously introduced, simplifying its internals and limiting its overheads. Few attempts at using segmentation have however been made, and the success of paging has unfortunately displaced segmentation to the point that support for generic segments was dropped in the x86_64 architecture.

2.5 Hardware capability machines

This notion of capability has also existed in various different hardware machines, without ever reaching commercial success. In practice, these machines conceptually implement hardware support for an object model. The most significant reason for such hardware features not to have become more popular is mainly the comparative simplicity of the now pervasive paging mechanisms, and the conflation of coarse grained security features with those mechanisms which was historically considered good enough.

Two approaches exist. Capabilities can be accessed directly in dedicated registers, or indirectly through a table (or set of tables, containing various bits of metadata).

The direct approach provides the benefit that referencing an object can be done without expensive table lookups. On the other hand, the required storage for object references is increased with each reference to an object.

The indirect approach has an advantage when trying to revoke a reference: the reference exists in a single centralised place, the table (or set of tables), and can therefore be located and altered or removed easily. Storing references in arbitrary locations make this process more complicated.

Note that the information qualifying the reference is stored only once in the indirect approach and only a table index needs to be preserved. In the direct approach, the information in the reference is duplicated with each copy of the reference. It follows that the memory footprint in the direct approach grows faster with the number of references than in the indirect approach. This cost is no longer a concern with modern computer system's memory sizes. Larger object references can however remain a problem for memory bandwidth requirements as we will see in **Chapter 5**.

2.5.1 Chicago Magic Number Machine

The Chicago Magic Number Machine [58] from 1967 had 16 general-purpose registers and 6 capability registers with among other fields a base, a length and some permissions. These capability registers were meant to access memory regions that could contain either data or capabilities, and were dereferenced with an offset into the region provided through a general purpose register or an immediate field. The effective C-List of an executing task would be defined by the set of capabilities present in register as well as all capabilities reachable in memory (N.B. that means with an arbitrary number of indirections, e.g. loading a capability to a region of memory holding more capabilities, nesting the process an arbitrary number of times).

2.5.2 MIT PDP-1

The MIT PDP-1 in 1967 was the first system with Dennis and Van Horn's ideas implemented [58]. Capabilities were stored in locations 0 to 77 of the process address space which could only be manipulated by the operating system. Only protected instructions could create capabilities. Capabilities were of a certain type, for example, capability to an I/O device, a file, a queue, etc. Operations were performed by using an "INVOKE" instruction that would take as argument the C-list index of the target capability and the operation to be performed. This computer was used to implement Ackerman and Plummer's multiprocessing computer system [3] which made use of capabilities for modularity and security.

2.5.3 Cambridge CAP computer

The Cambridge CAP computer [66] in 1970 allowed a process to name capabilities contained in up to 16 capability segments as operands to capability instructions via a capability specifier. To be used, these capabilities need to be present in the "capability unit". This capability unit consists of a memory of up to 64 "evaluated capabilities" i.e. entries with the base, length and access permissions available, as opposed to just the capability specifier. These capabilities are evaluated via some microcode that looks up this information into data structures indexed by the capability specifier and caches the result in the capability unit's memory. On ENTER/RETURN operations, which are used to go from one protection domain[3] to another, the capability unit enforces that the capabilities of the calling domain are no longer available to the callee, without necessarily requiring that they get flushed from the unit's memory.

2.5.4 Intel iAPX 432

The Intel iAPX 432 [98] conceived in 1975 tries to implement an object model in hardware to improve programming environment. All information manipulated is encapsulated in objects, with memory reference operations being checked for privileges and bounds. Manipulating pointers to objects themselves can only be performed by trusted hardware and microcode, and not directly by the user. Memory dereferenc-

[3]In CAP, a process's C-List is described by the fundamental segment PRL (Process Resource List)

ing can be performed through an "Access Descriptor" (AD) selector that ultimately refers to one specific object and an offset within this object. An AD is a capability in that it both describes a resource and the set of privileges that are granted on that resource. Performing the dereference is a non trivial operation as the AD selector will be used to get an actual AD that consists of a directory and a segment field each used as indices in two separate object tables (the reasons for having two separate tables mostly being the sizes of these tables [24]). Ultimately, the pointer to an object is reconstituted from a central object table that can describe up to 2^{24} objects. The full addressing path of the 432 involves yet more indirections, which effectively makes it a rather complex machine. Instructions use up to 3 operands. Only memory operands are supported, as opposed to register operands. Instructions require up to 3 times 93 bits to fully describe their operands, and the processor relies on a cache to actually speed up the operand lookup process.

The iAPX 432 was a commercial failure due to its inefficient implementation, despite the attractive programming model. Its complex design, use of two separate integrated circuits, slow instruction throughput along with the lack of well optimised compilers for the architecture prevented it from delivering good performance when compared to other contemporary processors [24, 44].

2.5.5 M-Machine

The 1994 M-Machine [14] implements guarded pointers, encoding a segment descriptor in the upper bits of every 64-bit pointer. Only privileged manipulation of these upper bits is permitted, preventing user code from forging pointers, but making it mandatory to switch to a kernel mode to effectively craft new ones. The M-Machine use a "compressed" pointer representation that is explained in more details in **Section 4.1**.

Even though these early attempts at capability hardware did not lead to actual adoption, the ideas that were presented are still sound when it comes to addressing modern security concerns.

2.6 Summary

A number of attacks rely on exploiting memory safety weaknesses. Current approaches to enforcing memory safety use hardware mechanisms with coarse granularity, and a few efficient software techniques offering some attack mitigations. Expensive software approaches providing better memory safety also exist but are rarely deployed in practice due to their cost.

Capability machines can provide memory safety to a computer system[4]. Software implementations exist, but come with performance costs. Older computer systems have attempted hardware capability machines implementations, but without commercial success at the time, mostly due to the success of paged virtual memory machines and the performance that comes with them.

The next chapter presents CHERI, a recent attempt at a capability machine in a RISC ISA, that builds on the original capability machines ideas, and that can be composed with a paged virtual memory system.

[4]Note that there exist orthogonal approaches to the *sandbox* model provided by capabilities. In the sandbox model, domains are protected from each other by permitting access only to resources that have explicitly been granted, following the principle of least privilege. The *enclave* model, provided to some extent by ARM Trustzone [6] (only a single enclave), or Intel SGX [60], prevents access to the enclave from any other domain, including operating systems or hypervisors. CHERI capabilities (see **Chapter 3**) naturally facilitate the resource delegation of the sandbox model, but would require further architectural guarantees to also enable an enclave model.

Chapter 3

The CHERI project

This chapter introduces the CHERI capability model, as well as the general infrastructure and the tools used in the CHERI project, in order to give some context to the work achieved in the rest of the document. The current state of CHERI's implementations will also be detailed as it will serve as the baseline for the optimisations presented in the following chapters.

3.1 The CHERI model

CHERI (Capability Hardware Enhanced RISC Implementation) [95, 100] is a capability machine, leveraging ideas from the computer systems presented in **Section 2.5**, implemented in a modern RISC ISA. CHERI aims to provide a capability mechanism that can give the same level of isolation as user/kernel separation, and that can scale down to individual pointer level to provide memory safety for fast and fine-grain use cases. More generally, CHERI memory protection allows the principle of least privilege [75] (see **Section 2.3**) to be applied to software.

3.1.1 Capabilities as a hardware primitive

The following paragraphs present the CHERI capability primitive type in more detail. This new type (the capability) is a bound checked pointer. Capabilities can be used as code and data pointers. A (code capability, data capability) pair can be used to form a compartment (see details later in this **Section 3.1.1**).

The CHERI capability primitive type

CHERI capabilities are composed of the following architecturally visible fields, as described in the CHERI Architecture Document [95]:

base
This 64-bit field is the **base** virtual address of the memory region described by a capability.

length
This 64-bit field is the **length** of the memory region described by a capability.

offset
This 64-bit field holds a signed **offset** between the **base** of the capability and a byte of memory that is currently of interest to the program that created the capability. It can be used as an index into an array stored in the capability to the array itself for example. No bounds checks are performed on **offset** when its value is set by the `CSetOffset` instruction: programs are free to set its value beyond the bounds of the capability as defined by the **base** and **length** fields. Bounds checks are performed when a program attempts to dereference a capability, to access memory at the address given by **base** + **offset** + a potential general purpose register offset.

s The **s** flag indicates whether a capability is usable for general-purpose capability operations. If this flag is set, the capability is sealed and it may be used only by certain of the CHERI instructions (specifically `CCall` or `CUnseal`, documented in the CHERI Architecture Document [95]). The sealing mechanism can effectively provide immutable software-defined capabilities along with the **otype** field.

otype
This 24-bit field holds the “type” of a sealed capability. Together with the **s** flag, this field restrict the use of the capability. By definition, it is only possible

to unseal a capability if you hold another capability granting the permission to `CUnseal`, and granting that permission to a range of types that include the type of the capability that you try to unseal. It is also possible to perform a `CCall` with a pair of code and data capabilities sealed with the same type, and land in a protection domain defined by these two capabilities now unsealed (other capability registers need to be explicitly cleared in order not to be leaked from one protection domain to another). The type field is intended to be software managed.

perms

The **perms** bit vector governs the permissions of the capability including read, write, execute and sealing/unsealing permissions among others described in **Table 3.1**.

uperms

The **uperms** bit vector may be used by the kernel or application programs for user-defined permissions. They can be masked and retrieved or checked using the same instructions that operate on hardware-defined permissions. User-defined permission bits can be used in combination with existing hardware-defined permissions (e.g. to annotate code or data capabilities with further software-defined rights), or in isolation of them (with all hardware-defined permissions cleared, giving the capability only software-defined functionality). For example, user-defined permissions on code capabilities could be employed by a user-space run-time to allow the kernel to determine whether a particular piece of user code is authorized to perform system calls. Similarly, user permissions on sealed data capabilities might authorize use of specific methods on object capabilities, allowing different references to objects to authorize different software-defined behaviours. By clearing all hardware-defined permissions, software-defined capabilities might be used as unforgeable tokens authorizing use of in-application or kernel services.

In addition to these fields, every 256-bit capability is accompanied by a single validity tag bit indicating whether the 256 bits are to be considered as a capability type or a raw data.

CHERI capability operations

A number of CHERI instructions exist to manipulate the capability primitive type. These instructions are detailed in the CHERI Architecture Document [95]. A non-exhaustive list of CHERI instructions grouped in several classes is presented as an overview of the operations available on the new capability primitive type:

Capability inspection

These instructions (`CGetBase`, `CGetLen`, `CGetOffset`, `CGetSealed`, `CGetType`, `CGetPerm`, `CGetTag`, `CSub`, `CtoPtr`, `CPtrCmp`) generally take a capability register argument and return a queried capability field or property in a general purpose register.

Index	Name	Description
0	Global	Allow this capability to be stored via capabilities that do not themselves have *Permit_Store_Local_Capability* set.
1	Permit_Execute	Allow this capability to be used in the PCC register as a capability for the program counter.
2	Permit_Load	Allow this capability to be used as a pointer for loading data into general-purpose registers.
3	Permit_Store	Allow this capability to be used as a pointer for storing data from general-purpose registers.
4	Permit_Load_Capability	Allow this capability to be used as a pointer for loading other capabilities.
5	Permit_Store_Capability	Allow this capability to be used as a pointer for storing other capabilities.
6	Permit_Store_Local_Capability	Allow this capability to be used as a pointer for storing local capabilities.
7	Permit_Seal	Allow this capability to be used to seal or unseal capabilities that have the same **otype**.
8	*reserved*	
9	*reserved*	
10	Access_System_Registers	Allow access to EPCC, KDC, KCC when this capability is in PCC.

Table 3.1: *Capability permission bits for the* **perms** *capability field*

Capability manipulation

These instructions (`CSetOffset`, `CIncOffset`, `CSeal`, `CUnseal`, `CAndPerm`, `CClearTag`, `CClearRegs`) alter a capability provided an authorising capability that holds at least as many rights as the ones expected to be held by the new altered capability. This is the monotonicity property of the CHERI instruction set architecture that guaranties that no escalation of rights is possible.

Control-flow

These instructions (`CBTS`, `CBTU`, `CJALR`, `CJR`, `CCall`, `CRetrun`, `CCheckPerm`, `CCheckType`) enable a program to control its execution flow based on capability register values, either through conventional jumps or through exceptions.

Memory

These instructions (`CL[BHWD][U]`, `CLC`, `CLL[BHWD][U]`, `CLLC`, `CS[BHWD]`, `CSC`, `CSC[BHWD] CSCC`) are used to access memory by explicitly using a capability. `CLC`, `CSC`, `CLLC` and `CSCC` specifically are used to load and store capabilities from the memory to capability registers and back. These must be used to interact with the memory and preserve the validity tag bit of a capability.

Capability creation

These instructions (`CFromPtr`, `CSetBounds`, `CSetBoundsExact`, and historically `CIncBase` and `CSetLen`) create a capability given an existing capability that holds at least as many rights as the ones expected to be held by the newly created capability. Again, this class of instructions contributes to the monotonicity property previously described. This class of instructions has been heavily influenced by the work presented in **Chapter 4**. Specifically, `CIncBase` and `CSetLen` have been removed from the CHERI ISA in favour of `CSetBounds` (and `CSetBoundsExact`) which atomically derives a new capability.

3.1.2 The CHERI MIPS architectural extensions

The CHERI model is implemented within the MIPS RISC ISA, as an ISA extension, but is conceptually portable to other ISAs. MIPS is a conventional load/store RISC architecture with 32 general purpose registers and a program counter register (PC). Instructions are fetched from the addresses stored in PC and data can be loaded and stored using addresses from general purpose registers. These addresses are virtual addresses visible to the program that is currently running, but get translated to physical addresses on their path to memory (MIPS has an MMU with a paging mechanism).

CHERI enforces that every memory access is checked against a valid capability. For that, it extends the MIPS ISA with 32 capability registers[1] that can be manipulated through dedicated capability instructions (see §"CHERI capability operations – Memory" in **Section 3.1.1**), a PCC (PC Capability) register, as well as new capability instructions using the coprocessor 2 opcode space, and a tagged memory. The capability installed in the PCC register should conceptually describe a region of memory containing the code of the application being run and have the executable permission set, as PCC is the capability used to check memory accesses due to instruction fetches. Similarly, data memory accesses are performed with respect to a capability in one of the capability registers. Legacy loads and stores that don't explicitly name a capability to use for the memory access implicitly use the default data capability (DDC) (which corresponds to capability register 0). Some of the capability registers are reserved for kernel use only, and cannot be manipulated by user code. Specifically, there exist a kernel code capability (KCC), a kernel data capability (KDC) and an exception PCC (EPCC) to store the PCC value at the time of an exception.

CHERI capabilities can only be derived from already existing capabilities. Arbitrary bit manipulation of an existing valid capability in memory will cause its associated tag bit to be atomically cleared, preventing maliciously forged capabilities from appearing. On boot, the system is given capabilities to the whole address space and granting all privileges to the code first executing. Capability manipulation operations are then used to subset those capabilities and craft the domains that will be required by later tasks. Capability manipulation instructions will only ever alter operand capabilities in such a way that the set of privileges they grant will be mono-

[1]The 32 capability registers can conceptually be implemented as extensions to the 32 general purpose registers of MIPS (in the fashion of the 64-bit registers of x86_64 being an extension of the 32-bit x86 registers) or as an extra register file alltogether. The CHERI project uses the later.

tonically reduced, i.e. have as many or fewer permissions set, see the bottom edge of the described memory region increased, see the top edge of the described memory region decreased, etc. It is never possible to craft a capability with a larger set of rights than those granted by the capability it is derived from (using formal modeling techniques as the CHERI L3 model introduced in **Section 3.2.2**, such properties of CHERI can be proved).

3.1.3 Discussion on the CHERI model

Jonathan Woodruff [99] explains that in order to successfully provide a good capability mechanism, the following principles are to be observed:

- All memory accesses need to be performed with a capability. By construction, this makes it impossible for memory accesses that take place not to have passed the checks that assert it is an authorized access.
- The manipulation of capabilities should be protected and not depend on kernel protection. Not relying on privilege mode enables inexpensive manipulations and promotes widespread use of the mechanism. Protected manipulations will enforce monotonicity, preventing any rights escalation.
- A program must be able to pass control between domains possessing different capabilities without granting additional capabilities to either domain. This is necessary to implement efficient compartmentalization and no longer rely on heavyweight process model for task separation.
- Capability-relative memory accesses should be as fast as conventional memory accesses. This is necessary to replace all memory accesses by capability-relative memory accesses.
- Capability related instructions should not require more than one cycle in a classical RISC pipeline. This ensures that capability code does not run slower than conventional code, and can get adopted without performance barriers.
- Capability adoption should be optional and incremental. This allows to run legacy software on the same system as capability code, enabling deployment of capability processors without the requirement to replace the existing codebase.

In the CHERI model, capabilities can be manipulated directly through processor instructions accessible from user-space, removing some of the costs of early capability machines related to switching to a supervisor context for capability manipulations. CHERI capabilities coexist with data in the main memory, and can be loaded in capability registers. All the information required for later dereferencing is present in the capability itself, removing the need for complex table structures with many indirections on the critical path of the machine, like it was the case in the Intel iAPX432 for example. The CHERI model uses a tagged memory (see implementation details in **Chapter 5**) with a single bit of tag for each capability-wide word of memory, to differentiate standard data from capabilities. This also allows the number of capabilities supported by the model to simply scale with the size of the memory as they don't require special hardware to be backed up other than this single tag bit space. Only tagged capability-wide memory words can be used as capabilities, and capabilities cannot be arbitrarily written to without their tag bit being atomically cleared, enforcing their unforgeability.

The CHERI model is also concerned with providing an easy adoption path by being compatible with existing programming models: unmodified binaries will run on a CHERI processor without using CHERI instructions. Unmodified sources can be recompiled to leverage new CHERI features, and overall, it is possible to consider replacing each individual pointer in a program with a hardware capability. New code can be written to explicitly compartmentalize an application using CHERI.

The ideas used by the CHERI model aim at making the capability approach to enforcing memory safety more practical. On a CHERI processor, the relevant checks on the reference are hardware-enforced, and performed on every memory access. This means that bounds checking measures (or lack thereof) to implement fine grained protection mechanisms are no longer a software overhead, but are instead efficiently implemented in hardware, similarly to what was intended by early capability machines. The overheads involved to perform the dereference are very small as all the information required for the tests is readily available in capability registers, and do not need expensive lookups to be fetched as was the case in most early capability machines[2].

Note that CHERI aware code has some overhead with respect to non-secure code that would simply use an integer as a pointer and ignore bounds checking: capability pointers need to be initialised with extra information when first derived: length, access rights and other permissions. Setting up this information can lead to a few extra instructions on pointer creation.

The model associates the protection properties with the references to the data rather than with the data location which is the case when memory virtualisation and memory protection is conflated. CHERI capabilities exist within a paged virtual address space, as the problem of memory virtualisation is best solved by a paging mechanism. In that sense, the CHERI model composes with current protection models rather than replace them. This is a key point playing in favour of the easy adoption of the CHERI model and its ability to run legacy code.

The existence of in address space capabilities enables in address space compartmentalization of applications, which means in address space mutually untrusted components as opposed to expensive process-based isolation. This makes things like sandboxed instances of libraries practical. Untrusted data with potentially malicious payloads being processed in such sandboxes will remain constrained to the sanboxes themselves, considerably limiting the impact of existing exploits in the sandboxed library's source code, preventing the rest of the application from being corrupted.

[2]Note that by moving from a "centralised" system approach where capabilities are stored in a table and where indices need to be indirected to access the capability, we also move away from having a straight forward tracking of capabilities which make the revocation of capabilities a less trivial problem. CHERI however provides some facilities to control the propagation of capabilities throughout memory (through the "global/local" and "permit store local" or "permit store global" permission presented in **Table 3.1**), in turn providing an answer to the revocation problem.

3.2 Tools and infrastructure

The CHERI project spans a wide range of fields within computer science, from micro-architecture to compilers, operating systems and applications. The set of tools and languages used is very large, and this section aims at giving a better idea of the framework within which this work takes place, specifically for the processor design aspects of the project.

3.2.1 Bluespec System Verilog and the CHERI FPGA prototype

One instance of CHERI is an FPGA soft core that can boot the FreeBSD operating system and run applications. The prototype is implemented in the Bluespec System Verilog [69, 70] hardware description language (BSV HDL). This is a high level language that benefits from features borrowed from functional programming languages such as Haskell [9] on which it based. It greatly improves the expressiveness over conventional HDLs. As an example, BSV is a strongly typed language, meaning that assignments of multi-bit signals to other multi-bit signals will be checked at compile time for compatibility where other HDLs would typically permit assignments to simply happen. This greatly simplifies the task of debugging. BSV also has features such as support for mapping functions over vectors, and automates a lot of the flow control between modules, simply exposing object-style "methods" through an interface type. Implementing a BSV module consists of implementing a set of methods to provide a desired interface and possibly a set of "rules" defining the internal behaviour of the module. Rules are an embodiment of "guarded atomic actions" which guarantee that all the behaviour implemented inside it fires atomically or does not fire at all [74], which allows the Bluespec compiler to reason in terms of rule scheduling, and lift a lot of parallelism out of the source code for the hardware being described.

BSV sources can be compiled to a cycle accurate C simulator, which allows early debugging and testing on the development machine. The same sources can be compiled to Verilog HDL. Verilog can be simulated through various software HDL simulators. Using the Altera Quartus toolchain [5], it is possible to use the Verilog description generated by the BSV compiler to enable fast and realistic FPGA based prototyping. The Terasic DE4 boards used for the CHERI prototype have a Stratix IV FPGA chip that contains several types of primitive resource blocks used by the Quartus synthesis tool. The synthesis numbers provided by the toolchain for this FPGA are summarised in a few categories:

ALUTs Adaptive Look Up Tables, which are configurable tables performing arbitrary logic functions.

ALMs Adaptive Logic Modules, which are composed of configurable logic, adders and registers, with some multiplexing to allow for various routings.

Logic Registers Registers holding rapidly accessible state.

Block Memory bits Bits of state accessible through a memory block module.

With the caveat that the Quartus toolchain can be somewhat non-deterministic between runs and allocate FPGA resources differently, resource utilisation numbers can be used as a metric to estimate the hardware costs of a design.

In order to make some performance measurements, I augment the CHERI FPGA prototype with a set of counters, keeping track of various events, principally in the memory subsystem. A generic counter gathering module was developed, and made accessible through some of the MIPS `RDHWR` instruction's registers to enable easy counters sampling. BSV high level features were heavily used in designing this counter setup, which makes it easily extensible: adding new counters is a straightforward process, enabling rapid experimental feature evaluation within the new framework.

3.2.2 L3 domain specific modelling language and the CHERI ISA level simulator

To enable ISA level testing against a CHERI "golden model", an instruction level model of the CHERI MIPS extensions is required. As part of this work, I developped one such model as an extension to Anthony Fox's MIPS model written in the L3 domain specific language [40, 41]. L3 source code is close to the pseudo-code that can be found in architecture description documents, and can be compiled to SML, effectively yielding an executable model, as well as a HOL4 (theorem prover) description that can be used for formal analysis.

The L3 language being developed at the University of Cambridge Computer Laboratory by Anthony Fox is designed to facilitate ISA descriptions. It has primitives for bit lists manipulations, user defined record types, algebraic data types, and a few other functional features. It also has some mutable variables that can be used to represent the architectural state of a processor. One of the most useful aspect of this language from a model designer's perspective is its streamlined process for defining instructions. Rather than updating an "instruction" data type as well as a list of semantic functions implementing those instructions and a decoder to actually map these functions with their opcodes, the language uses the "define" built-in keyword to both implement instruction behaviours and infer the instruction types, effectively avoiding the time consuming and error prone manual process of updating the many places in the sources, reducing the amount of code required to describe an ISA.

The original model covers most of the features available in MIPS, with the exception of a few features such as floating point instructions. With the help of Matthew Naylor, this model has been able to boot the FreeBSD operating system (both in a single core and a dual core configuration). It is therefore usable as a framework for further exploration of capability related strategies.

In order to experiment with capabilities, the basic CHERI capability model needs to be implemented. The coding style enabled by the L3 language makes identifying specific sections of code and mapping them back to the feature they implement easier than it would be with other non-domain-specific languages. The existing L3 MIPS code can be modified with stubs in the places where its behaviour would change for the CHERI version, allowing for a common code base.

CHERI extensions to the MIPS ISA not only add new instructions and capability registers, they also affect standard memory accesses since capability validity tags must also be stored. Implementing a tagged memory is made easy by the presence of algebraic types in the L3 language. Each accessible memory location is implemented as an L3 construct (that is L3's sum type or tagged union type) with a `RawData` constructor and a `Capability` constructor. The core of the model can easily pattern match on these, and make the appropriate decisions to trigger an exception, update the capability register file, etc.

The CHERI L3 model was used in this work to enable exploration at a relatively high level of abstraction. A number of configurations have been made available as a product of these explorations, among which capability sizes (see **Chapter 4**) and caches. The CHERI L3 model has reached a point where it is not only able to run bare metal capability code, but also to boot the CheriBSD operating system (see following **Section 3.2.3**) and run user capability code.

Some side benefits of this modeling work include finding bugs and ambiguities in the CHERI specifications. The CHERI L3 model being an executable specification, it meets the requirement of serving as a "golden model" for fuzz testing the BSV prototype. Matthew Naylor contributed a script generating instruction streams that get executed on the L3 model and the BSV implementation side by side, and a comparison of the outputs helps finding bugs in the BSV implementation of CHERI. The HOL4 export of the CHERI L3 model has been used by Brian Campbell and Ian Stark [12] to generate more useful test cases and helped find more bugs in both the specification and the implementation of CHERI. The HOL4 export also enabled formal work to be undertaken by Kyndylan Nienhuis, who recently proved the monotonicity property of the CHERI capability model in an unpublished draft [68]. The CHERI L3 model has become a valuable tool within the CHERI project, and the approach taken here is now believed to be generally useful and applicable.

3.2.3 High level overview of the other aspects of the CHERI project

In the CHERI project, beyond the hardware models already presented exist a rich software infrastructure. Watson et al. [90, 91, 95] created CheriBSD, a port of the FreeBSD operating system to CHERI architecture. CHERI specific features are used in CheriBSD that enable the kernel to make some use of capabilities. In particular, it provides in process sandboxes for fast protection-domain crossing.

Chisnall et al. [21] developed a Clang LLVM compiler that can compile C code with CHERI as a target. Simple MIPS code can run on CHERI, but CHERI-Clang can also target a "pure capability" ABI, making use of capabilities for all pointers (code and data). Thanks to this compiler, it is possible to rapidly port C applications to CHERI and use them as benchmarks, comparing the MIPS baseline to the pure capability mode.

In addition to my work on BSV hardware counters (see **Section 3.2.1**), I developed a CheriBSD library to enable straight-forward integration of the counters with user C code. The library API presents a high level interface allowing for a snapshot of the set of counters to be taken at arbitrary points in the program and stored in a C struct. A function performing a "struct-wise" difference can generate the count of events between two sampling points. This counter gathering process can also be made implicit, without source code modification, by linking the library's whole archive. This will embed a constructor and a destructor function in the produced binary. The functions will be run before and after the program's `main()`, performing the sampling and the difference automatically. The gathered information can then be displayed in a human readable format, or in a csv format, making the library very useful to gather repeated benchmark runs in a single file.

Finally, a very important part of the CHERI project is its use of continuous integration techniques. The Jenkins continuous integration framework [50] is used to automate builds of the various parts of the project as well as the running of regression tests. I made sure to add to the existing Jenkins framework the various tasks I realised as part of this work. In particular, I created a set of Jenkins jobs building the L3 compiler, building the CHERI L3 model simulator with its various configurations, and running the CHERI test suite on the various CHERI L3 model simulators. With the help of Theodore Markettos, the Bluehive machine [62] was repurposed to serve as a jenkins slave. Bluehive has 16 Terasic DE4 FPGA boards, the main development board for the CHERI FPGA prototype. Next to the already existing Jenkins jobs building the various FPGA bitfiles and CheriBSD kernels, I added jobs to build the Clang LLVM SDK, build a set of benchmarks for the various ABIs supported by CHERI, and run these benchmarks on Bluehive (using the Jenkins generated kernels, bitfiles, and benchmarks artefacts to program the FPGA boards available on Bluehive).

3.2.4 Benchmarks

Three benchmark suites have been used to produce the numbers from hardware runs presented throughout this document. In each case, the benchmarks have been built and run on x86 first to extract their pointer usage profile (see **Section 5.4.1** for more details). This will generally be how the benchmarks are sorted (from low pointer usage to high pointer usage). They were then built for MIPS and CHERI and ran on the different flavours of the FPGA prototype of the processor.[3]

Typical C benchmark: Mibench

The Mibench[43] suite includes bitcount, susan, qsort, jpeg, patricia, stringsearch, blowfish, rijndael, sha, crc32, fft and adpcm. It is representative of typical data-centric C code. Apart from Patricia which manipulates a more complex and pointer based data structure, MiBench benchmarks have low pointer density and minimal memory impact for use of capabilities of any size.

Pointer-heavy applications: Olden

The Olden[13, 73] benchmark suite includes treeadd, perimeter, mst and bisort. The Olden benchmarks use dynamically allocated data structures such as lists or

[3]Note that the floating point benchmarks present in the benchmark suites are generally avoided here as CHERI is built without hardware support for floating point.

trees. These structures involve a significant amount of pointers, making these benchmarks a good “worst-case” for our capability mechanism.

Object-like behaviours with JavaScript: Duktape & Octane

The Octane benchmark suite includes earley-boyer and splay. These are written in JavaScript and run in the Duktape [55] JavaScript interpreter ported to CHERI by David Chisnall. These benchmarks are representative of object-oriented programs using pointer-based objects.

Note that initial effort was made to port the SPEC benchmark suite to the CHERI platform by Theo Markettos. Even though some of the benchmarks could be compiled, it required extensive build system modifications to use the CHERI clang compiler. Unfortunately, the CHERI clang compiler cannot compile the C++ benchmarks. Moreover, the MIPS version of the SPEC benchmarks uses gcc (no engineering effort was put into porting the vanilla MIPS build to use clang), which makes it virtually impossible to compare CHERI against a MIPS baseline. Finally, on top of a 100MHz clock frequency that leads to prohibitively long runtimes (this is still very perceivable when running the JavaScript Octane Benchmarks), the CHERI platform only has 1 GiB of DRAM memory that is used as a mem-disk as well as program memory. On top of the space already occupied by the FreeBSD mdroot image (that is embedding all of the filesystem), the SPEC benchmarks along with their large input files would have to be fitted in the remaining space, and there would still be a need for extra freespace for the programs to actually run. In practice, all this does not fit in the 1GiB DRAM and prevents the subset of SPEC that was ported from being executed on CHERI.

3.3 CHERI implementation

3.3.1 256-bit capabilities

A first implementation of this CHERI capability consists of 256 bits of memory and a validity tag bit, with the fields laid out as shown on **Figure 3.1**.

Access to the memory is in practice more exercised than getting the offset of a capability. As opposed to the architecturally visible **offset** field, an **address** field is used by this implementation. It holds the actual address currently pointed at by the architecturally visible **offset** + **base** fields. This micro-architectural choice allows dereference of an address without the extra addition operation at the cost of extra arithmetic to compute the **offset** architectural field.

3.3.2 The FPGA prototype

The FPGA prototype of CHERI (and BERI, a conventional 64-bit MIPS processor, effectivelly CHERI without capability support) uses a single core 6-stage MIPS pipeline running at a target frequency on an Altera Stratix IV of 100MHz. Its cache hierarchy configuration is meant to be similar to a typical Cortex A53 based SoC

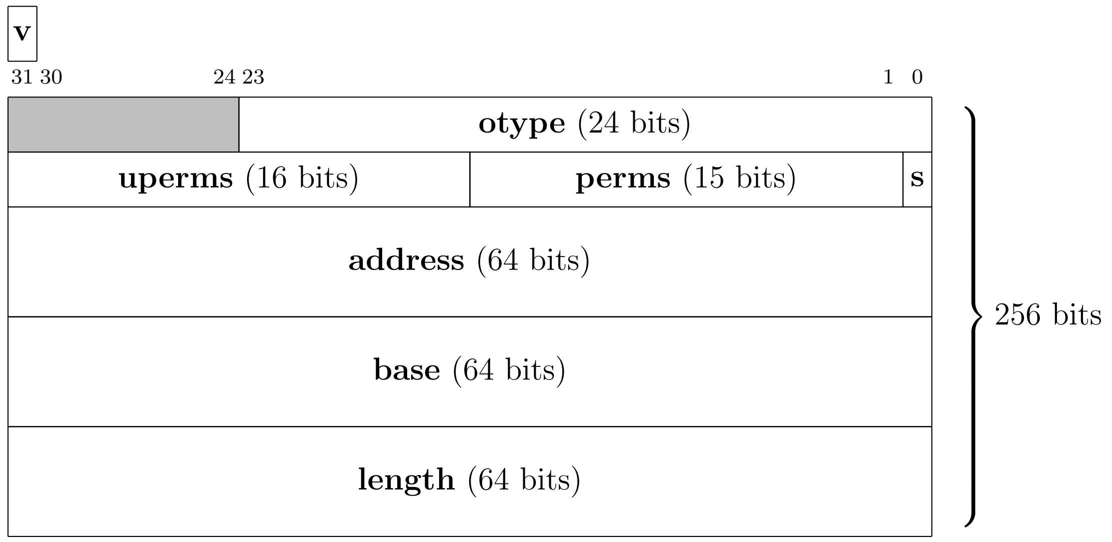


Figure 3.1: *A 256-bit CHERI capability's memory representation showing the fields described in* ***Section 3.1.1***

such as the MT6738 used for entry level smartphones. It has 32KiB instruction and data L1 caches, and a 512KiB L2 cache. CHERI (and BERI) use a 4-way associative write-through L1 data cache, a 2-way associative write-through L1 instruction, cache and a 4-way associative write-back L2 cache. An overview of the cache hierarchy can be seen in **Figure 3.2**.

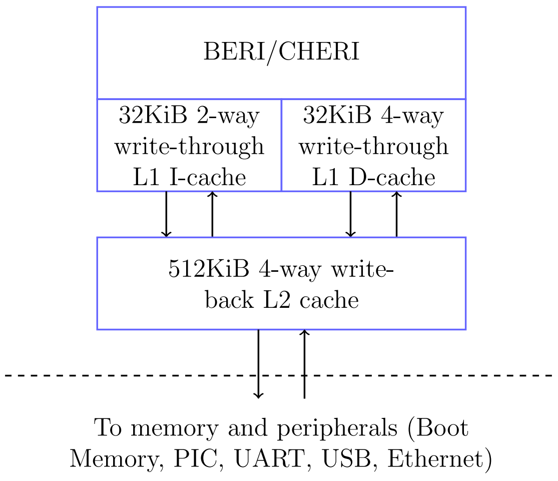
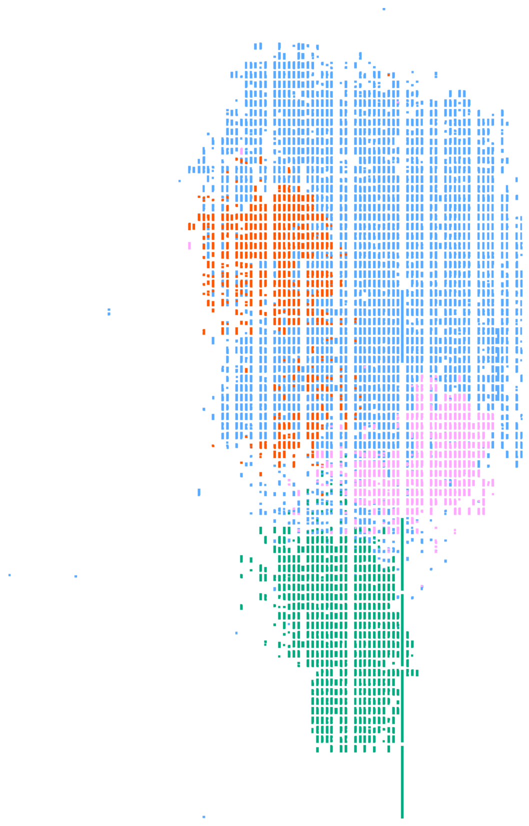
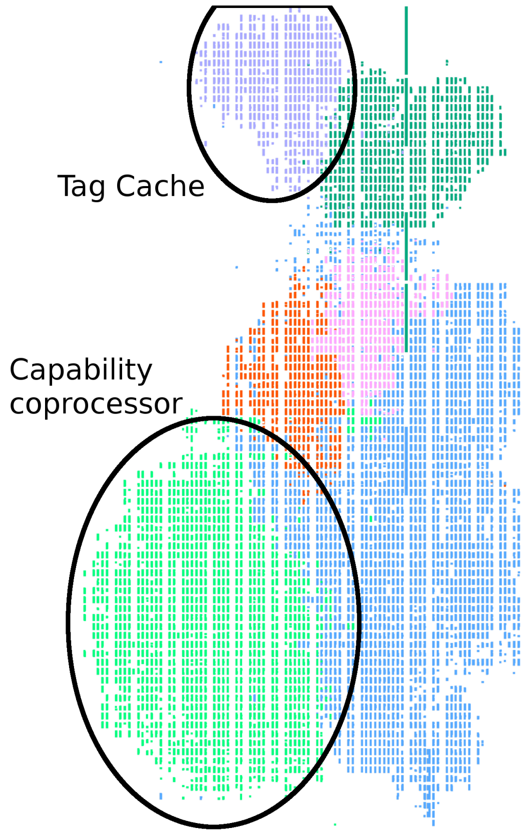

BERI/CHERI
32KiB 2-way write-through L1 I-cache
32KiB 4-way write-through L1 D-cache
512KiB 4-way write-back L2 cache
To memory and peripherals (Boot Memory, PIC, UART, USB, Ethernet)

Figure 3.2: *The configuration of the BERI/CHERI FPGA prototype.*

3.3.3 Extra logic in a RISC pipeline

A CHERI capability system still comes with some costs to be considered. The following section will discuss these costs. A typical 64-bit RISC processor (such as our BERI processor, a 64-bit MIPS pipeline) accesses memory through simple load and store instructions. To implement the various features required for a hardware capability system, the path from the pipeline to the memory followed by these instructions has to be augmented with some circuitry aimed at enforcing memory safety. Memory accesses now need to be performed through capabilities. To guarantee that a memory access is valid, the additional operations to be performed should include checking for the validity of the capability being used. The set of permissions provided by the capability should also allow the current operation to be performed (e.g. the read permission should be set to perform a load, the execute permission should be set to fetch an instruction, etc.). It is also necessary to check that the address of the memory access is within the bounds of the memory object described by the capability. The CHERI MIPS ISA extension reports violation of any of the required conditions (capability's tag, permissions, bounds, or various combinations of these based on the requirements of the specific capability instruction being executed) by raising an exception that can then be handled in kernel mode. The additional logic introduces costs that can be evaluated with our Bluespec implementation, comparing FPGA synthesis results between BERI and CHERI.


(a) *BERI FPGA layout*



(b) *256-bit CHERI FPGA layout*

Figure 3.3: *Comparison of FPGA synthesis results for BERI and 256-bit CHERI. On the 256-bit CHERI FPGA layout, we see the added capability coprocessor and the tag-cache in light green and purple*

Figures 3.3a and **3.3b** were generated using the Altera Quartus version 15.1.0 chip planner tool. They show that CHERI adds a capability coprocessor to the BERI processor, as well as a tag cache[4]. Quartus synthesis results (see **Section 3.2.1** for

[4]CHERI's tagged memory is implemented as a subset of the main memory that can't be directly addressed by software. It is the tag cache that pairs a 256-bit chunk of memory with its appropriate tag. **Chapter 5** further details this mechanism.

	ALUTs	ALMs	Logic Registers	Block Memory bits
BERI TOTAL	36295	32301	26047	3145520
DCACHE	5310	5025	3376	270592
ICACHE	2932	2669	1480	271380
L2CACHE	6225	5825	4352	2162716
MIPSCORE	21828	18782	16839	440832

(a) *BERI resources usage*

	ALUTs	ALMs	Logic Registers	Block Memory bits
CHERI TOTAL	59227	49147	39779	3457400
DCACHE	5347	5045	3444	271616
ICACHE	2771	2551	1484	272408
L2CACHE	5848	5869	4372	2170920
TAGCACHE	9430	7606	5326	282936
CAPCOP	11450	9096	6690	16544
MIPSCORE	24381	18980	18463	442976

(b) *256-bit CHERI resource usage*

	ALUTs	ALMs	Logic Registers	Block Memory bits
TOTAL	63.18 %	52.15 %	52.72 %	9.92 %
DCACHE	0.70 %	0.40 %	2.01 %	0.38 %
ICACHE	-5.49 %	-4.42 %	0.27 %	0.38 %
L2CACHE	-6.06 %	0.76 %	0.46 %	0.38 %
MIPSCORE	11.70 %	1.05 %	9.64 %	0.49 %
CAP overhead	62.34 %	54.37 %	40.33 %	8.29 %

(c) *256-bit CHERI resource usage increase relative to BERI*

Table 3.2: *BERI and 256-bit CHERI raw FPGA resources usage*

details on the different FPGA resources) show the detailed resource usage reported for the two builds in **Tables 3.2a** and **3.2b**. **Table 3.2c** shows a relative comparison between the two builds.

From these results, we gather that the implementation costs of a capability system are mostly noticeable in terms of logic and registers rather than memory bits. **N.B.** here we talk about the on chip block memories. The cost of storing capabilities in the software accessible memory are discussed in the following paragraph.

3.3.4 Memory overhead

CHERI capabilities are a burden on the memory subsystem. Specifically, there are two ways in which supporting the CHERI model has a significant impact: capability sizes and their memory footprint with respect to conventional integer pointers, and tagging of the memory.

Tagged memory overheads

In a capability machine, the currently executing task is granted its rights to perform specific actions by the capability list it has access to. In CHERI, the rights are defined by the memory accessible through the capabilities to code and data memory. It is important that these capabilities are not modified by external processes that would not have access to them in the first place. Specifically, capabilities stored in memory risk being overwritten. One mechanism in place to enforce the unforgeability property of capabilities in CHERI is the use of a tagged memory.

Conceptually, for each 256 bits of memory, one extra bit holds the information of whether the 256 bits of memory represent standard data or a valid capability. When writing a valid capability to memory through a “write capability” instruction, this tag bit will be set. Any standard write to memory will clear this tag, effectively making it impossible to forge a capability and for example increase the level of permissions it provides by overwriting its **perms** field. These tag bits do not exist in a non capability system, and are therefore an overhead.

Since one extra bit is required for each 256-bit chunk of memory, the storage overhead required is relatively small : $\frac{1}{256}$, that is less than 0.4%. CHERI currently splits main memory in two parts that hold respectively the 256-bit data chunks and the associated tags, and uses an extra layer of caching for tag bits. Possible optimisations will be studied in **Chapter 5**.

Capability size memory overheads

To fully exploit CHERI, an application should use capabilities in place of conventional integer pointers. A modified LLVM C compiler [21] allows us to compile C programs that use capabilities in place of pointers. This can be done selectivelly on some pointers only, or for all data pointers (including heap and stack allocations) to enforce spatial memory safety, and for all return addresses and function pointers to enforce Control-Flow Integrity (CFI) [2].

On a 64-bit architecture, pointers that used to be 64-bit values are now replaced by 256-bit capabilities. This effectively means that pointers are now 4 times as big as they used to be. The memory bandwidth requirements are therefore greater when using capability code.

I evaluate the current CHERI implementation using the FPGA prototype presented in **Section 3.3.2**.

Figure 3.4 shows overheads in bytes fetched by the processor for CHERI code with respect to MIPS code. These overheads are with respect to MIPS code compiled for and running on a 256-bit CHERI processor bitfile. To make the comparison as fair as possible, MIPS built for 256-bit CHERI can make use of 256-bit capability registers for `memcpy()`.

Most applications see memory overheads when running with capabilities. We see that pointer heavy benchmarks such as those from the Olden benchmark suite have an overhead much greater than other benchmarks as their working set is particularly sensitive to pointer size.

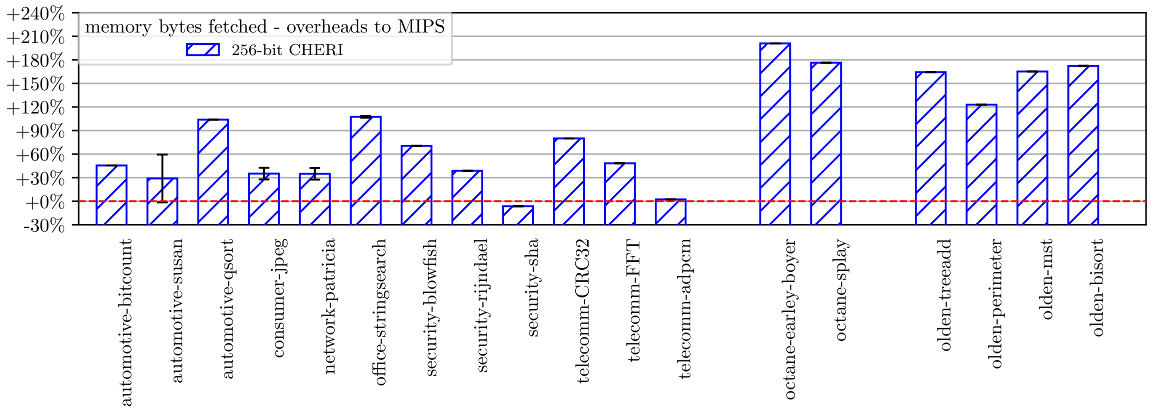


Figure 3.4: *Data memory bytes fetched overhead for 256-bit CHERI with respect to MIPS*

The presented results need to be considered in light of the dynamic instruction counts. **Figure 3.5** shows the difference in number of instructions executed in CHERI binaries with respect to MIPS binaries. It showcases the state of maturity of the CHERI compiler toolchain and SDK which is constantly improving but can still explain some of the odd results. The ability to leverage the CHERI capability instructions and registers is not necessarily systematically leveraged in the MIPS case, and can lead to some unfair advantages for CHERI code, as for example in the case of the “sha” benchmark.[5]

In “sha”, specifically in the `sha_transform()` function, the first loop performing a copy of `long`s gets unrolled in the CHERI case, leading to a flat cost of 32 instructions. In the MIPS case, the compiler generates a call to `memcpy()`. This function is a wrapper around the capability `memcpy_c()` that is able to copy capability wide chunks of memory, and comes with its own cost of 24 instructions. The memory copy itself then takes place, but only using byte copies as the array operands are double-word aligned and not capability aligned, preventing the use of `CLC` and `CSC` capability instructions and leading to a cost of 512 instructions. This is an extreme case of unfairness in favour of CHERI and clearly appears in **Figure 3.5**.

Figure 3.6 shows the share of dynamic instructions spend in a TLB handler. This graphs presents the share of dynamic instructions spent in a TLB handler for both CHERI and MIPS rather than presenting an overhead of CHERI with respect to MIPS. This is in order to avoid emphasizing differences in benchamrks that have very few TLB misses overall: a difference between 1 and 2 TLB misses lead to a 100% overhead, but may not have a significant impact on the benchmark. Note that the presented metric is computed by multiplying the number of TLB misses by 50 (an

[5]Note that the results presented in the CHERI ISCA paper [100] (Figure 3) are estimations based on standard MIPS instruction traces, where information relevant to bounds checking was extracted, and extra memory accesses and instruction fetches were simulated. There was no CHERI-aware compiler, which explains the optimistic results presented, and the difference that exists with the number presented in this document (particularly in this section and in **Section 4.5**), which come from measurements of on chip counters, running CHERI-aware code.

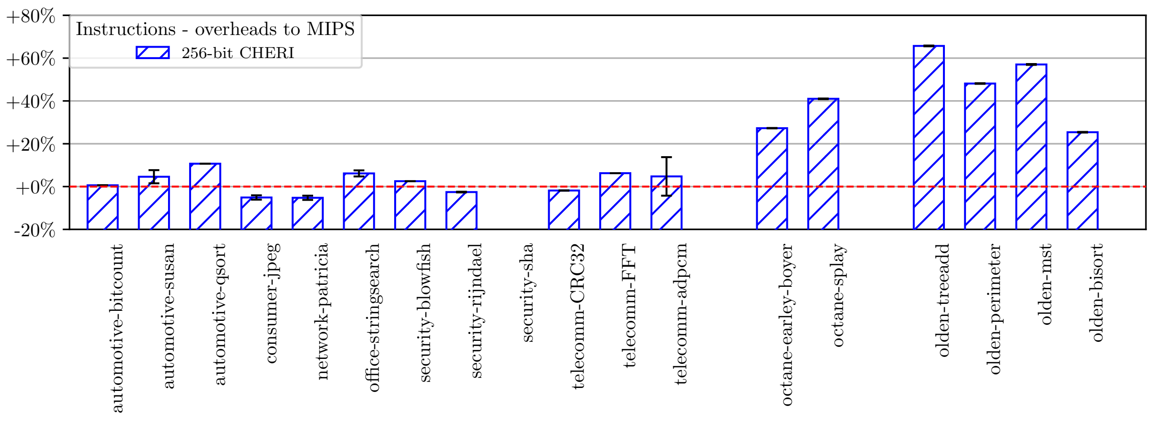


Figure 3.5: *Dynamic instructions count overhead for 256-bit CHERI with respect to MIPS*

approximation of the number of instructions ran in a TLB handler[6]) and dividing it by the number of dynamic instructions in the benchmark.

Applications with large working sets that contain a lot of pointers will see an increased TLB pressure. As we can se clearly on **Figure 3.6**, this effect is particularly present in the pointer heavy Olden benchmarks.

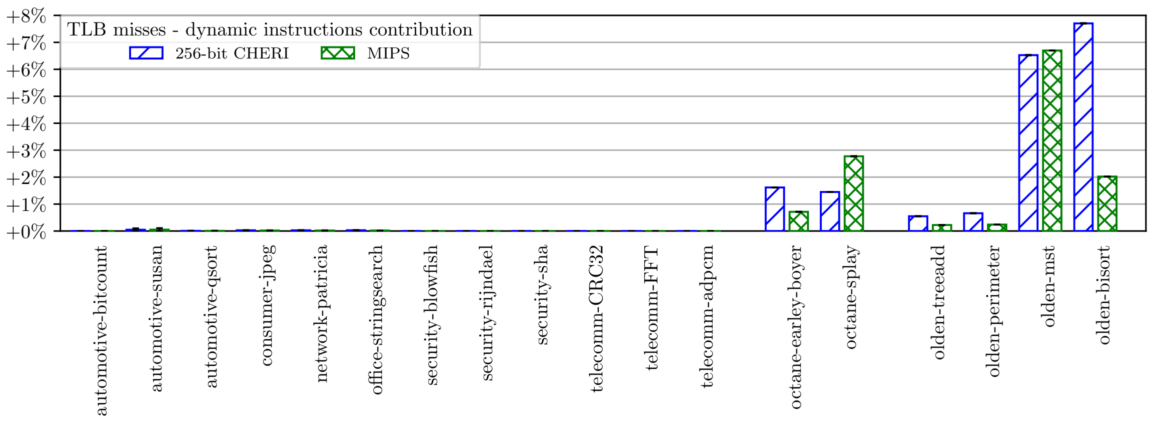


Figure 3.6: *TLB miss contributions to dynamic instruction count, effectively the share of dynamic instructions spent in a TLB handler*

Figures 3.7 and **3.8** show the impact of 256-bit CHERI capabilities on the cache hierarchy. Pointer heavy applications see overheads up to more than 250% with respect to MIPS. More typical applications, specifically from the Mibench suite, see more modest overheads. Overall, we observe that the use of capabilities has a relatively limited impact on applications manipulating few pointers and that pointer heavy applications suffer more significantly.

Figure 3.9 presents DRAM traffic overheads with respect to MIPS. Note that even if MIPS code does not rely on capability validity tags, these are still fetched from the DRAM by the hardware itself. **Chapter 5** explains the tagged memory

[6]The number of instructions in the FreeBSD/CheriBSD TLB miss handler was measured on a dynamic instruction trace of the bisort benchmark. When running under contention, the most common TLB misses for already allocated pages consistently required 47 instructions. 50 is an overestimation to cope with the more rare TLB miss events that would perform more work.

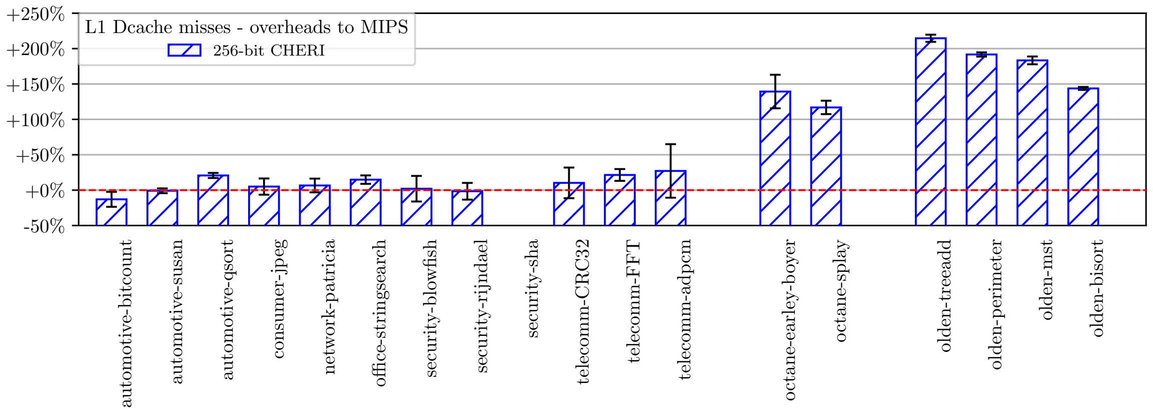


Figure 3.7: *L1 data-cache miss overhead for 256-bit CHERI with respect to MIPS*

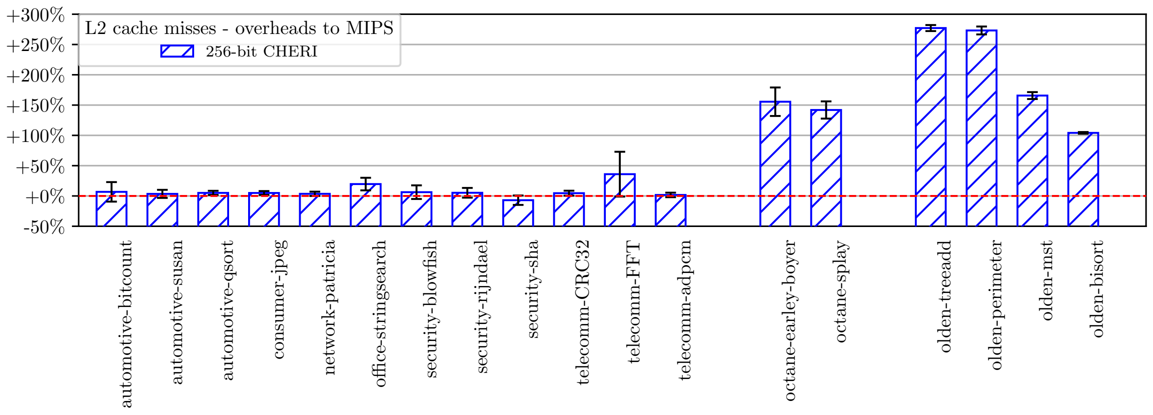


Figure 3.8: *L2 cache miss overhead for 256-bit CHERI with respect to MIPS*

mechanisms in greater details. Ultimately, this graph is very similar to **Figure 3.8**, showing that the added tag traffic is a small fragment of DRAM traffic.

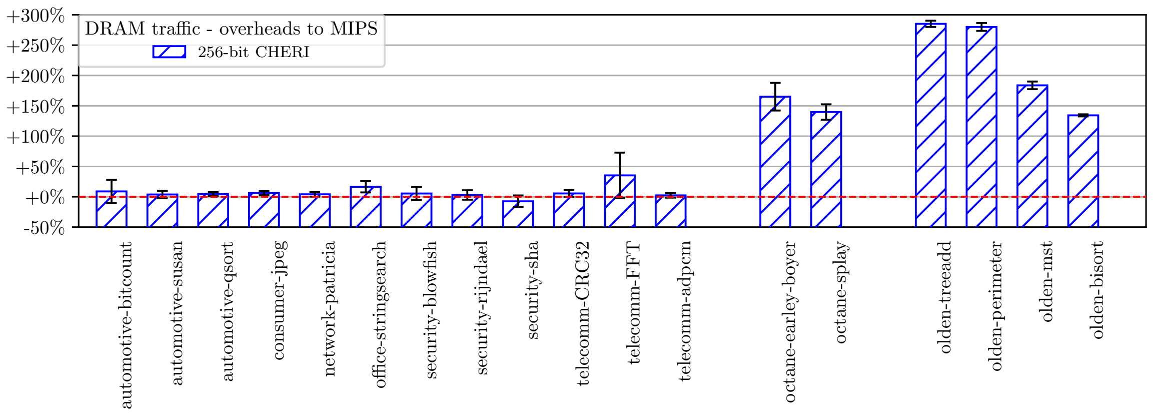


Figure 3.9: *DRAM traffic overhead for 256-bit CHERI with respect to MIPS*

As these results show, using 256-bit capabilities for pointers has a noticeable impact on the memory sub-system. An obvious way to mitigate these memory requirements is to make capabilities smaller. We will later see in **Section 4.5** how these numbers compare to a 128-bit compressed capability implementation.

3.4 Summary

The CHERI processor is a capability machine introducing a new architectural primitive that can be targetted by compilers. It has several implementations at different levels of abstraction, and is currently packaged as a MIPS ISA extension.

A formal L3 architectural model serves as a golden model, and provides easy and fast experimentation for new ideas. A Bluespec implementation enables rapid RTL development and FPGA prototyping. Building and testing of the various CHERI tools and models are automated within the Jenkins continuous integration framework.

CHERI provides memory safety at the architectural level, with some microarchitectural costs that can still be mitigated. The next chapters will focus on minimizing some costs in order to make CHERI more appealing as a memory safe computer system.

Chapter 4

Compressed Capabilities

The original CHERI capability format was 256-bits long allowing easy experimentation with fields. However, this verbose format has a negative impact on memory footprint which in turn increases cache miss rate, thereby impacting performance. In this chapter we explore a 128-bit compressed format to mitigate this loss of performance.

I explore existing pointer compression mechanisms and their limitations. I present a first naive implementation of the compressed capability mechanism. I then refine this implementation and present a mature CHERI-128 mechanism.

4.1 Pointer compression methods

The most significant cost of replacing 64-bit pointers with 256-bit capabilities is the memory footprint overhead. Here, we explore more compact structures to mitigate this overhead.

In the original implementation of 256-bit CHERI capabilities, the bounds and the pointer of a capability were all accurately represented with three 64-bit wide fields **base**, **length** and **offset**. Another 64 bits were used for some additional permissions and other capability fields.

We observe that there exist some redundancy between the bounds and the pointer fields. The most significant bits are often similar, especially in capabilities to small regions of memory. This observation suggests that it should be possible to accurately store the pointer along with some reduced amount of relevant information to regenerate the actual bounds of the capability. Doing this would allow us to trade the 64-bit **base** and **length** fields for smaller fields and a controlled impact on our ability to accurately represent memory regions.

Additionally, other fields such as the **perms** or the **otype** could see their sizes reduced to save space. We aim for as few architecturally visible capability features as possible to be affected by our target compression scheme. The main expected gain is from the redundancy between the pointer and the bounds. We will now explore a few encodings schemes that can help us use this property and mitigate the costs of 256-bit capabilities.

4.1.1 M-Machine compression scheme

The M-Machine introduced in **Section 2.5.5** was an early attempt at implementing compressed bounded pointers with hardware support. The encoding is presented in **Figure 4.1**. In this approach, a 6-bit length (L) field and a 54-bit address field are used to represent a $(54 - L)$-bit segment and a L-bit offset in that segment. This would allow a very compact representation of the **base**, **length** and **offset** fields. A restriction imposed by this scheme is that the represented memory region must have a power-of-two size, and its address must be aligned on this size. Byte granularity bounds checking is made impossible with this approach for arbitrary object sizes. This approach also comes with some memory fragmentation problems. Objects of arbitrary non-power-of-two size will potentially require rounding up to still enforce memory safety: an object of 33 bytes will have to be allocated a 64-byte segment, effectively wasting almost half of the segment's memory. This fragmentation limitation is mitigated in the M-Machine paper by observing that it takes place in the virtual memory space rather than in physical memory[14].

4.1.2 Low-Fat pointer compression scheme

The Low-Fat pointer scheme [57] addresses the fragmentation issue from the M-Machine approach by using a floating point representation. Low-Fat pointers have a 6-bit "**B**" field that encodes a block size (effectively an exponent in a floating point

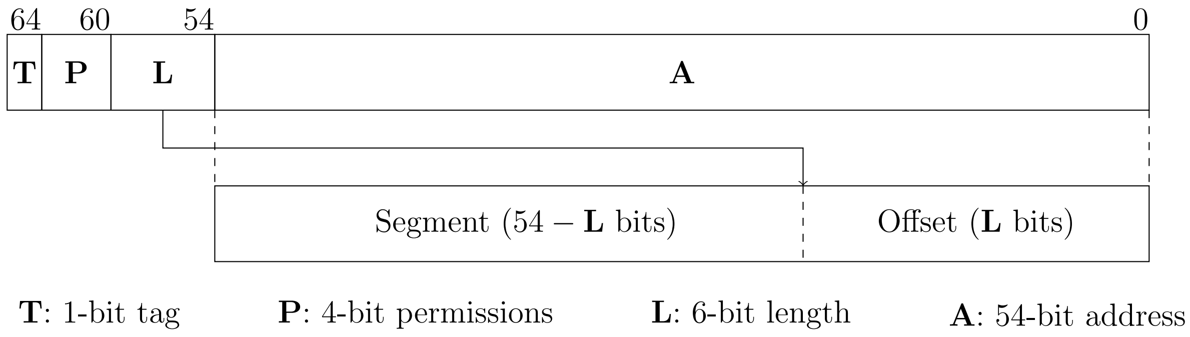
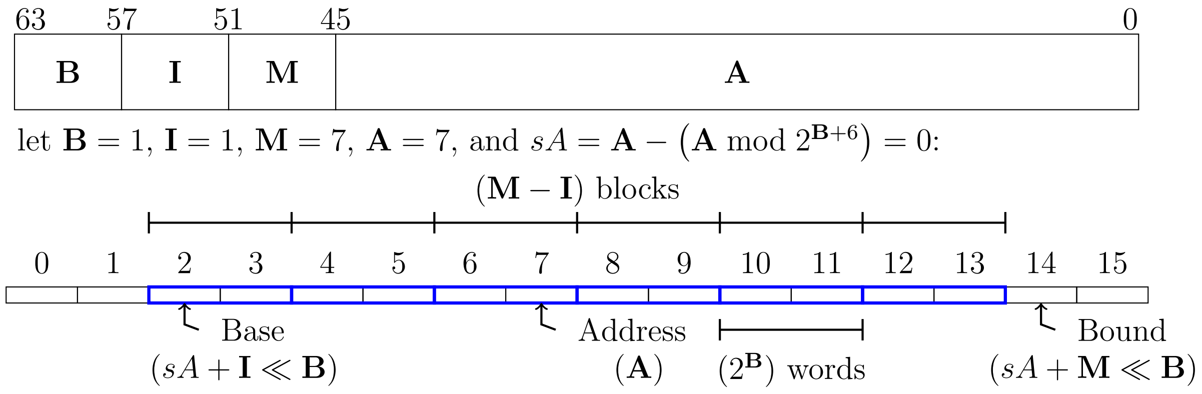


Figure 4.1: *An M-Machine pointer with a* $(54-\boldsymbol{L})$*-bit segment number and a* $(\boldsymbol{L})$*-bit offset both encoded within the address* $\boldsymbol{A}$

scheme). The address "**A**" of the pointer is precisely encoded, and two extra 6-bit fields "**I**" and "**M**" contain the bits to be substituted in the address to get respectively the base and the bound of a region as shown on **Figure 4.2**. **B** expresses how much to shift **I** and **M** left before replacing the bits in the address to retrieve the base and bound addresses. The bottom **B** bits are implied zeroes. The Low-Fat pointer paper [57] mentions a carry, potentially required to adjust the upper bits of **A** to match the desired base or bound address[1]. The Low-Fat pointer encoding scheme can represent more fine grained regions of memory than the M-machine pointer encoding scheme.



63
57
51
45
0
B
I
M
A
let $\mathbf{B}=1$, $\mathbf{I}=1$, $\mathbf{M}=7$, $\mathbf{A}=7$, and $sA=\mathbf{A}-\left(\mathbf{A} \bmod 2^{\mathbf{B}+6}\right)=0$:
$(\mathbf{M}-\mathbf{I})$ blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Base
$(sA+\mathbf{I}\ll\mathbf{B})$
Address
$(\mathbf{A})$
$(2^{\mathbf{B}})$ words
Bound
$(sA+\mathbf{M}\ll\mathbf{B})$

Figure 4.2: *A Low-Fat pointer along with a use-case pointing at address 7, within the blue region of memory defining the software object being manipulated*

4.2 Requirements

The M-Machine and Low-Fat compression mechanisms have limitations that prevent them from being used as a substitution for standard pointers. Here we explore these limitations with the intention of later mitigating them through a proposal for compressed CHERI capabilities.

[1]I explain the necessity for such a correction in the paragraph about "representable regions" in **Section 4.4.2**.

4.2.1 Actual pointer size

In both the presented schemes, we note that some of the most significant bits of the pointer are used to store information on the bounds of the accessible region. This means that the full 64-bit virtual address space is no longer available to the application. A full 64-bit pointer is necessary to support the ever growing memory requirements of modern applications.

4.2.2 On dereference bounds check and out-of-bound pointers

The two presented schemes do not allow representation of out-of-bound pointers. The Low-Fat paper [57] points out that this avoids the need for bounds checking circuitry when dereferencing the pointer, as all verification occurs when manipulating the pointer itself. However, not performing bounds checks at the time of the memory access limits the kind of instructions that can be used with this approach. Specifically, instructions with address offsets as arguments (register or immediate) cannot be implemented, as the effective address of the memory access is only known when trying to perform the load or store operation. These instructions are useful for efficient execution of higher level language constructs such as "C structs" (accessing a specific struct's field via its offset within the struct) for example. The lack of such instructions in the ISA requires extra steps in the source code to perform the required pointer arithmetic, effectively growing the code size and the run time.

A second implication of these schemes is that once a pointer has gone out of bounds, it will remain invalid forever. The C11 specification actually states:

> "Pointer arithmetic that ends outside of the original object is undefined, with the exception that a pointer may point at the next element past the end of arrays. Such a pointer is however only valid for comparison, and not for dereferencing."

More generally, as observed by Chisnall et al.[21], actual C code makes use of idioms that take liberties with the C specification. Specifically, several instances of the "mask" idiom, embedding data in the low bits of a pointer and masking them off on dereference, are found in ffmpeg, FreeBSD libc, bash, perf, python, wget and zsh. Instances of the "invalid intermediates" idiom, taking a pointer past the limit of a referenced object before bringing it back in for dereference, are found in ffmpeg, libX11, FreeBSD libc, bash, libpng, tcpdump and python. There are no good measurements of how far out of bounds a pointer would go in these cases. Hongyan Xia in the CHERI team observed that libz uses an intermediate pointer value 514 bytes below the base of its described object. This worst observed case fits in the generous 4-KiB page-sized buffer proposed in **Section 4.4.3**[2]. Ultimately, for real world C code to work, it should be possible for pointers to wander out of bounds. Note that a memory safety hazard only exists at the point of dereferencing a pointer, and it is safe to simply allow a pointer to reference a memory location out of bounds if no

[2]A conservative buffer also allows for future studies to be undertaken to better identify actual use cases of out-of-bounds pointers. The L3 model also has a monitoring feature to identify out-of-bounds pointers.

dereference can occur. Therefore, taking a pointer out of its bounds and bringing it back inside them at a later point should be a supported operation.

Low-Fat [57] discusses this issue and makes an attempt at solving it. To address it, they use their Hardware Type Unit that can manipulate multi-bit tags on each memory word to implement a new dedicated “Out-of-Bounds-Memory-Location” hardware type. Tagging a memory word with this type prevents a pointer dereference from happening but avoids turning the pointer into an “Out-of-Bounds-Pointer”, effectively allowing further pointer manipulation in the future. By allocating one more word than requested for new objects and tagging the extra memory location with the “Out-of-Bounds-Memory-Location” hardware type, Low-Fat allows the very basic C11 specification use-case to work, but at the cost of supporting multi-bit tags on all data words, and the inability to use this word of data for any other purpose. Moreover, should you extend this mechanism to more general use-cases and tag more memory locations as “Out-of-Bounds-Memory-Location” to be used as “out of bounds buffer” for the pointer, all those memory locations could no longer store useful data (as “Out-of-Bounds-Memory-Location” are not dereferenceable).

4.2.3 Requirements for a CHERI compression scheme

The limitations that were just exposed need to be considered when deriving a compression scheme for CHERI. Our additional requirements will therefore be to preserve a full 64-bit pointer available for software use, enabling existing software to exploit the CHERI mechanism. Equally it is necessary to make sure that out-of-bounds pointers can be supported.

It is worth noting that one of CHERI’s features is to enable fast capability manipulations by making the new instructions available to user space. Even though the compression mechanism will preserve this feature, it is important to point out that the other schemes did not allow it, or simply did not implement different privilege levels. Preserving this feature for CHERI is necessary for industrial adoption.

The compression schemes which attempt to meet these requirements and that will now be presented have first been explored with the help of the CHERI L3 model (described in **Section 3.2.2**) before being implemented in BSV for FPGA prototyping. The L3 model enabled rapid design space exploration of these schemes, allowing different field size trade-offs to be considered with ease.

4.3 A first 128-bit CHERI compression scheme

To compress CHERI capabilities, we will use a floating point representation technique. We use two fields called **toBase** and **toBound**, representing signed offsets from the **address** to the base address of a memory region and to the bound address of a memory region respectively. An **e** field is used to shift left the values of **toBase** and **toBound**, and allows for various granularities to be supported. In this approach, the additional requirement that a valid pointer can go out of bounds is dealt with by using signed offsets from **address**. A pointer going out of bounds from the top of the region will have a negative **toBound** and a negative **toBase**.

Similarly, a pointer going out of bounds from the bottom of the region will have a positive **toBound** and a positive **toBase**. A pointer in bounds, being above its base and below its bound, will has a positive **toBound** and a negative **toBase**. A full 64-bit pointer is available for unsealed capabilities. For sealed capabilities (see **Section 3.1.1**), this format still uses the upper 16 bits of the **address** to store the **otype**. The different fields of this compressed CHERI representation shown on **Figure 4.3** are defined as follows:

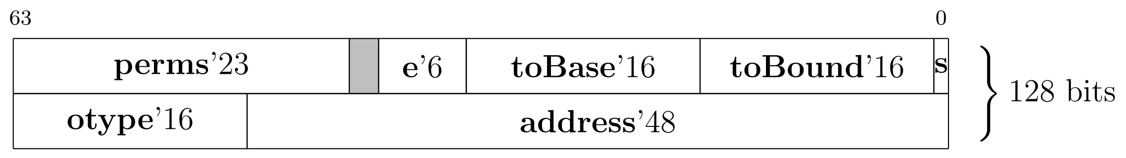


Figure 4.3: *A 128-bit CHERI capability's memory representation for the first proposed compression scheme*

toBase A 16-bit field containing a signed integer shifted left by **e** and added to **address** (with the lower **e** bits set to 0) to give the architectural **base** of the capability. This field must be adjusted upon any update to **address** in order to preserve the architectural **base** of the capability.

$$\begin{aligned}\mathbf{mask} &= -1 \ll \mathbf{e}\\ \mathbf{base} &= (\mathbf{toBase} \ll \mathbf{e}) + \mathbf{address}\ \&\ \mathbf{mask}\end{aligned}$$

toBound A 16-bit field containing a signed integer shifted left by **e** and added to **address** (with the lower **e** bits set to 0) to give the upper bound (architectural **base**+**length**) of the capability. The architectural **length** of the capability can be generated by subtracting the architectural **base** from the capability's upper bound. This field must be adjusted upon any update to **address** to preserve the architectural **length** of the capability.

$$\begin{aligned}\mathbf{mask} &= -1 \ll \mathbf{e}\\ \mathbf{base} + \mathbf{length} &= (\mathbf{toBound} \ll \mathbf{e}) + \mathbf{address}\ \&\ \mathbf{mask}\\ \text{or} \qquad \mathbf{length} &= (\mathbf{toBound} \ll \mathbf{e}) + \mathbf{address}\ \&\ \mathbf{mask} - \mathbf{base}\end{aligned}$$

address A 64-bit pointer. It can hold an absolute virtual address, equal to the architectural **base** + **offset**. It is the full 64-bit MIPS virtual address when the capability is unsealed, and a compressed virtual address when the capability is sealed. MIPS specifies the meaning of the 5 upper bits of the virtual address. When sealing a capability, these bits are placed in **address**[47:43], replacing "unused bits"[3] of the virtual address.

$$\mathbf{address} = \mathbf{base} + \mathbf{offset}$$

s A bit indicating whether the capability is sealed or not.

[3]CHERI has a 40 bit physical address space, hence the possibility to use bits above bit 39 of the **address** to store information. When unsealing, the segment bits are placed in **address**[63:59] and **address**[58:0] become a "sign" extension of **address**[42:0]. **N.B.** this goes against the requirement of providing a full 64-bit pointer to the software, but this is mitigated by only using this technique for sealed capabilities.

otype	An optional 16-bit field containing a sealed capability's object type. If **s** is cleared, the object type is implied to be zero, and this field corresponds to the upper bits of **address**.
perms	A 23-bit field with the same 15 hardware permission bits as the 256-bit version. The remaining 8 bits are user-defined permissions.
e	A 6-bit exponent for the **toBase** and **toBound** fields. The exponent **e** is used to shift **toBase** and **toBound** left before adding them to **address** in order to bounds check memory accesses or generate the architectural **base** or **length** of the capability.

This 128-bit version of the CHERI capability is meant to be functionally equivalent to the 256-bit version. It is, however, not without any repercussions on the architectural model of a capability. We now discuss these repercussions.

4.3.1 Limitations of this approach

The presented compressed capability encoding addresses the requirements expressed at the beginning of this chapter. Specifically, it allows for capabilities to reference memory locations outside of their bounds. However, for this feature to work, both the top and base addresses of the represented region need to be reconstructible from the information contained in the capability, or that capability would not be usable. Let us ignore fields of capabilities that are irrelevant to bounds representation, and consider a capability to simply consist of a triple $\{\textbf{toBound}, \textbf{toBase}, \textbf{address}\}$.

As an example, consider **toBound** and **toBase** fields that can take values in the $[-20 : 20]$ range, and a **address** field that can represent addresses in the $[0 : 100]$ range for the purpose of this example (as opposed to the actual ranges of $[-2^{15} : 2^{15} - 1]$ and $[0 : 2^{64} - 1]$). Given a capability $\{10, 0, 50\}$, one can compute its top to be 60 (effectively adding the **toBound** to the **address**) and, similarly, its base to be 50. When moving the **address**, the **toBound** and **toBase** fields are updated: moving the **address** to 54 will yield the new capability $\{6, -4, 54\}$.

It is possible to push the **address** out of the object bounds and still represent the capability. For instance, moving the **address** to 63 (which is above the object's top address) will be represented by the triple $\{-3, -13, 63\}$. However, as the ranges for **toBase** and **toBound** are finite (in our example, $[-20 : 20]$), it is not possible to represent a capability with a **address** too far from the object such as 79 : such a capability would be represented by the triple $\{-19, -29, 79\}$, where -29 is an illegal value as $-29 \notin [-20 : 20]$.

Additionally, we observe that even though the value of -29 for **toBase** is not representable, the value of -19 for **toBound** is within the authorised range. This means that some of the possible bit arrangements of a capability with this proposed encoding mechanism are not mapped to valid capabilities. This is an undesirable feature as it effectively results in wasted encoding space.

We see in **Figure 4.4** how the size of an object (in green) affects the size of the out-of-bounds buffers (in blue) for the **address**, and the amount of wasted space (in

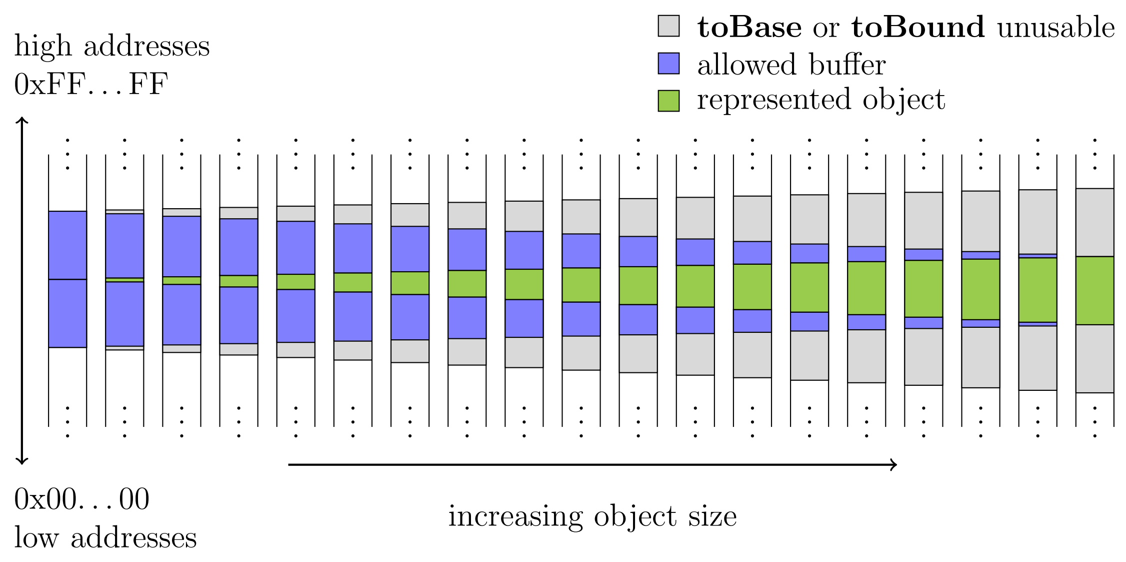


Figure 4.4: *Limitations of the first naive compression scheme: wasted encoding space (in grey). The wasted space grows and the available buffers (in blue) shrink as the object's size increases*

grey). In this figure, an address can wander in the green and blue regions as the base and top of the object can be reconstructed safely from there. One can consider the "reachable by toTop" and "reachable by toBase" regions, two sliding windows drifting apart as the length of the represented object grows. The intersection of those two region defines the region from which we can safely re-derive our uncompressed capability (i.e. the union of the green and blue regions).

The very left end of **Figure 4.4** has those two regions sitting on top of each other. From left to right, as the object size increases, the "reachable by toTop" region gets slightly offset towards the higher addresses and the "reachable by toBase" region gets offset in the opposite direction. We see that the (blue) out-of-bounds buffers above and below the object shrink as the object grows. Additionally, (grey) regions of the size of the object appear above and below the allowed buffers, representing the wasted bit patterns of the encoding for which only one of the two bounds of the object could be reconstructed.

Overall, we see that smaller objects best exploit the available bit patterns, and that this issue scales poorly with the size of the object in the proposed encoding scheme.

We note that it is possible to guarantee the presence of out-of-bounds buffers (avoiding the right most cases in **Figure 4.4**). This can be achieved when deriving the exponent of a new capability (at the time of `CSetBounds`), by artificially inflating the requested length on the fly. This makes sure that the derived exponent will be able to cope with both the object's actual size and some extra buffer memory. The cost of this technique is that of requiring a larger exponent faster, effectively sacrificing precision.

4.3.2 Architectural repercussions

As the top and bottom of the memory region described by a capability are now encoded as 16-bit signed offsets (**toBound** and **toBase**) from the currently pointed location (i.e. **address**), the ability to accurately describe this region is affected. As long as the top and bottom of the region are no further than $\pm 2^{15}$ bytes away from the **address** (that is 15 bits worth of range for our 16-bit signed integer fields), the region can be described precisely, with a byte granularity. For larger regions, the **e** field will define how many times to shift the **toBound** and **toBase** fields, describing $2^{\mathbf{e}}$ aligned regions, effectively sacrificing granularity.

Deriving a new capability was previously done by first increasing the base address of a base capability through the `CIncBase` instruction, and then setting a new length through the `CSetLength` instruction. We replace these instructions by a new `CSetBounds` instruction performing these two operations atomically. It requires a base capability with its offset describing the new desired base, and a desired length. The derived capability will be contained within the parent capability and have the same permissions (or fewer if further restricted later on). It is the size of the requested memory region that determines whether it is accurately representable or not. By providing the requested length through `CSetBounds` rather than after having first set the base, we can pick the best representation for the newly derived capability at the time of derivation. When the number of significant bits in the requested length is greater than the size of our **toBase** and **toBound** fields (that is more than 15 bits for the currently presented representation), a non 0 **e** is required. Setting the location and length of a capability is now done atomically, in a single instruction which also decides on the **e** field's value.

The values of **toBase** and **toBound** are now expressed in blocks of $2^{\mathbf{e}}$ bytes. This means that only regions with sizes multiple of $2^{\mathbf{e}}$ can be represented. This new requirement can be approached as follows:

- When deriving a new capability, it is possible to return a superset of the requested region (as long as it fits within base capability provided for the operation). This implies that the length provided when deriving a new capability is a lower bound of the actual architectural **length** of the allocated region. (The architectural **length** can later be queried through the `CGetLen` instruction.)

- When deriving a new capability, on detection of a non accurately representable region (that is, the request length is not a multiple of $2^{\mathbf{e}}$ for the derived **e**), it is possible to throw an exception, or return an invalid capability.

The first option appears to offer flexibility, however the ability of a compiler to keep track of the granularity of memory allocations makes the second option desirable. The compiler can pick appropriate argument combinations to the `CSetBounds` instruction when deriving capabilities, therefore only requesting accurately representable regions. The CHERI ISA now offers both options. The first option is supported through the `CSetBounds` instruction, and the second one is supported through the

`CSetBoundsExact` instruction[4]. Note that memory safety is still enforced even if there is a loss in precision.

The size of the **perms** field is shrunk, which only has few repercussions at the architectural level. All hardware permission bits are kept the same as in the 256-bit capability scheme, and it is only the number of software accessible bits that is reduced. The only architecturally visible change is therefore the number of available software permission bits (effectively going down from 16 to 8).

Finally, the **otype** field size is reduced from 24 to 16 bits. Where 256-bit CHERI allows for 2^{24} ($\approx$ 16M) different types, this 128-bit CHERI scheme can only represent up to 2^{16} ($\approx$ 65K) different types. Khilan Gudka conducted a study on the number of classes that exist in a few systems, giving some indication on the order of magnitude of the number of capability types that could be required when sealing objects. He found that Chromium v49.0.2623.110 (C++) uses around 18000 classes, around 20300 classes in OpenOffice v4.1.2 (Java and C++), and around 12400 classes in the OS X v10.11.3 Framework (ObjC). A 16-bit **otype** seems acceptable. However, between 149000 and 283000 classes were found in Android 6.0.1 build MHC19J (Java), which is an order of magnitude higher. The capability format presented in **Section 4.4** addresses this issue by using a 24-bit **otype** like 256-bit CHERI.

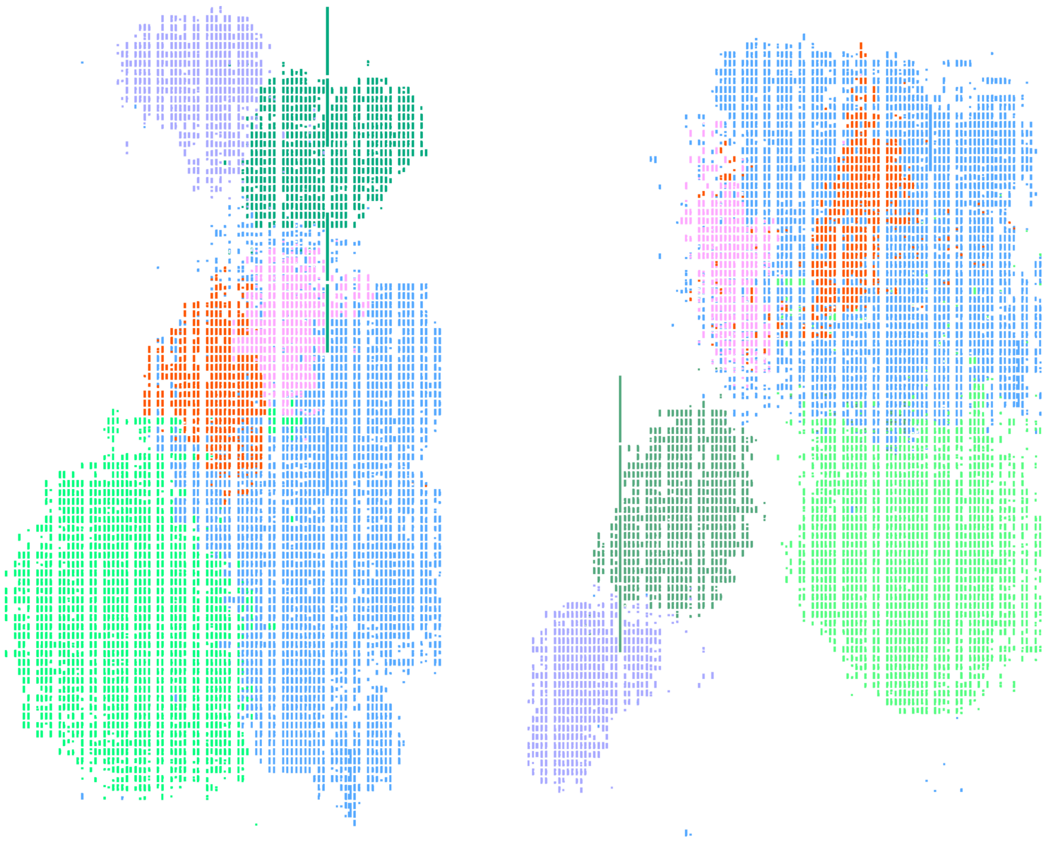
4.3.3 Micro-architectural repercussions

A further limitation of this proposed approach is that pointer manipulations involve changes in the **toBound** and **toBase** fields. This can be implemented by dynamically computing the difference between the new pointer and the old pointer values, and use it to correct **toBound** and **toBase** fields accordingly, keeping them consistent. This operation would have to be performed every time the **address** field changes. Alternatively, one can consider having an "uncompressed" capability register file, in which the actual top and base addresses are stored (rather than the compressed **toBound** and **toBase** fields) when the capability is first fetched. This approach potentially increases the "load to use" delay in a pipelined implementation as the decompression circuitry may not fit within the same stage as the one receiving the data from memory. This also implies some circuitry on the way to memory to transform back into the compressed in-memory representation format.

4.3.4 Implementation costs

Figures 4.5a and **4.5b** show the different FPGA layouts generated by a Quartus synthesis of a 256-bit CHERI and a 128-bit CHERI. **Table 4.1a** reports the resource usage of 128-bit CHERI on a Stratix IV FPGA. When compared with the results for 256-bit CHERI (as presented in **Table 3.2b**) we get a relative increase in resource usage presented in **Table 4.1b**.

[4]The `CSetBoundsExact` instruction throws an exception rather than clearing the capability tag. In the CHERI pipeline, throwing exceptions at later stages is not a problem. However, for more complex, super-scalar, out-of-order industrial processors, it might be appropriate to consider returning an invalid capability and let the standard check on memory access detect the violation. This arguably has an impact on debug-ability.



(a) *256-bit CHERI FPGA layout*

(b) *128-bit CHERI FPGA layout*

Figure 4.5: *Comparison of FPGA synthesis results for 256-bit CHERI and 128-bit CHERI. Both synthesis yield FPGA layouts with similar resource utilisation.* ***Tables 4.1a*** *and* ***4.1b*** *detail the synthesis results*

256-bit CHERI capability added a tag bit for every 256 bits of data, which represented a negligible memory overhead of 0.4% ($\frac{1}{256}$). With the newer 128-bit representation, this overhead grows to around 0.8% ($\frac{1}{128}$). Tag storage is discussed in greater detail in **Chapter 5**. Overall, we observe that the hardware cost overheads between 128-bit CHERI and 256-bit CHERI are small compared to the ones existing between 256-bit CHERI and BERI (see **Table 3.2c**), which makes this 128-CHERI scheme an appropriate option to reduce the memory bandwidth costs related to capabilities.

4.4 A mature 128-bit CHERI compression scheme

An attempt at addressing the limitations described in the previous section led to a new micro-architecture for 128-bit CHERI compressed capabilities. This section introduces the working principles of this new solution and describes its format. The main goals of this mature representation are to maximise the use of the available bit patterns as well as to limit the complexity of its implementation.

	ALUTs	ALMs	Logic Registers	Block Memory bits
CHERI TOTAL	58219	49079	38035	3463464
DCACHE	5305	4917	3417	272640
ICACHE	2822	2634	1522	273436
L2CACHE	5058	5665	4383	2179124
TAGCACHE	8985	7361	5408	283992
CAPCOP	11654	9628	5105	11296
MIPSCORE	24395	18874	18200	442976

(a) *128-bit CHERI resource usage*

	ALUTs	ALMs	Logic Registers	Block Memory bits
CHERI TOTAL	-1.70 %	-0.14 %	-4.38 %	0.18 %
DCACHE	-0.79 %	-2.54 %	-0.78 %	0.38 %
ICACHE	1.84 %	3.25 %	2.56 %	0.38 %
L2CACHE	-13.51 %	-3.48 %	0.25 %	0.38 %
TAGCACHE	-4.72 %	-3.22 %	1.54 %	0.37 %
CAPCOP	1.78 %	5.85 %	-23.69 %	-31.72 %
MIPSCORE	0.06 %	-0.56 %	-1.42 %	0.00 %

(b) *128-bit CHERI resource usage increase relative to 256-bit CHERI*

Table 4.1: *128-bit CHERI raw FPGA resource usage*

We now refer to the Low-Fat encoding presented in **Section 4.1**. The approach used two **I** and **M** fields substituted in the address **A** of a pointer to represent the base and top of the associated object. The **I** and **M** fields are fixed for any given object, therefore removing the need for a dynamic update on pointer modification, as required by the signed offset approach of the first compressed 128-bit CHERI approach. Building upon this feature of the Low-Fat encoding, we can avoid the limitations of the original Low-Fat scheme and achieve the goals we set:

- The Low-Fat encoding avoids the need for dynamic updates of the top and base fields where the first presented scheme requires updating of these fields on every pointer modification. Implementation of pointer modification operations are thus greatly simplified.
- The top and base fields in the first encoding scheme are tied together in the sense that taking one away from the pointer brings the other closer to preserve the object size. Because of this behaviour, and as already seen in **Figure 4.4**, some of the values of those fields are never used, wasting encoding space. The Low-Fat encoding happens to limit this by virtue of having these two fields independent from each other, allowing all possible values to be taken without risking to map to an invalid capability representation.
- By making the pointer manipulation operations use the compressed fields of the capability, the decompression costs are avoided and the required operations

can be implemented with simpler logic, on a reduced amount of bits.

I use the Low-Fat ideas and add scalable support for out-of-bounds pointers, without requiring to make memory unusable and without requiring to store extra tag types for the memory being used as out-of-bounds buffer (see **Section 4.2.2** for the Low-Fat approach). I add support for the other CHERI features and present a 128-bit CHERI format. I also introduce further optimisations to make more bits available to invest in future CHERI features (particularly, an alternative exponent encoding presented in **Appendix A**).

4.4.1 High level working principles

In this section, we derive a means to efficiently derive the **base** and **top** addresses (where **base** < **top**) from a full pointer **address** and small $\mathbf{base_{bits}}$ and $\mathbf{top_{bits}}$ fields.

Let us consider two sets of values: a set of "full-sized" values where each element can have a large range (e.g. 64-bit addresses), and a set of "reduced-sized" values where each element can only have a limited range (e.g. 20-bit offset values). From the first set will be drawn full-sized values that describe a unique memory location but that require a large number of bits to be represented. From the second set will be drawn reduced-sized values that can only describe a location relatively to a full-sized value, but representable in a smaller number of bits. In the following paragraphs, we will describe how the operations that previously relied on full-sized values can be performed with reduced-sized values (hence allowing for a cheaper hardware implementation), and still allow us to re-derive all the relevant full-sized values when necessary.

First, we introduce the relation linking full-sized addresses and reduced-sized offsets. It uses simple modulo arithmetic and will be exploited throughout this section.

An arbitrary memory location's address *loc* (drawn from the set of full-sized values) in an arbitrary region of size S, can be viewed as an offset loc_{off} (drawn from the set of reduced-sized values) from the immediately inferior S-aligned address loc_{align} (drawn from the set of full-sized values). We can map a full-sized value to a reduced-sized value in a S-sized region using this property as exposed in **Equation** (4.1).

$$loc = loc_{align} + loc_{off} \qquad \text{where} \qquad loc_{align} = loc - (loc \bmod S) \tag{4.1}$$

Any memory location within a power-of-two S-sized region will share some redundant top bits. These common bits describe an S-aligned memory location loc_{align}. For a 64-bit *loc* and $S = 2^{20}$, **Equation** (4.1) can be rewritten as **Equation** (4.2).

$$\begin{aligned} loc_{align} &= loc[63:20] \ll 20 \\ loc_{off} &= loc[19:0] \end{aligned} \tag{4.2}$$

We can exploit the redundancy that exists between several *loc* sharing the same loc_{align} by only storing one of them as a full-sized value (i.e. its loc_{align} and its loc_{off})

along with a set of reduced-sized values counterpart for the others (i.e. only their loc_{off}). It follows that representing capabilities can be done by keeping one full-sized value for the pointer **address** and two reduced-sized values for the bounds $\mathbf{base_{bits}}$ and $\mathbf{top_{bits}}$, where the "$\mathbf{_{bits}}$" subscript represent the notion that the value is represented as only a few middle bits. Conceptually, those few bits (and possibly implied zeroes in extra bottom bits as explained below) correspond to the loc_{off}, and the ignored top bits with implied zeroes as bottom bits correspond to loc_{align}.

As a consequence of keeping **address**, we trivially get access to $\mathbf{address_{bits}}$ which is the reduced-sized counterpart of **address**. $\mathbf{base_{bits}}$ and $\mathbf{top_{bits}}$ can simply replace the bottom bits of **address** and act as offsets from the loc_{align} associated with the **address**, **with the caveat** that a reduced-sized value may have too small a range to capture the size of the represented object, leading to some shifting and alignment requirements as explained in the following paragraphs.

base and **top** can be further apart than what can be represented by the length of $\mathbf{base_{bits}}$ or $\mathbf{top_{bits}}$. Just like the previous encoding scheme, this is handled by using an exponent **e** to amplify the value represented by $\mathbf{base_{bits}}$ and $\mathbf{top_{bits}}$, with some impact on the accuracy of the represented addresses: it is now required to set the lower **e** bits of **address** to zero, and replace the corresponding bits of **address** ($\mathbf{address}[(length(\mathbf{base_{bits}}) - 1) + \mathbf{e}{:}\mathbf{e}]$) as opposed to simply replacing its bottom bits. We note that the size S of the reduced-sized set of values is a function of **e**:

$$S = 2^{length(\mathbf{base_{bits}})+\mathbf{e}}$$

We now consider a "natural alignment region", that is a region of size **S**, starting at an address which is a multiple of **S**. When **address** is in the same natural alignment region as **base**, it is possible to reconstruct **base** from $\mathbf{base_{bits}}$ and **address**: **base** and **address** share the same loc_{align} already captured in **address**, and the specific loc_{off} of **base** is captured in $\mathbf{base_{bits}}$. By a similar reasoning, **top** can be reconstructed from **address** and $\mathbf{top_{bits}}$.

When the **base** or **top** address being reconstructed belongs to a different natural alignment region than the **address**, the associated loc_{align} is no longer directly available through **address**. It is however possible to reconstruct it with a few extra computations. We will now completely specify the new mechanism by describing how one can systematically reconstruct the **base** and **top** addresses, and how bounds checking can be performed using only reduced-sized values.

4.4.2 Detailed explanation of the compression mechanism

Let us call the "representable region" the **S**-sized region available for the pointer to move in. The representable region, from its $\mathbf{base_{edge}}$ to its $\mathbf{top_{edge}}$, must span the object being described by the capability as well as memory locations both below and above the object to allow for out-of-bounds pointers to exist. Such a representable region can be positioned in many different ways relatively to the currently described object.

Figure 4.6 shows an object (in green) and its associated representable region (in blue). In the leftmost sub-figure, $\mathbf{base_{edge}}$ and **base** are superposed; no buffer

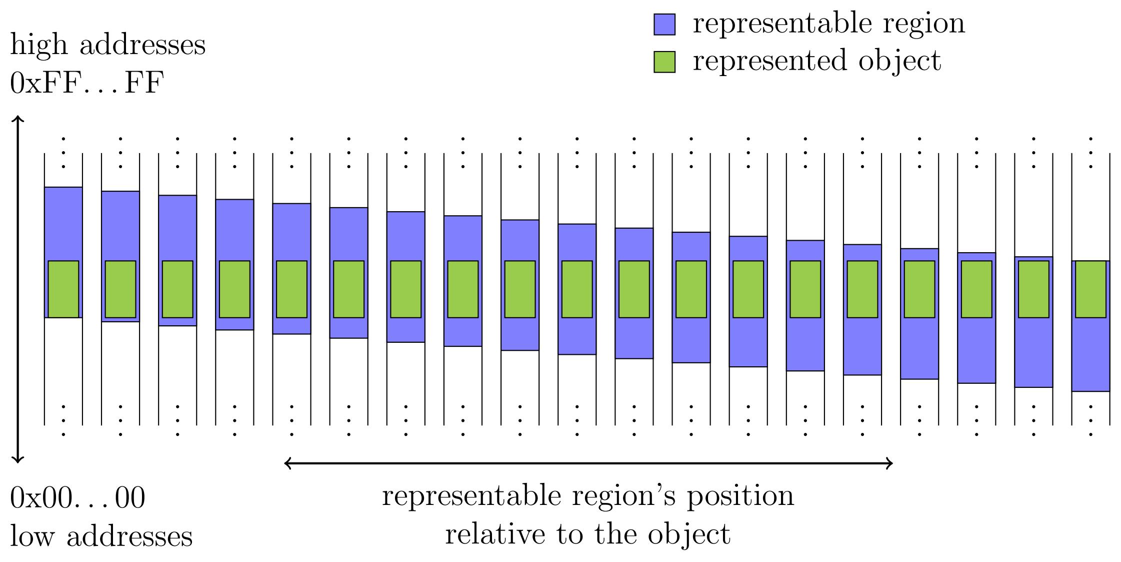


Figure 4.6: *Representable region's position relative to the represented object. To the left, representable regions allow for more buffer above the object, and to the right for more buffer below*

is available for the pointer to go below **base**, and a large buffer is available to go above **top**. In the rightmost sub-figure, $\mathbf{top_{edge}}$ and **top** are superposed; no buffer is available for the pointer to go above **top**, and a large buffer is available to go below **base**. The other cases show the various trade-offs available with, in the middle, a representable region centred around the object, with as much available buffer for out-of-bounds pointers above and below the object.

We observe that $\mathbf{base_{edge}}$ and $\mathbf{top_{edge}}$ are **S** apart by definition, or $\mathbf{top_{edge}} = \mathbf{base_{edge}} + \mathbf{S}$. This means that the reduced-sized counterpart of both $\mathbf{base_{edge}}$ and $\mathbf{top_{edge}}$ for a **S**-sized space of reduced-sized values will map to the same value $\mathbf{edge_{bits}}$ (where the use of the "$_{\mathbf{bits}}$" subscript is explained in **Section 4.4.1**), where

$$\begin{aligned}\mathbf{edge_{bits}} &= (\mathbf{top_{edge}} \bmod \mathbf{S}) \gg \mathbf{e} \\ &= (\mathbf{base_{edge}} \bmod \mathbf{S}) \gg \mathbf{e}\end{aligned}$$

The right shift by **e** (or division by $2^{\mathbf{e}}$) is present to slice the value at the appropriate position, ignoring the bottom bits for future computations. Disambiguation is made possible by the fact that $\mathbf{top_{edge}} = \mathbf{base_{edge}} + \mathbf{S}$. Furthermore, we observe that $\mathbf{base_{edge}}$ or $\mathbf{top_{edge}}$, and therefore $\mathbf{edge_{bits}}$, can be expressed relative to the represented object, as a function of one or more of the values characterising it (**base**, **top**). In other words, It is possible to fully qualify the representable region with one single reduced-sized value which is a function of the information already qualifying the object.

Allowing for the representable region's position to be dynamically determined would go against the requirement for simplicity of the hardware needed to implement the mechanism. Selecting one point on the spectrum of possible representable regions will allow for the circuitry necessary to qualify the representable region (that is, to generate the single reduced-sized value $\mathbf{edge_{bits}}$) to remain simple.

The leftmost and rightmost points of the spectrum (**Figure 4.6**) allow for a very straight-forward implementation, as in these cases, $\mathbf{edge_{bits}}$ directly map to the appropriate bits of the address of **base** or **top**. Unfortunately, these cases suffer from the lack of a buffer on one side of the object.

The middle case presents a representable region centred around the object. Here, both edges of the representable region are half a region size away from the middle of the object. We have

$$\mathbf{edge_{bits}} = \left(\left(\frac{\mathbf{base} + \mathbf{top}}{2} \pm \frac{\mathbf{S}}{2}\right) \bmod \mathbf{S}\right) \gg \mathbf{e}$$

In this case, the space available for the buffers is symmetrically distributed around the object, but the complexity of the required circuitry to compute $\mathbf{edge_{bits}}$ is increased.

In the remaining cases, we have an asymmetric distribution of the space for the buffers above and below the object. The remaining left half has more buffer available for accesses going above **top**, and the remaining right half has more buffer available for accesses going below **base**. In both cases, $\mathbf{edge_{bits}}$ can be defined as a fixed distance to one of the object bounds. In particular, when considering the left half:

$$\mathbf{edge_{bits}} = ((\mathbf{base} - X) \bmod \mathbf{S}) \gg \mathbf{e}$$

with

$$X < \mathbf{base} - \left(\frac{\mathbf{base} + \mathbf{top}}{2} \pm \frac{\mathbf{S}}{2}\right)$$

This last case has the advantage of still having buffers available above and below the object. Moreover, X can be an arbitrary function that could take into account, say, the size of the object that we aim to represent. Being concerned about ease of implementation, we can pick X to be a constant providing a minimum size for a fixed buffer, and a straightforward hardware implementation (see **Section 4.4.3**).

We note that constraining $\mathbf{edge_{bits}}$ could lead to further simplification. Specifically, forcing some alignment requirements on $\mathbf{edge_{bits}}$ will lead to $\mathbf{edge_{bits}}$ values with some zeroes in their least significant bits, making the arithmetic involving that value to operate on fewer bits. Given a desired guaranteed buffer size, a general expression of an $\mathbf{edge_{bits}}$ value implementing this idea would be:

$$\mathbf{edge_{bits}} = ((Y - Z) \bmod \mathbf{S}) \gg \mathbf{e}$$

with

$$Y = \mathbf{base} - (\mathbf{base} \bmod Z)$$
$$Z = \text{guaranteed buffer size}$$

Here, all the significant bits of $\mathbf{edge_{bits}}$ exist in the top $(\mathit{length}(\mathbf{base_{bits}}) - \mathbf{buffer_{bitsize}})$ bits (where $\mathbf{buffer_{bitsize}}$ is the number of bits required to represent Z the guaranteed buffer size, for values of $\mathbf{buffer_{bitsize}} < \mathit{length}(\mathbf{base_{bits}})$). This means that arithmetic operations involving $\mathbf{edge_{bits}}$ can be turned into $(\mathit{length}(\mathbf{base_{bits}}) - \mathbf{buffer_{bitsize}})$ bit arithmetic.

Now that we know how to qualify a representable region, we introduce one last new notion that will then allow us to systematically re-derive full-sized addresses within a representable region from their reduced-sized counterpart.

A representable region can either correspond to one natural alignment region or span exactly two natural alignment regions (see **Figure 4.7**). In the general case, one can consider two **sub-region**s of a representable region on each side of a spanned natural alignment boundary. The **sub-region** of the representable region that is above the spanned natural alignment boundary is the "**hi-region**", and the one below is the "**low-region**". As we already stated, a full-sized value *loc* representing a specific location in a given representable region can be represented as a pair of:

- an offset loc_{off}, the reduced-sized counterpart of *loc*.
- an aligned address loc_{align}, the full-sized address to which we add loc_{off} to reconstruct *loc*

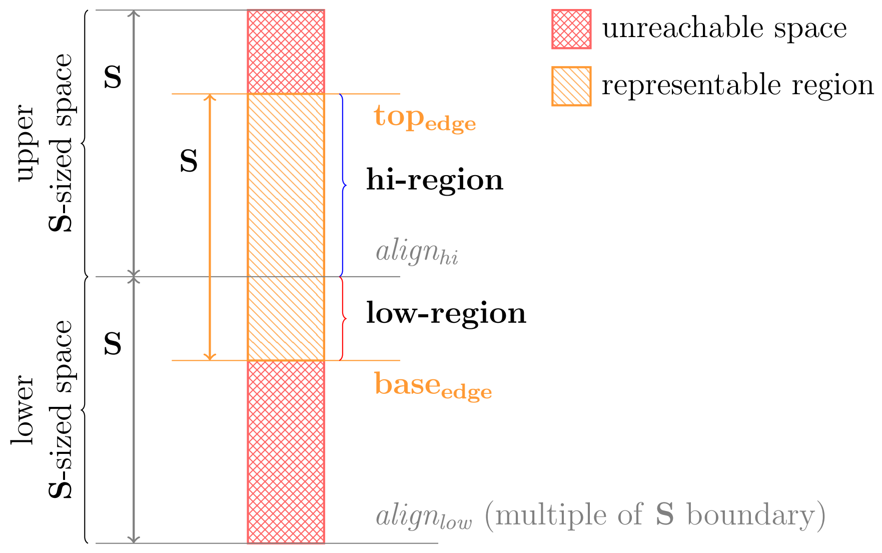


Figure 4.7: *Two* ***sub-regions*** *within the* ***S****-sized representable region. Multiple of* ***S*** loc_{align} *addresses like* $align_{low}$ *and* $align_{hi}$ *separate* ***S****-sized spaces.* $align_{hi}$ *separates the representable region in the* ***hi-region*** *(between* $align_{hi}$ *and* $\mathbf{top_{edge}}$*, in the upper* ***S****-sized space) and the* ***low-region*** *(between* $align_{hi}$ *and* $\mathbf{base_{edge}}$*, in the lower* ***S****-sized space)*

Let us call $align_{hi}$ the loc_{align} associated with a *loc* located in the **hi-region**. $align_{hi}$ is exactly the alignment boundary splitting the representable region into two **sub-region**s. Let us call $align_{low}$ the loc_{align} associated with a *loc* located in the **low-region**. $align_{low}$ is the alignment boundary immediately underneath the representable region. **Figure 4.7** shows two **S**-sized spaces spanned by a representable region delimited by its $\mathbf{base_{edge}}$ and $\mathbf{top_{edge}}$, and situates **hi-region** in blue and **low-region** in red accordingly, as well as $align_{hi}$ and $align_{low}$.

As loc_{off} is a reduced-sized value, it should identify a unique *loc* within the **S**-sized representable region. Some of the loc_{off} will identify a *loc* located in the **hi-region**, and some will identify a *loc* in the **low-region**.

Observe that when loc_{off} is smaller than $(\mathbf{edge_{bits}} \ll \mathbf{e})$[5], $align_{hi} + loc_{off}$ identifies a loc within the representable region (specifically in the **hi-region**), and $align_{low} + loc_{off}$ identifies a loc outside the representable region (below $\mathbf{base_{edge}}$). Similarly, when loc_{off} is greater than $(\mathbf{edge_{bits}} \ll \mathbf{e})$, $align_{hi} + loc_{off}$ identifies a loc outside the representable region (above $\mathbf{top_{edge}}$), and $align_{low} + loc_{off}$ identifies a loc within the representable region (specifically in the **low-region**).

In other words, given the loc_{off} of an arbitrary location within a representable region, it is possible to determine its associated **sub-region** by comparison with $(\mathbf{edge_{bits}} \ll \mathbf{e})$:

$$region(loc_{off}) = \begin{cases} \textbf{hi-region} & \text{if} \quad loc_{off} \leq (\mathbf{edge_{bits}} \ll \mathbf{e}) \\ \textbf{low-region} & \text{if} \quad loc_{off} > (\mathbf{edge_{bits}} \ll \mathbf{e}) \end{cases}$$

We can now derive the systematic way of reconstructing any full-sized address (including **base** and **top**) within a representable region from a reduced-sized value. Given **address** (its associated reduced-sized value $\mathbf{address_{bits}}$, and **S**-aligned counterpart $address_{align}$) and an arbitrary loc_{off}, we derive the expression for loc (the full-sized location represented by loc_{off}):

$$loc = \begin{cases} \textbf{if } (region(\mathbf{address_{bits}}) = region(loc_{off})) \textbf{ then} \\ \qquad address_{align} + loc_{off} \\ \textbf{if } (region(\mathbf{address_{bits}}) = \textbf{low-region}) \wedge (region(loc_{off}) = \textbf{hi-region}) \textbf{ then} \\ \qquad align_{hi} + loc_{off} = address_{align} + \mathbf{S} + loc_{off} \\ \textbf{if } (region(\mathbf{address_{bits}}) = \textbf{hi-region}) \wedge (region(loc_{off}) = \textbf{low-region}) \textbf{ then} \\ \qquad align_{low} + loc_{off} = address_{align} - \mathbf{S} + loc_{off} \end{cases}$$

where

$$address_{align} = \mathbf{address} - (\mathbf{address} \bmod \mathbf{S})$$

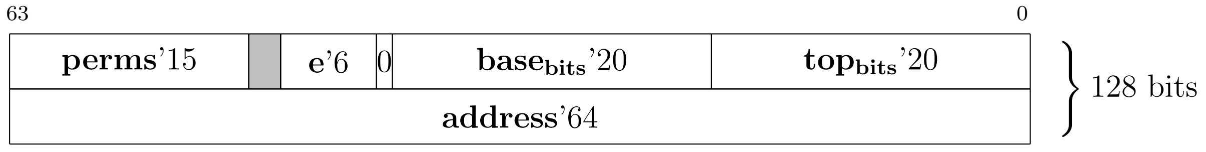
Note that the correction by $\pm\mathbf{S}$ on $address_{align}$ corresponds to a ± 1 correction in the first bit above $\mathbf{address_{bits}}$.

4.4.3 CHERI's specific implementation

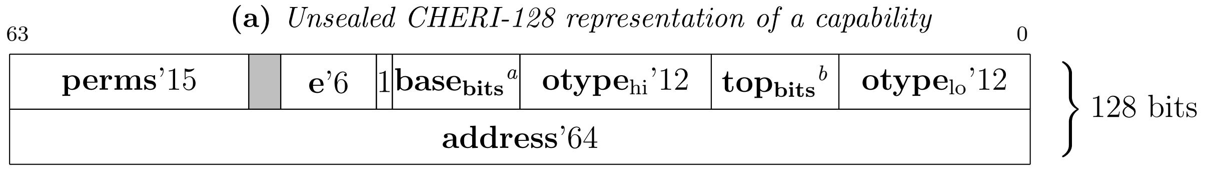
A representation of the CHERI-128 format is shown on **Figure 4.8**. The **address** is a full-sized value where as just some of the bits of the **base** and **top** addresses are stored (20-bits each of $\mathbf{base_{bits}}$ and $\mathbf{top_{bits}}$). 6 bits are allocated to the exponent **e**. In comparison with the 256 bits implementation of CHERI, the number of permission **perms** bits has been reduced to 15. The **otype** field is assumed to be zero in unsealed capabilities, and only exists in sealed capabilities. It reuses bits from the $\mathbf{base_{bits}}$ and $\mathbf{top_{bits}}$ which now have implied zeroes in their least significant bits,

[5] As an alternative way of representing this, one can think of $(\mathbf{edge_{bits}} \ll \mathbf{e})$ as the size of the **hi-region**. By construction, the distance between $align_{low}$ and $\mathbf{base_{edge}}$ is define as $(\mathbf{edge_{bits}} \ll \mathbf{e})$. The **low-region** size is therefore equal to $\mathbf{S} - (\mathbf{edge_{bits}} \ll \mathbf{e})$. Additionally, we know that the two **sub-region**s have a cumulated size of **S**, therefore **hi-region** has a size of $(\mathbf{edge_{bits}} \ll \mathbf{e})$. This can be seen easily on **Figure 4.7**.

forcing stricter alignment requirements for sealed capabilities. The details of these requirements have been described in the CHERI architecture document[95].



(a) *Unsealed CHERI-128 representation of a capability*



(b) *Sealed CHERI-128 representation of a capability*

[a]$\mathbf{base_{bits}}$[19:12], implied $\mathbf{base_{bits}}[11{:}0] = 0$
[b]$\mathbf{top_{bits}}$[19:12], implied $\mathbf{top_{bits}}[11{:}0] = 0$

Figure 4.8: *A 128-bit CHERI capability's memory representation for the newly proposed compression scheme*

The CHERI processor guarantees capabilities with a representable region providing a buffer above and below the object **base** of at least 4KiB (2^{12} bytes). To keep the implementation simple, we compute $\mathbf{edge_{bits}}$ as a simple function of $\mathbf{base_{bits}}$ (leaving $\mathbf{edge_{bits}}$ independent of **e**):

$$\mathbf{edge_{bits}} = \mathbf{base_{bits}} - 2^{12}$$

Note that the buffer available below the object grows with the exponent **e** (specifically, it doubles with each increment of **e**).

With this simple implementation of $\mathbf{edge_{bits}}$ we can perform the relevant arithmetic on only our small sized values, that is on 20-bit values. If we want to further minimize the resource usage for the arithmetic to meet more demanding critical paths requirements, we can impose some more restrictions on $\mathbf{edge_{bits}}$ as previously explained:

$$\mathbf{edge_{bits}} = \mathbf{base_{bits}} - (\mathbf{base_{bits}} \bmod 2^{12}) - 2^{12}$$

This operation is trivial in hardware:

$$\mathbf{edge_{bits}}[19:12] = \mathbf{base_{bits}}[19:12] - 1$$
$$\mathbf{edge_{bits}}[11:0] = 0$$

This version of $\mathbf{edge_{bits}}$ has the property that it has 12 zeroes in its least significant bits, effectively guaranteeing a buffer below the object of at least 4KiB (for an object whose **base** ends in 12 zeroes), up to one byte fewer than 8KiB (for objects whose **base** ends in 12 ones). These sizes scale up with **e** but cannot scale down. With this $\mathbf{edge_{bits}}$, the arithmetic operations on small-sized values reduces to 8(20 − 12)-bit arithmetic. It is of course possible to push this technique further with stronger alignment requirements on $\mathbf{edge_{bits}}$ to further reduce the number of bits required for the implemented arithmetic.

Deriving e

To derive an exponent **e** for a region of size *length*, the CHERI processor tries to maximize precision while also guaranteeing buffers that allow the pointer to move out of bounds. Specifically, for the format presented in **Figure 4.8**, CHERI guarantees that there is at least a 4KiB gap to the representable region's edges both above and below the object, by ensuring:

$$(length) \leqslant (2^{20} - 2^{13}) \cdot 2^{\mathbf{e}} \qquad \Leftrightarrow \qquad \frac{length}{2^{20} - 2^{13}} \leqslant 2^{\mathbf{e}} \tag{4.3}$$

Such a division is a non-trivial operation in hardware. Ideally, we want to divide by a power of two so that a simple shift operation can be used. For this, we will try to artificially inflate *length* to always account for the extra 8KiB of out-of-bounds buffer.

Let us consider a worst case length, i.e. a length 8KiB less than the maximum size precisely representable in 20 bits. To maximize precision, we want to keep **e** small. We want to find the smallest inflation factor $(1 + 2^x)$ (where x is a small negative number) that can be used to artificially grow the length to account for the out-of-bounds buffers, guaranteeing $2^{20} < (2^{20} - 2^{13}) \cdot (1 + 2^x)$.

For $x = -7$, the representable space available for buffers would be 64 bytes short of 8KiB: $(2^{20} - 2^{13}) \cdot (1 + 2^{-7}) - (2^{20} - 2^{13}) = 8128 < 8192$. Therefore, we find the smallest usable inflation factor to be $(1 + 2^{-6})$, giving a representable space available for buffers just 128 bytes smaller than 16KiB: $(2^{20} - 2^{13}) \cdot (1 + 2^{-6}) - (2^{20} - 2^{13}) = 16256 < 16384$.

Given that $2^{20} < (2^{20} - 2^{13}) \cdot (1 + 2^{-6})$ and that it is safe for **e** to increase ($\mathbf{e} \leqslant \mathbf{e}'$), we can elaborate on **Equation** (4.3) as follows:

$$\frac{(length) \cdot (1 + 2^{-6})}{(2^{20} - 2^{13}) \cdot (1 + 2^{-6})} \leqslant \frac{(length) \cdot (1 + 2^{-6})}{2^{20}} \leqslant 2^{\mathbf{e}'}$$

Therefore we can compute the $\mathbf{e}'$ we are interested in as follows:

$$\mathbf{e}' = \left\lceil log_2 \left(\frac{(length) \cdot (1 + 2^{-6})}{2^{20}} \right) \right\rceil$$

We are interested only in $\mathbf{e}' \geqslant 0$; in hardware this reduces to some simple arithmetic and a log_2 that turns into a function to determine the index of the most-significant bit set (idxMSBSet):

$$\mathbf{e}' = \text{idxMSBSet}((length + (length \gg 6)) \gg 19)$$
$$\text{where } (length + (length \gg 6)) \text{ is a 65-bit result}$$

Note that if speed of computation of **e** is preferred over precision of the allocation, a simpler expression for **e** that still trivially guarantees the presence of a 4KiB buffer above and below the object could be to double the length:

$$\mathbf{e}'' = \text{idxMSBSet}(length \gg 18)$$

CHERI uses $\mathbf{e} = \mathbf{e}'$. Every allocation of size *length* aligned on a $2^{\mathbf{e}}$ boundary has precise bounds. **e** is encoded in the 6-bit exponent field seen on **Figure 4.8**.

Note that if we are willing to sacrifice certain values of the exponent, it is possible to use fewer bits. For example, only using multiple-of-4 exponents would allow us to imply the bottom two bits of **e** that could then be represented on 4 bits.

An alternative way of encoding **e** is presented in **Appendix A**. It trades alignment requirement on the top and base addresses for exponent bits. This encoding is based on the recently presented “posits” numbers by John Gustafson.

Decompressing the object bounds

As we have seen in the previous section, we can derive a full bound address from a pointer and the bound bits saved when creating the capability. The reasoning exposed earlier with the notion of sub-regions was useful to understand the working principals of the mechanism. It can be simplified using the corrections c_b and c_t (defined in **Table 4.2**) as in the following:

$$\begin{aligned}
base_{dec}[63:20+\mathbf{e}] &= \mathbf{address}[63:20+\mathbf{e}] + c_b \\
base_{dec}[19+\mathbf{e}:\mathbf{e}] &= \mathbf{base_{bits}} \\
base_{dec}[\mathbf{e}-1:0] &= 0
\end{aligned}$$

$$\begin{aligned}
top_{dec}[64:20+\mathbf{e}] &= \mathbf{address}[63:20+\mathbf{e}] + c_t \\
top_{dec}[19+\mathbf{e}:\mathbf{e}] &= \mathbf{top_{bits}} \\
top_{dec}[\mathbf{e}-1:0] &= 0
\end{aligned}$$

Note that for sealed capabilities, $\mathbf{base_{bits}}[11:0] = 0$ and $\mathbf{top_{bits}}[11:0] = 0$ (due to those bits being allocated to the **otype**).

top_{dec} is a 65-bit quantity in order to allow the upper bound to be larger than the address space. This is used on reset to allow the default data capability to address all of the virtual address space as top_{dec} must be one byte above the highest accessible address. In this special case, $\mathbf{e} \geqslant 45$.

$\mathbf{address_{bits}} < \mathbf{edge_{bits}}$	$\mathbf{base_{bits}} < \mathbf{edge_{bits}}$	c_b
$\mathbf{address_{bits}} < \mathbf{edge_{bits}}$	$\mathbf{top_{bits}} < \mathbf{edge_{bits}}$	c_t
0	0	0
0	1	+1
1	0	−1
1	1	0

Table 4.2: *Calculating c_b and c_t*

Representable region checking

When being modified, the **address** of a capability needs to still point within the capability’s representable region, otherwise, the capability tag needs to be cleared.

On pointer modification, we verify that the result of the addition of an increment i to the pointer will not take it beyond the limits of the representable region. This is done only using i and the original pointer, avoiding the requirement to wait for the computation of the new pointer to happen before the test, hence guaranteeing the possibility of an efficient implementation of the mechanism.

There are two main components to the test. The *inLimits* test checks that the reduced-sized counterpart of i does not take the pointer past $\mathbf{edge_{bits}}$. The *inRange* test checks the magnitude of i with respect to the representable region's size $\mathbf{S}$.

The *inRange* test succeeds if the absolute value of i is less than $\mathbf{S}$, the size of the representable region:

$$\mathit{inRange} = -\mathbf{S} < i < \mathbf{S}$$

This reduces to a test that all the bits of i_{top} ($i[63 : E + 20]$) are the same (i.e. all zero or all one).

The *inLimits* test requires $\mathbf{edge_{bits}}$, $\mathbf{address_{bits}}$, i_{mid} ($i[E + 19 : E]$), and the sign of i to ensure that neither of the limits of the representable region have been crossed:

$$\mathit{inLimits} = \begin{cases} i_{mid} < (\mathbf{edge_{bits}} - \mathbf{address_{bits}} - 1)\,, & \text{if } i \geqslant 0 \\ i_{mid} \geqslant (\mathbf{edge_{bits}} - \mathbf{address_{bits}}) \wedge \mathbf{edge_{bits}} \neq \mathbf{address_{bits}}, & \text{if } i < 0 \end{cases}$$

Note that with a restricted 4KiB-aligned $\mathbf{edge_{bits}}$ (i.e. with 12 zeroes in the least significant bits), the ($\mathbf{edge_{bits}} - \mathbf{address_{bits}}$) operation does not require a full 20-bit operator, but can be implemented using only an 8-bit operator, as the alignment of $\mathbf{edge_{bits}}$ guarantee that no carry can be generated from the lower 12 bits of the operation.

We conservatively subtract one from the representable limit when we are incrementing upwards to account for any carry that may propagate up from the lower bits of the full pointer addition.

The final fast *representable* check composes the previous tests and also ensures that for $\mathbf{e} \geqslant 44$, any increment is representable :

$$\mathit{representable} = (\mathit{inRange} \wedge \mathit{inLimits}) \vee (\mathbf{e} \geqslant 44)$$

4.4.4 Working example of a compressed capability

We will now consider an object with a given base and top. We will create the CHERI-128 compressed representation of a capability to this object. We will then extract the original base and top addresses from the CHERI-128 compressed representation. This will be explained for an example that showcases the mechanism when there is a loss of precision. Note that a formal proof that "*extracted_base*" $\leqslant$ "*initial_base*" and "*extracted_top*" $\geqslant$ "*initial_top*" has been conducted in HOL4 as a collaboration with Anthony Fox.

In **Figure 4.9**, we have an example object that spans the memory from address `0x0010000000200000` to address `0x0010000001000FFF` (that is, first byte outside

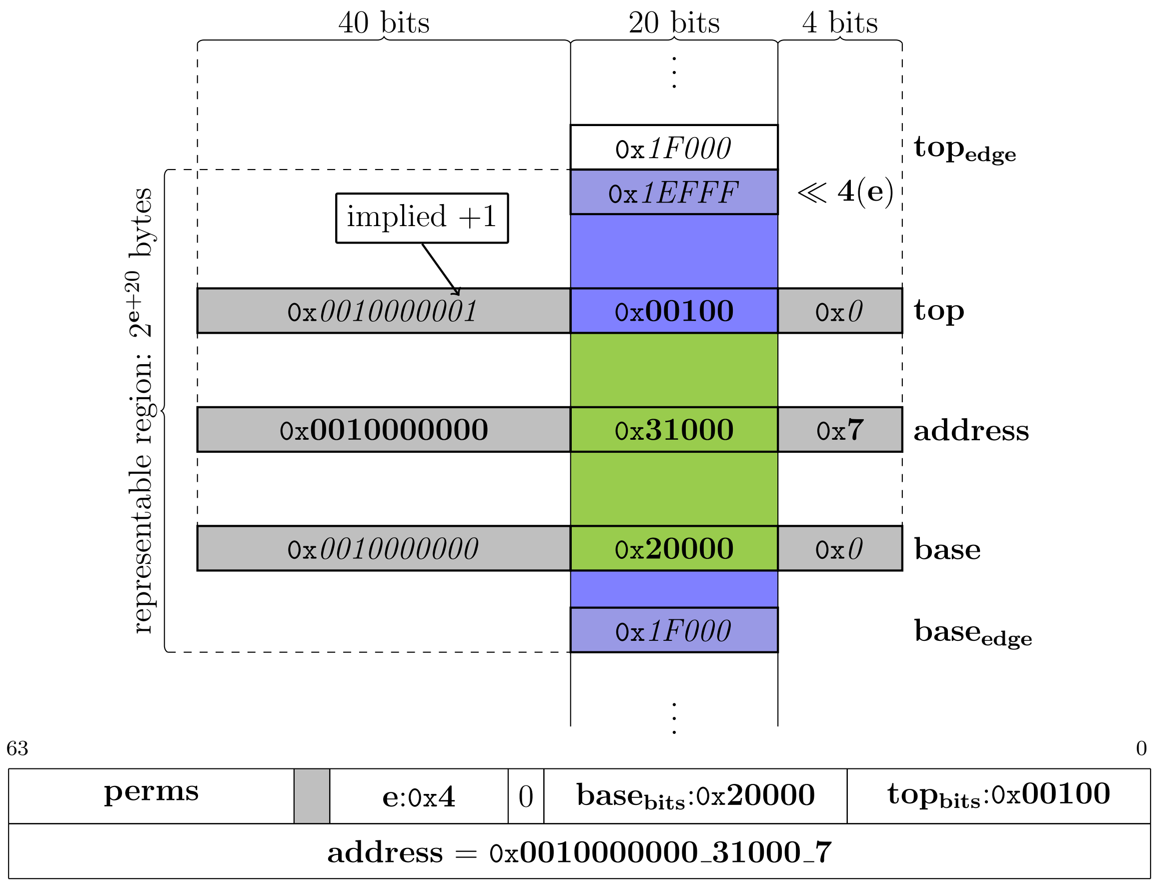


Figure 4.9: *Example capability with **base** = 0x0010000000200000, **top** = 0x0010000001001000 and **address** = 0x0010000000310007*

the object is at address 0x0010000001001000, and the object size is 0xE01000). We want to create a capability to that object, that is, we want to create a capability to an object with **base** = 0x0010000000200000 and **top** = 0x0010000001001000 with length *objlen* = **top** − **base** = 0xE01000).

Let us first infer the **e**, $\mathbf{base_{bits}}$ and $\mathbf{top_{bits}}$ fields in our encoding for a capability representing such an object.

Characterizing the capability - 1 of 2

Finding $\mathbf{e}$

$\mathbf{e}$ is a function of *objlen* = `0xE01000` only.

$$\begin{aligned}\mathbf{e} &= \text{idxMSBSet}\left((\texttt{0xE01000} + \texttt{0xE01000} \gg 6) \gg 19\right)\\ &= \text{idxMSBSet}\left(\texttt{0x1C}\right) = \text{idxMSBSet}\left(\texttt{0b00011100}\right)\\ &= \texttt{4}\end{aligned}$$

Finding $\mathbf{base_{bits}}$ *and* $\mathbf{top_{bits}}$

With $\mathbf{e}$, we can select the appropriate 20 bits from the **base** and **top** address of the object. In this case, bits 4 to 23.

$$\begin{aligned}\mathbf{base_{bits}} &= \mathbf{base}[19+\mathbf{e}:\mathbf{e}]\\ &= \texttt{0x0010000000200000}[23:4]\\ &= \texttt{0x20000}\end{aligned}$$

$$\begin{aligned}\mathbf{top_{bits}} &= \mathbf{top}[19+\mathbf{e}:\mathbf{e}]\\ &= \texttt{0x0010000001001000}[23:4]\\ &= \texttt{0x00100}\end{aligned}$$

These three fields are encoded in the capability and are shown in bold on **Figure 4.9**.

We further specify the capability to the object by inferring $\mathbf{edge_{bits}}$ and $\mathbf{S}$. These fields are not stored in the capability but are computed in a straight forward manner from already available information.

Characterising the capability - 2 of 2

To be able to check for the validity of pointer manipulations, we need $\mathbf{edge_{bits}}$ and $\mathbf{S}$ to characterise the representable region of the capability.

Finding $\mathbf{edge_{bits}}$

$$\begin{aligned}\mathbf{edge_{bits}} &= \mathbf{base_{bits}} - 2^{12}\\ &= \texttt{0x20000} - \texttt{0x1000}\\ &= \texttt{0x1F000}\end{aligned}$$

Finding $\mathbf{S}$

$$\begin{aligned}\mathbf{S} &= 2^{\text{length}(\mathbf{base_{bits}})+\mathbf{e}}\\ &= 2^{20+4}\\ &= \texttt{0x1000000}\end{aligned}$$

With $\mathbf{edge_{bits}}$ = `0x1F000` and $\mathbf{S}$ = `0x1000000`, we can qualify the representable region for the given capability: it spans the memory from $\mathbf{base_{edge}}$ = `0x00100000001F0000` to $\mathbf{top_{edge}}$ = `0x00100000011F0000` (excluded).

As already explained, capabilities are stored compressed in the register file. One could consider a microarchitecture that does some decompression and extends the capability registers with a few bits of state to avoid recomputing $\mathbf{edge_{bits}}$ and $\mathbf{S}$ on every use, and simply generate them when first loading the capability in the register. This was judged unnecessary in our implementation, but would not significantly change anything about the presented mechanism.

Now that we have gathered this information about the capability, we can give an example of how to manipulate the pointer. We first study the case of an address that stays within the object itself, as is it is the case on **Figure 4.9**. Let us consider an initial **address** pointing at the base of the object (`0x0010000000200000`), and move it at an offset of $1114119\text{bytes} = \texttt{0x110007}$ inside the object.

Manipulating the address - 1 of 2

*Move the **address** within representable bounds*

We increment the **address** by $i = \texttt{0x110007}$ (+1114119 bytes). We have $i \geqslant 0$:

$$\begin{aligned}
\mathit{inRange} &= \text{allSame}\,(i[63:\mathbf{e}+20]) \\
&= \text{allSame}\,(\texttt{0x110007}[63:24]) \\
&= \text{allSame}\,(\texttt{0x0}) \\
&= \texttt{true} \\
\mathit{inLimits} &= i[\mathbf{e}+19:\mathbf{e}] < (\mathbf{edge_{bits}} - \mathbf{address_{bits}} - 1) \\
&= \texttt{0x110007}[23:4] < (\texttt{0x1F000} - \texttt{0x20000} - 1) \\
&= \texttt{0x11000} < \texttt{0xFEFFF} \\
&= \texttt{true} \\
\mathit{representable} &= (\mathit{inRange} \wedge \mathit{inLimits}) \vee (\mathbf{e} \geqslant 44) \\
&= (\texttt{true} \wedge \texttt{true}) \vee (4 \geqslant 44) \\
&= \texttt{true}
\end{aligned}$$

The capability is still representable. Its address has been incremented and is now:

$$\begin{aligned}
\mathbf{address} &= \texttt{0x0010000000200000 + 0x110007} \\
&= \texttt{0x0010000000310007} \\
\mathbf{address_{bits}} &= \texttt{0x31000}
\end{aligned}$$

Note that it is possible to find an increment that would bring the address outside of the object but still within representable bounds.

We continue to modify our capability, incrementing it by `0xFFFFFFFFFF100000` (-15728640 bytes), which should take the address out of the representable region from below.

Manipulating the address - 2 of 2

*Move the **address** out of representable bounds*

We increment the **address** by i = `0xFFFFFFFFFF100000` (-15728640 bytes). We have $i < 0$:

$$\begin{aligned} \mathit{inRange} &= \text{allSame}\,(i[63 : \mathbf{e} + 20]) \\ &= \text{allSame}\,(\texttt{0xFFFFFFFFFF100000}[63 : 24]) \\ &= \text{allSame}\,(\texttt{0xFFFFFFFFFF}) \\ &= \texttt{true} \\ \mathit{inLimits} &= i[\mathbf{e} + 19 : \mathbf{e}] \geqslant \\ &\quad (\mathbf{edge_{bits}} - \mathbf{address_{bits}}) \wedge \mathbf{edge_{bits}} \neq \mathbf{address_{bits}} \\ &= \texttt{0x10000} \geqslant (\texttt{0x1F000} - \texttt{0x31000}) \wedge \texttt{0x1F000} \neq \texttt{0x31000} \\ &= \texttt{0x10000} \geqslant \texttt{0xEE000} \wedge \texttt{0x1F000} \neq \texttt{0x31000} \\ &= \texttt{false} \wedge \texttt{true} \\ &= \texttt{false} \\ \mathit{representable} &= (\mathit{inRange} \wedge \mathit{inLimits}) \vee (\mathbf{e} \geqslant 44) \\ &= (\texttt{true} \wedge \texttt{false}) \vee (4 \geqslant 44) \\ &= \texttt{false} \end{aligned}$$

When performing the full pointer increment on 64 bits, we get:

$$\begin{aligned} \mathbf{address} &= \texttt{0x0010000000310007 + 0xFFFFFFFFFF100000} \\ &= \texttt{0x0010000000310007 - 0xF00000} \\ &= \texttt{0x000FFFFFFF410007} \end{aligned}$$

We had seen that $\mathbf{base_{edge}}$ = `0x00100000001F0000`. We observe that:

$$\texttt{0x000FFFFFFF410007} < \texttt{0x00100000001F0000} \implies \mathbf{address} < \mathbf{base_{edge}}$$

The new **address** falls down under the $\mathbf{base_{edge}}$ of the representable region. Hence, this attempted pointer modification will yield an invalid capability / throw an exception[6].

Let us now try to dereference the capability with the address `0x0x0010000000310007`, that is inside the object bounds.

[6]The choice of invalidating a capability or throwing an exception makes little difference for the CHERI model, as all that is required is that the unexpected event is captured somehow. The exception approach gives the advantage that one could precisely identify the cause of a failure whereas clearing the tag of a capability would only be detected when later attempting to dereference / use the capability. However, most micro architectures will be greatly simplified by reducing the sources of potential exceptions. In that sense, we tend to adopt the clearing of the tag approach.

Dereferencing the address (decompressing object bounds) - 1 of 2

*Dereference the **address** within object bounds*

Let us consider our capability with an in bounds **address** = 0x0010000000310007.

$$\begin{aligned}\mathbf{address}_{\mathbf{bits}} < \mathbf{edge}_{\mathbf{bits}} &= \texttt{0x31000} < \texttt{0x1F000}\\ &= \texttt{false}\\ \mathbf{base}_{\mathbf{bits}} < \mathbf{edge}_{\mathbf{bits}} &= \texttt{0x20000} < \texttt{0x1F000}\\ &= \texttt{false}\\ \implies c_b &= \texttt{0}\end{aligned}$$

We find $base_{dec}$:

$$\begin{aligned}base_{dec}[63:24] &= \mathbf{address}[63:24] + c_b\\ &= \texttt{0x0010000000}\\ base_{dec}[23:4] &= \mathbf{base}_{\mathbf{bits}}\\ &= \texttt{0x20000}\\ base_{dec}[\mathbf{e}-1:0] &= \texttt{0}\end{aligned}$$

With an analogous reasoning, we find top_{dec}. We have:

$$base_{dec} = \texttt{0x0010000000200000} \text{ and } top_{dec} = \texttt{0x0010000001001000}$$

The object was indeed accurately represented by the capability as **base** = $base_{dec}$ and **top** = top_{dec}. Furthermore, we see that the **address** can be dereferenced safely:

$$\begin{aligned}&\texttt{0x0010000000200000} \leqslant \texttt{0x0010000000310007} < \texttt{0x0010000001001000}\\ \implies &base_{dec} \leqslant \mathbf{address} < top_{dec}\end{aligned}$$

We now conclude this example by finally looking at an attempt at dereferencing a capability with an address 0x00100000010FF000 out of the object bounds.

Dereferencing the address (decompressing object bounds) - 2 of 2

*Dereference the **address** out of object bounds*

Let us consider our capability with an out-of-bounds **address** = `0x00100000010FF000`.

$$\begin{aligned} \mathbf{address_{bits}} < \mathbf{edge_{bits}} &= \texttt{0x0FF00} < \texttt{0x1F000} \\ &= \texttt{true} \\ \mathbf{base_{bits}} < \mathbf{edge_{bits}} &= \texttt{0x20000} < \texttt{0x1F000} \\ &= \texttt{false} \\ \implies c_b &= \texttt{-1} \end{aligned}$$

We find $base_{dec}$:

$$\begin{aligned} base_{dec}[63:24] &= \mathbf{address}[63:24] + c_b \\ &= \texttt{0x0010000001} - \texttt{1} \\ &= \texttt{0x0010000000} \\ base_{dec}[23:4] &= \mathbf{base_{bits}} \\ &= \texttt{0x20000} \\ base_{dec}[\mathbf{e}-1:0] &= \texttt{0} \end{aligned}$$

With an analogous reasoning, we find top_{dec}. We have:

$$base_{dec} = \texttt{0x0010000000200000} \text{ and } top_{dec} = \texttt{0x0010000001001000}$$

The $base_{dec}$ and top_{dec} addresses are still correctly recovered with the out-of-bounds **address** = `0x00100000010FF000`. In this case however, the **address** cannot be dereferenced safely:

$$\begin{aligned} &\texttt{0x0010000000200000} \leqslant \texttt{0x00100000010FF000} \not< \texttt{0x0010000001001000} \\ &\implies base_{dec} \leqslant \mathbf{address} \not< top_{dec} \end{aligned}$$

4.4.5 Further discussion of the new scheme

This CHERI-128 example uses 20 bits for the top and base fields. It may still be possible to cover most use cases with even smaller fields if more free bits became required. Reducing the size of the $\mathbf{base_{bits}}$ and $\mathbf{top_{bits}}$ fields is also a trade off on the size of the **otype** field.

As previously mentioned, some bits are to be gained in the encoding of the exponent (see **Section 4.4.3**, "**Deriving e**").

This scheme also still has some redundant encodings, specifically when representing well aligned capabilities, as several exponents could theoretically be used with bounds field with similar values shifted appropriately. A further optimization here would be to require $\mathbf{top_{bits}}$ to always be at least 2^{20} greater than $\mathbf{base_{bits}}$ (that is **top** to be at least $2^{\mathbf{e}+20}$ greater than **base**) for exponents greater than zero. This would remove the possibility for object of a given size to be represented with a given **e** as one of the big objects, and at the same time as one of the small objects with the next **e**. Fixing this requirement allows us to imply the most significant bit of $\mathbf{top_{bits}}$ rather to store it, also gaining a bit for further compression.

Finally, note that this compression technique can be used for different pointer sizes, e.g. for future very large address spaces, one may consider deriving a CHERI capability using this compression technique to provide 256-bit capabilities capable of holding a 128-bit virtual address.

Equally, further research conducted within the CHERI project, specifically by Hongyan Xia, aims at exploiting this compression technique within the context of Internet Of Things, by using a 32-bit virtual address space and a 64-bit capability. A use cases coverage study for reduced size $\mathbf{base_{bits}}$ and $\mathbf{top_{bits}}$ fields would be particularly relevant in this context, as well as the optimisation presented in **Appendix A**.

4.5 Evaluation of the compression scheme

This section presents an evaluation of the compression mechanism that has been presented in this chapter. The evaluation was performed using several Stratix IV FPGA evaluation boards to support a CHERI processor with the same cache hierarchy configuration as in **Section 3.3.2**, and the FreeBSD OS (and CheriBSD).

4.5.1 128-bit CHERI memory impact

Figure 4.10 shows the raw memory overheads that were presented in **Section 3.3.4** for 256-bit CHERI next to new numbers for 128-bit CHERI, for benchmarks running pure capability code and normalised to the pure MIPS case. As already explained, these are raw overheads of data memory bytes fetched by the processor's pipeline, which do not directly translate to actual memory traffic at the end of a cache hierarchy. These numbers still show that the 128-bit CHERI format can lower the memory costs present when using the 256-bit CHERI format.

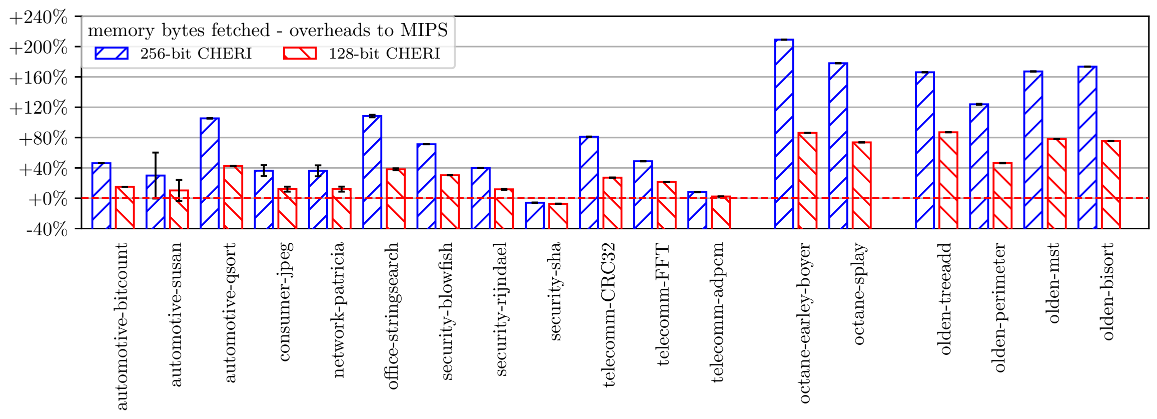


Figure 4.10: *Data memory bytes fetched overhead for 128-bit and 256-bit CHERI with respect to MIPS*

Similarly to what was observed in **Section 3.3.4**, these effects propagate up the cache hierarchy as observed on **Figures 4.11** to **4.13**. We see that, unsurprisingly, 128 bits per pointer is much more acceptable than 256 bits for pointer-heavy workloads.

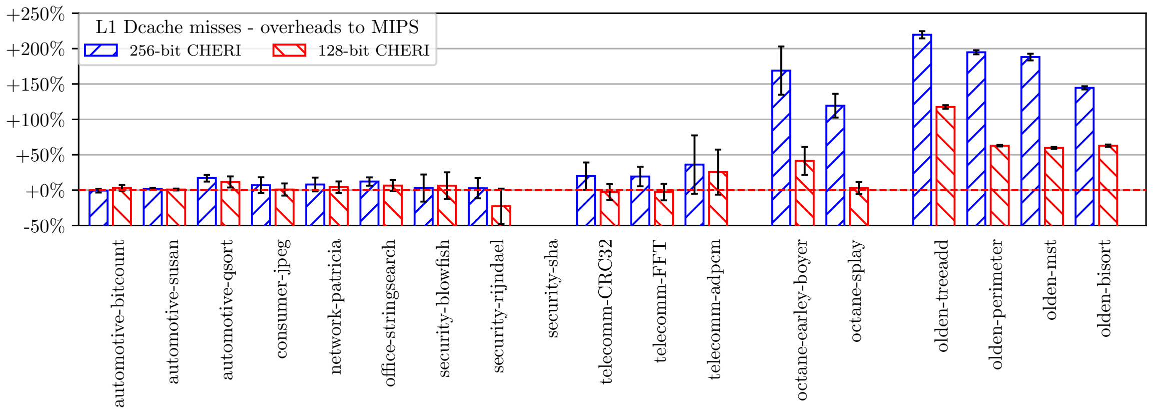


Figure 4.11: *L1 data-cache miss overhead for 128-bit and 256-bit CHERI with respect to MIPS*

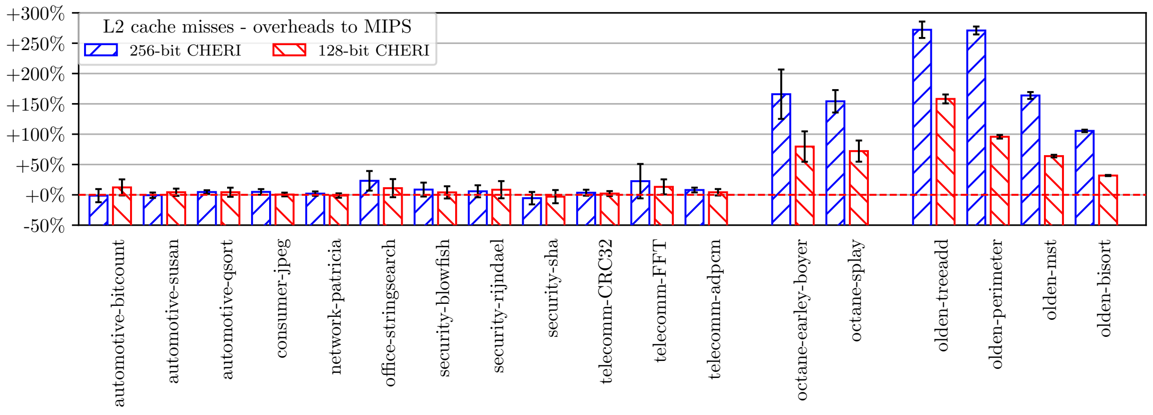


Figure 4.12: *L2 cache miss overhead for 128-bit and 256-bit CHERI with respect to MIPS*

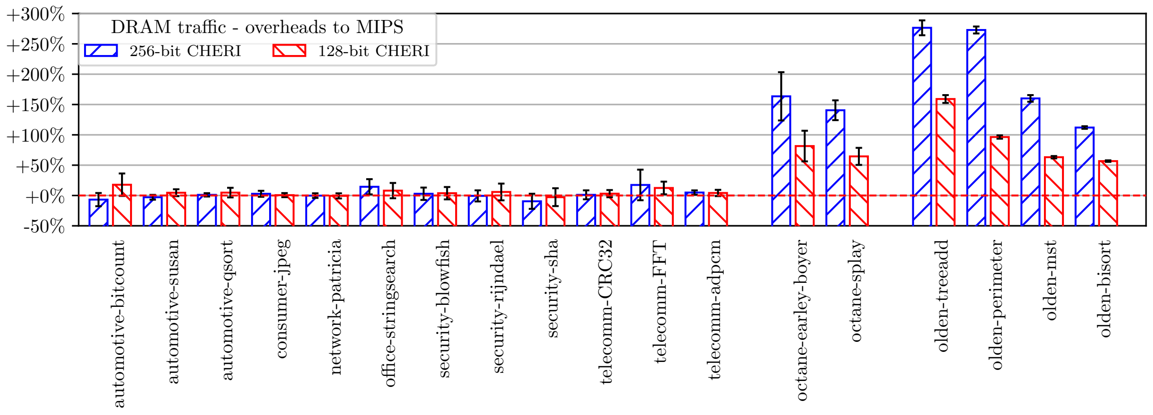


Figure 4.13: *DRAM traffic overhead for 128-bit and 256-bit CHERI with respect to MIPS*

4.5.2 128-bit CHERI performance impact

Once again, the results presented should be considered in light of the dynamic instruction counts presented in **Figure 4.14**. The state of maturity of the CHERI compiler toolchain is such that some negative overheads can be seen, as already explained in **Section 3.3.4**.

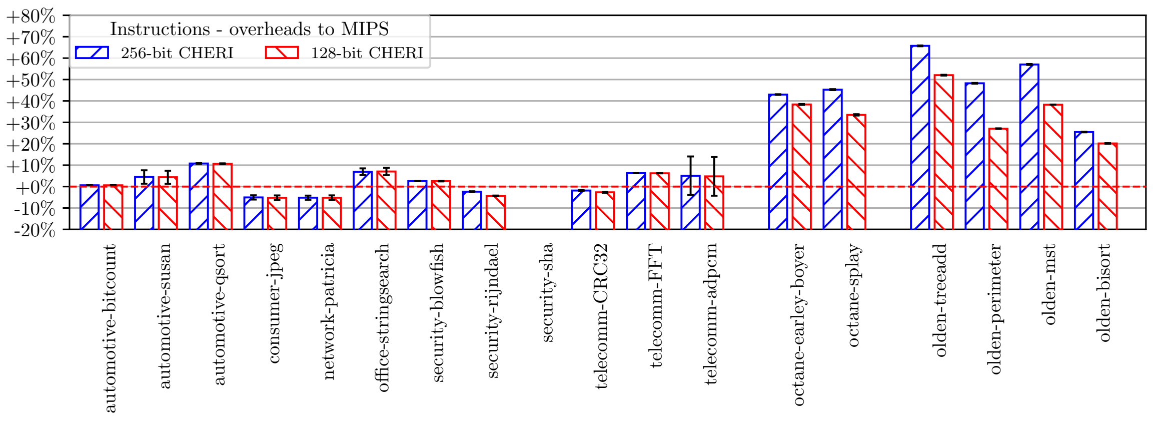


Figure 4.14: *Dynamic instructions count overhead for 128-bit and 256-bit CHERI with respect to MIPS*

Even though 128-bit CHERI mitigates the costs coming from the size of pointers in 256-bit CHERI, there remains optimisations required to bring CHERI to its full potential. If we look at **Figure 4.15**, we see that runtime overheads remain relatively high.

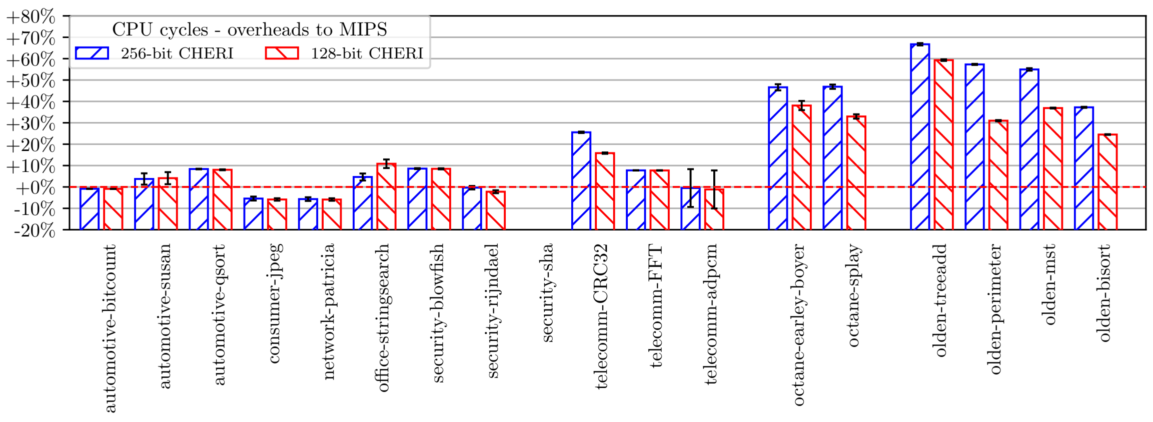


Figure 4.15: *Runtime overhead for 128-bit and 256-bit CHERI with respect to MIPS*

Figure 4.14 explains these increased run times: a strong correlation exists between **Figures 4.14** and **4.15** for pointer heavy applications. The increased working set size will put significant pressure on the TLB mechanism which is software managed in the MIPS architecture. Each TLB miss triggers an exception which will transition the processor into kernel mode, setup/fetch the appropriate TLB entry before coming back to the useful benchmark code. This process consumes a significant number of instructions as observed in **Figure 4.16**.

For benchmarks that are less pointer intensive, this effect is no longer dominating the run time overheads. The reason some runtime overheads do still exist are the additional instructions generated for capability code. These will likely be reduced to some extent as the CHERI compiler matures. This, however, is a cost shared by both 256-bit CHERI and 128-bit CHERI. 128-bit CHERI almost always consistently performs better than 256-bit CHERI as expected from the memory related gains, but in rare cases, when the runtime is no longer memory dominated, 128-bit CHERI actually performs slightly worse. There are two reasons why this can happen. Having

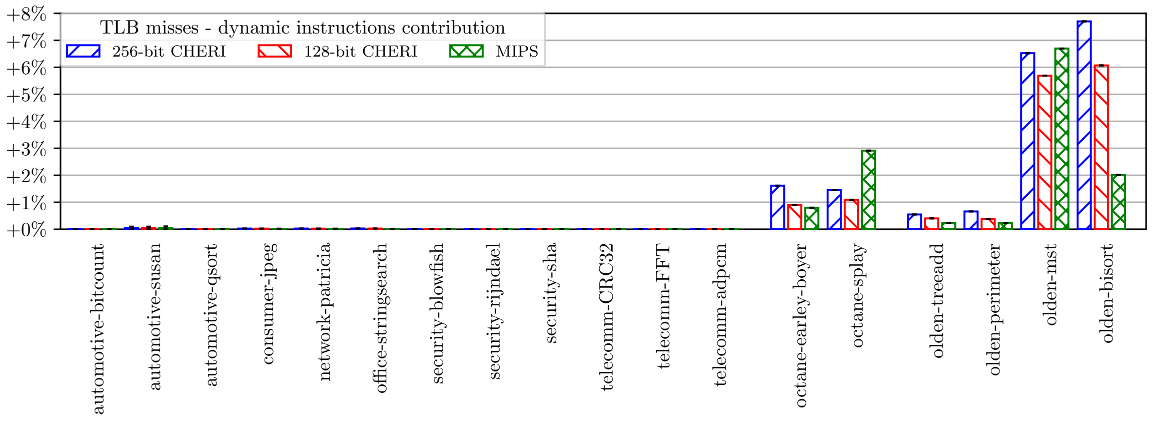


Figure 4.16: *TLB miss contributions to dynamic instruction count, effectively the share of dynamic instructions spent in a TLB handler*

access to 256-bit registers rather than 128-bit registers will allow `memcpy` to be faster on 256-bit CHERI. Finally, as explained in this chapter, the compression scheme presented optimises for the most frequent pointer operations, that is loading/storing pointers and manipulating them (`CLoad`, `CStore`, `CIncOffset`...). To achieve that, the capability is stored compressed in the register file. If it is ever necessary to interact with a value that is not compressed (that is when specifying 64 bit values when creating a capability with `CSetBounds` or getting 64 bit values when querying the base or length with `CGetBase` or `CGetLen`), additional delay is required. This means that the later class of instruction may require an additional cycle for its results to be used. Again, ideally, the compiler can be taught to take these pipeline details into account when generating code.

Chapter 5

Efficient tagged memory

The CHERI capability model requires capabilities to be unforgeable. In particular, capabilities stored in memory are not to be altered in undesired ways. To enforce this property, CHERI uses hidden validity tag bits that are logically associated with each memory location able to store a capability. Any write to such a location will clear the associated capability validity-tag bit, unless it is an explicit write of a valid capability. We achieve this using a tagged memory. In this chapter, I explore how to efficiently implement such a tagged memory.

A 256-bit CHERI capability system has a $\frac{1}{256}$ (less than 0.4%) overhead in terms of tag storage costs, as one capability validity-tag is required for every 256 bits of data. In other words, for every 1GiB of memory that can contain capabilities, an extra 4MiB of memory is required to store the tags. For a 128-bit CHERI capability system, this overhead grows up to $\frac{1}{128}$ (around 0.8%), requiring 8MiB of tags for every 1GiB of data.

CHERI separates the validity-tag bits from the data memory in a non-software-accessible shadowspace. This approach does not require any software support, making compatibility less of an issue, and also prevents software from (intentionally or unintentionally) overwriting tag bits.

The storing and retrieving of these tag bits does not require software support in normal operation but comes with some associated hardware costs.

5.1 Hardware support for tagged memory

Tagged memory is used in many computer systems, giving the ability to keep track of pointers [18, 26, 59], enabling implementation of unforgeable capability tokens [11, 14, 23, 57, 100], tracking programmable information-flow [84, 87, 88], and even implementing general-purpose watch-point systems to support both debugging and software-defined security invariants [42, 103].

Some approaches use a *single-bit tag* (SBT) while others use a *multi-bit tag* (MBT). An MBT allows for rich metadata to qualify the tagged word. An SBT only provides binary information. Implementing an MBT architecture comes with a large fixed cost compared to an SBT architecture in terms of extra storage requirement for the tag shadowspace. CHERI capability integrity can be enforced with a single tag bit, therefore the SBT approach is the chosen option[1]. There exist several ways to implement an SBT shadowspace, each coming with its own set of advantages and disadvantages. The following sections review what these options are.

5.1.1 Storing tags in a wider DRAM

The ideal solution to the tag storage problem would be to have the tag bit stored in the DRAM itself, as an extra bit embedded in each memory location able to hold a capability. For 256-bit CHERI capability systems, the memory would actually return 257 bits for each 256 bits of data requested (129 for a 128-bit CHERI capability system). The advantage of this solution is that the tag is always present at no extra latency cost. The obvious disadvantage is that this requires a custom memory that actually holds this extra tag bit (in the same fashion as error correction bits for ECC memories). This is, in the long term, what we would like to see. However, industrial adoption of the CHERI mechanism will be facilitated if a more pragmatic solution exists, involving only off-the-shelf memory.

5.1.2 Storing tags in a dedicated memory

An alternative to the first solution is to have a memory dedicated to tag storage. With this approach, we preserve the advantages of having the tags always present

[1] Note that CHERI can conceptually be seen as a 64-bit pointer scheme enhanced with rich metadata. CHERI's multi-bit metadata is stored in data memory, next to the pointer itself, which makes CHERI an *embedded metadata* architecture, like [14, 57]. None of these architectures use an MBT shadowspace (all the useful metadata can be embedded in the words, therefore there is no need to put pressure on the shadowspace). On the other hand, *external metadata* architectures store the metadata in a shadowspace. Some [42, 84, 88] simply require an SBT shadowspace for simple binary information ([42] actually uses a 2-bit tag scheme, one for read watchpoints and one for write watchpoints. They can be viewed as two SBT applications as neither actually require extra metadata.), and others [87] require an MBT shadowspace. "External metadata MBT shadowspace" architectures can be implemented as "embedded metadata SBT shadowspace" architectures provided that software is aware that the metadata is embedded in the data memory. "External metadata MBT shadowspace" architectures do not require software to be aware of the extra metadata and are therefore somewhat more compatible, but to the cost of a big fixed overhead. Embedded metadata architectures have the added benefit that the cost of metadata scales with use, and can be implemented with an SBT shadowspace which make the fixed cost component much smaller. CHERI is an "embedded metadata SBT shadowspace" architecture.

and available with no extra main memory requests or latency cost. Using a dedicated off chip memory would require an all new memory channel with its own dedicated pins on the chip, which is costly and undesirable. As previously explained, in order to cover 1GiB of data memory, 4MiB (8MiB for 128-bit CHERI) of tags are required, with this value growing linearly with the size of the data memory. Even a 4MiB tag memory has a significant cost for an on-chip memory (this size is comparable to the typical 8MiB of an L3 cache on modern Intel Core i7 processors). For these reasons, I explore a third option.

5.1.3 Storing tags in a subset of the DRAM

CHERI uses the DRAM itself to hold the tags in a dedicated segment (individual tags are not attached to the corresponding 256-bits / 128-bits chunks of data). This approach has the advantage of not relying on any custom memory or extra memory module, which means it is compatible with off-the-shelf memories.

In this approach, both the data and the tags which are stored in different parts of the DRAM need to be fetched. This leads to extra memory traffic and extra latency. Later sections show how to mitigate these costs.

5.2 Tag-awareness in the memory sub-system

CHERI implements tagged memory with a “merged-cache” approach, in the same fashion as [18, 31, 81, 82]. The alternative “split-cache” approach is taken by [42, 78, 79, 80, 84, 87, 88, 89].

The merged-cache approach stores extra bits of tag along with the data for each cache line in all caches in the cache hierarchy.

The split-cache approach uses a dedicated L1 cache for tags and usually has the tags competing for space with the data in the lower shared levels of the cache hierarchy. This approach generally also comes with complications in the pipeline due to an extra memory access stage to lookup the tag cache separately from the data access, and may require a separate register file.

Additionally, the split-cache approach has the problem that it needs to guarantee consistency between tags and data which exist in different cache lines in the cache hierarchy; it is necessary for a data write and the associated tag write to appear atomic. The FlexiTaint paper [88] points out that having individual tag lines mapping to several data lines leads to a new kind of false sharing effect. Because a cache line containing tags covers a great number of data cache lines, enforcing atomicity of writes and keeping caches coherent becomes more challenging. A 64-byte line containing single-bit tags for 32-bit words needs to be updated on writes to any of the 32 64-byte lines of data covered, meaning that coherence for these tag lines happens on a kilobyte/multi-kilobytes granularity.

Note that split-cache designs also tend to tag virtual addresses, leading to per address space tag tables, increasing the complexity of the design. On the other hand,

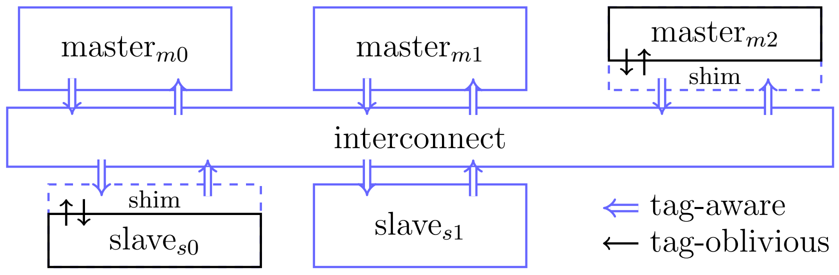


Figure 5.1: *A hybrid system with tag-awareness shims. Tag-oblivious components are in blue, tag-aware components are in black*

tagging physical addresses only requires a single tag table to be maintained, for a fixed amount of dedicated memory.

There exist hybrid approaches such as the one presented by Arora et al. [8], which use merged-cache style L1 caches but still suffer from the split-cache technique consistency challenges which they use in their shared L2 cache.

CHERI uses physical tags and a merged-cache approach where a tag propagates atomically with the data throughout the cache hierarchy, avoiding all the issues of the split-cache approach, with the cost of widening the data path to fit the tag. In the context of CHERI, this cost is negligible as the data path is already widened to the size of a capability word, and the one extra bit of tag on top of this cost is minimal. Most of CHERI's modules in the memory subsystem manipulate data chunks and their associated tags as a single atomic unit. For future discussions, we refer to such modules as being "tag-aware". Modules of the memory subsystem that manipulate data memory requests without the notion of associated tags are referred to as "tag-oblivious".

Conventional off-the-shelf memories can be considered tag-oblivious. We identify the need for a "shim" module acting as a bridge between a tag-aware world and a tag-oblivious world as drawn on **Figure 5.1**.

A shim to turn a tag-oblivious DRAM into a tag-aware DRAM will consist of a state machine turning a single memory request into a pair of memory requests for both the data and its tag, and recombining both the responses into a single response to the initial request. A share of the original DRAM module must be dedicated to tag storage, effectively reducing the amount of usable memory. The amount of memory dedicated to tags is negligible when compared to the total size of the DRAM, as previously discussed. The obvious significant cost is that of the extra DRAM accesses.

5.2.1 Tag-awareness in the CHERI hardware

We see on **Figure 5.2** a schema showing how a CHERI system is capable of using off-the-shelf memory to store tags. Here the shim module is situated after the last level cache of the CHERI core. It makes assumptions as to what slaves are present in the systems and ends up being somewhat specialised to the particular SoC configuration assumed by CHERI. A more realistic system would move its tag-awareness shim module(s) closer to the actual tag-oblivious slave(s), making more of the overall system tag-aware. The fundamental features of the tag-awareness shim are nonetheless still the same. This chapter will present an optimisation to the existing approach in the form of a more generic module, with fewer assumptions on the overall system, better suited for arbitrary memory slaves.

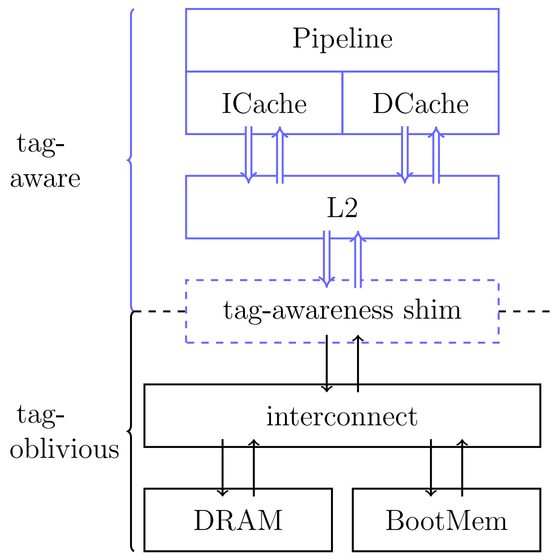


Figure 5.2: *Tag-awareness in CHERI's memory subsystem. The main core and caches are all tag-aware. The DRAM and peripherals are tag-oblivious*

The current CHERI implementation uses a tag-awareness shim that mitigates the memory traffic overheads by exploiting spatial and temporal locality on tags using a cache filled by standard DRAM requests to a segment of the DRAM module dedicated to tag storage. Note that CHERI has a boot memory module meant to be capable of storing capabilities. The segment reserved for tags at the top of the DRAM module currently covers both the data section of the DRAM and this boot memory[2].

Any memory request the shim receives is forwarded to DRAM, and a tag lookup is also initiated in the tag cache. On a cache hit, the shim waits for the data memory response to come back from the DRAM module, and simply pairs the tag bits with the data before forwarding the response to the master who issued the memory request. On a cache miss, the tags corresponding to the data targetted by the initial memory request need to be fetched. An additional memory request is generated by the shim, targetting the tag segment of DRAM. Once both memory responses (data and tags) are received by the shim, the tag-aware memory response can be crafted and forwarded back to the initial master[3].

[2]A more realistic approach would be to use a separate tag-awareness shim module for each tag-oblivious slave rather than sharing one like it is currently done, leading to tags for the boot memory being stored in the DRAM, forcing an interconnect topology with both the DRAM and boot memory module hidden behind the same shim.

[3]To protect the tag segment of DRAM, any write memory request targetting that region is dropped in the shim. Read requests to that same region simply behave normally (note that the physical region containing tags still needs to be mapped in a virtual address space explicitly for this to happen in user space). Ideally, such requests should not be generated by software, and it is expected that a device tree (or equivalent hardware discovery mechanism) would report a DRAM size not including the tag segment, masking the tag segment of DRAM.

One very powerful feature of this approach is its leveraging of a cacheability amplification factor naturally present in the CHERI system: single bit tags cover 256 (or 128) bits of memory. In other words, the 32KiB tag cache used in CHERI's shim module caches tags for 8MiB (or 4MiB) worth of data memory. This results in cache behaviour similar to large caches in terms of hit rates, but for modestly sized caches as a few tag bits cover a large number of data bits. Note however that this cacheability amplification factor is reduced by the 128-bit CHERI format presented in **Section 4.4**. This further motivates potential optimization work on the tag storage mechanism.

5.2.2 Potential for optimisation

The tag DRAM traffic overhead generated by CHERI's tag cache shown on **Figure 5.3** can be an obstacle to widespread adoption of CHERI. This graph shows the share of the DRAM traffic due to tags $\left(\frac{\text{traffic out of tag shim} - \text{traffic out of L2 cache}}{\text{traffic out of tag shim}}\right)$. Note that the very high overheads can be correlated to the results shown in **Chapter 4**, specifically **Figures 4.13** and **4.16**: benchmarks such as bisort with a large, pointer-heavy working set that won't fit in caches see more TLB misses and more DRAM traffic than the others, leading to a more drastic increase in tag DRAM traffic. Additionally, the use of 128-bit CHERI further increases tag density, leading to even higher tag-related overheads (this is visible for all benchmarks).

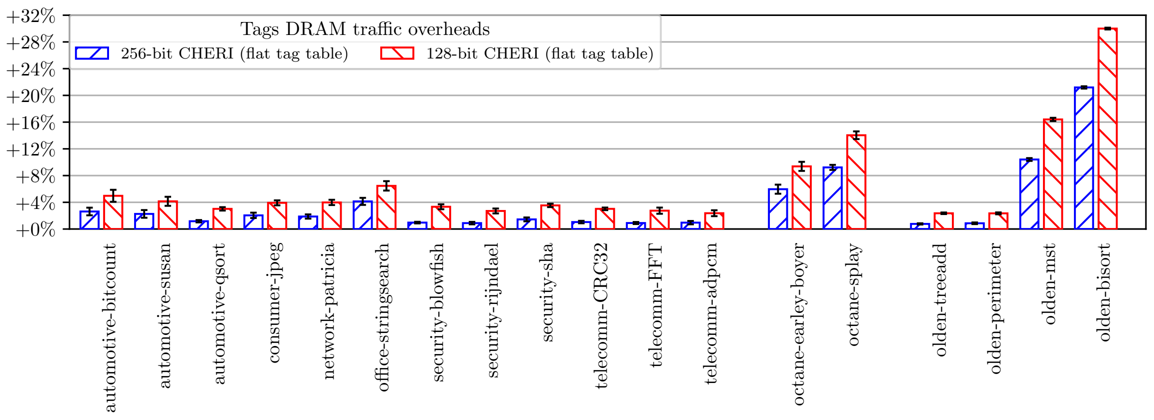


Figure 5.3: *Share of the DRAM traffic due to capability validity tags for 128-bit CHERI and 256-bit CHERI with 32KiB tag cache*

The FPGA prototype currently used for CHERI development is not really impacted by increases in the number of DRAM accesses, as the main processor's speed does not enable it to saturate the DRAM bandwidth. This is largely due to the relatively slow logic on FPGA vs. "hard" DRAM logic. Commercial implementations will however often be concerned about any increase in the DRAM traffic. Additional requests to a DRAM module can significantly impact the overall power consumption of the system. The extra bandwidth consumption itself will also be a concern, as SoCs are designed such that available memory bandwidth is fully utilised (e.g. by extra cores or embedded GPUs). This emphasizes the importance of tightly controlling the number of additional DRAM requests for the tags mechanism.

Legacy applications that do not use capabilities do not require tag information. However, extra memory requests fetching unset tag bits are still generated in the current CHERI setup.

The rest of this chapter focuses on mitigating these two aspects of the tag mechanism: reducing the extra memory costs for applications that don't use tags, and better caching of tags that are used.

5.3 Characterizing the existing tag cache

CHERI's tag awareness shim along with its tag cache is situated at the lowest level of the cache hierarchy, after the last level (L2) cache. It therefore sees the traffic that a last-level cache would see, that is memory accesses that missed the upper levels of the cache hierarchy, and on their way to the DRAM. As a reference, Jaleel [49] presents cache miss numbers for a three-level cache hierarchy with 32KiB instruction and data L1 caches, a 256KiB unified L2 cache and a 2MiB unified L3 cache running SPEC CPU2000 and SPEC CPU2006, and finds an average miss rate above 45% for the last level L3 cache across both benchmark suites. Such a miss rate for the tag awareness shim's tag cache would map to at least a 45% increase in memory accesses. Results will demonstrate that a tag cache's miss rate is much lower.

5.3.1 Dynamic tag cacheability study

A dynamic study of cacheability of tags can be performed by replaying a trace of DRAM accesses on a parametrizable simulated tag cache. Here we assume that every 64-bit word has an associated tag bit that must be fetched even if all tags are zero.

DRAM and TLB traces were generated by Robert Kovacsics, who instrumented two ARMv8 gem5 simulators. The first simulator configuration referred to as "small" has a single core with a 256KiB L2 last-level cache. The second "big" configuration has four cores and an 8MiB L3 last-level cache with prefetching enabled. Since an ARM architecture was used, the ports of the various benchmarks to CHERI cannot be trivially evaluated of this new setup. A limited number of benchmarks more readily available was run instead: Earley-Boyer of the Octane suite under Google V8 and FFMPEG. Earley-Boyer is a Javascript workload that embodies an unfavourable pointer distribution. FFMPEG is a media centric C program with fewer pointer accesses. Both benchmarks manipulate a data set of around 60MiB. In the big system, three instances of Earley-Boyer were run to increase cache stress, but only single instance of FFMPEG as it is multithreaded.

Given DRAM traces and the associated TLB traces, it is possible to approximate which requests are accessing pointers. This is achieved by matching bit patterns in the DRAM trace with those of mapped address ranges found in the TLB trace. False positives should be rare, as it is unlikely that non-pointer values will match a valid 64-bit address, especially when turning on address space layout randomization features. New "tag" traces are generated using this technique, providing a sequence of accesses to the tag cache for every 64-bit pointer detected, that can be used as input to a tag cache simulator.

With the help of Jonathan Woodruff, I developed a parametrizable python simulator to allow exploration of the possible tag cache configurations. By virtue of detecting 64-bit pointers in this study, we observe that this simulator will see a natural amplification factor of 64 (where 128-bit and 256-bit CHERI respectively see a 128 and 256 natural amplification factor). Where we usually see high miss rates in last level data caches, this tag cache using 64-byte lines and 8-ways performs well even for relatively small sizes, as seen on **Figure 5.4**. For example, for Earley-Boyer big, despite the fact that it covers only the same amount of data as the 8MiB L3 cache that it serves, a 128KiB tag cache manages to hit over 90% of the time, inflating DRAM accesses by less than 10% for both fills and writebacks.

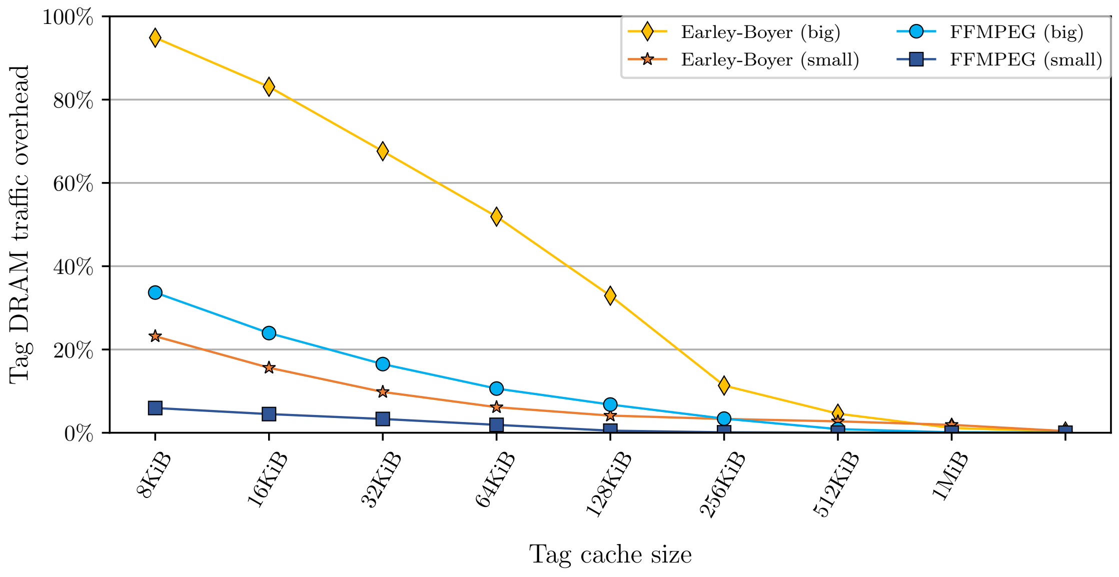


Figure 5.4: *Tag DRAM traffic overhead dropping as the tag cache size increases. A 128KiB tag cache hits over 90% of the time*

A 5% overhead can be achieved on the small system with a 32KiB tag cache covering 2MiB of data, that is 8 times as much as the 256KiB L2 last-level cache of the system. On the big system, 5% overhead is reached with a tag cache as small as 256KiB covering 16MiB, or only twice what is covered by the 8MiB last-level L3 cache of the system. Earley-Boyer and FFMPEG have 4.91% and 1.31% overheads respectively for the small system, and 3.72% and 0.50% for the big system.

The reason for this high tag-cache hit rate is spatial locality on a page scale. The share of tag accesses that are misses, spatial hits (that is on tags that have not previously been accessed in the cache, i.e. that have been brought in due to a miss on a nearby tag) and temporal hits (that is tags that have previously been accessed) are shown on **Figure 5.5**. These simulation results are for the Earley-Boyer big system with a 256KiB tag cache, 8-way associative, and for increasing tag-cache line sizes. We see that as the cache line size increases, the number of misses decreases. Bigger lines benefit spatial hits more than they harm temporal hits until lines of approximately 512 bytes or 4096 tags (that is covering 32KiB, or 8 4KiB pages of data). After that point, no more spatial locality seems to be harvested from larger lines, but the number of temporal hits still decreases, harming overall hit-rate.

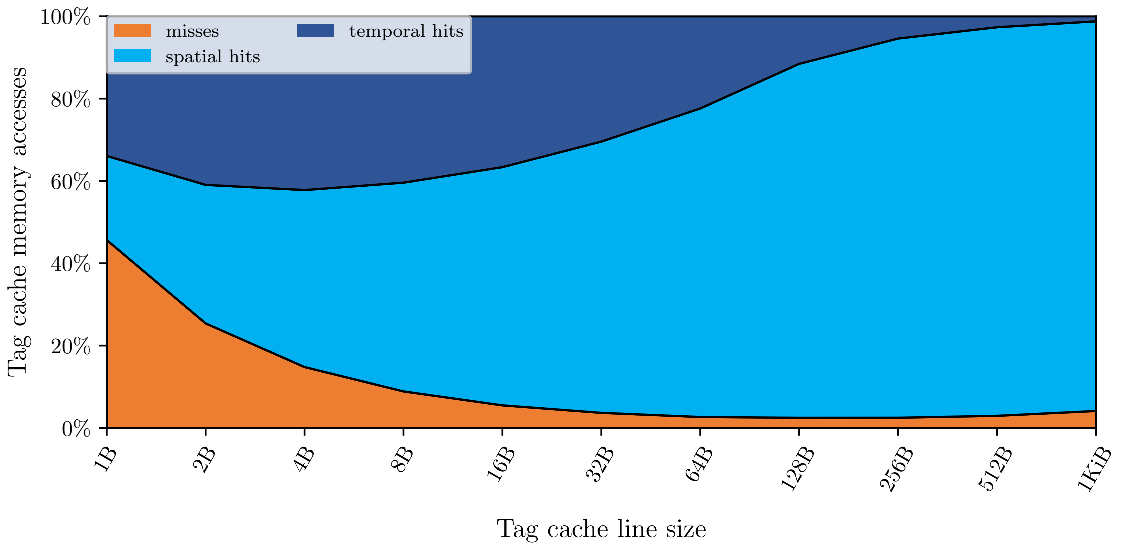


Figure 5.5: *Earley-Boyer(big) tag cache accesses vs tag cache line size. As the line size grows, the accesses are dominated by spatial hits*

We note that write requests on tags may leave the tags unchanged, and this particularly often for untagged data. This kind of write (identified in the context of standard cache accesses) is called "redundant store" by Molina et al. [61]. Redundant stores are particularly common for legacy applications carrying zeroed tags. A coarse line granularity will increase the probability of dirty lines building up in the tag cache even when no change in the actual content of the line has happened. The tag cache simulator is enhanced with a flag that enables the detection and elimination of redundant stores. **Figure 5.6** shows the improvement achieved by this optimization. For the two "big" workload, we see that this optimization can lead to almost 50% of tag DRAM traffic overhead reduction for a 4KiB cache. The gains are more modest for the "small" workloads.

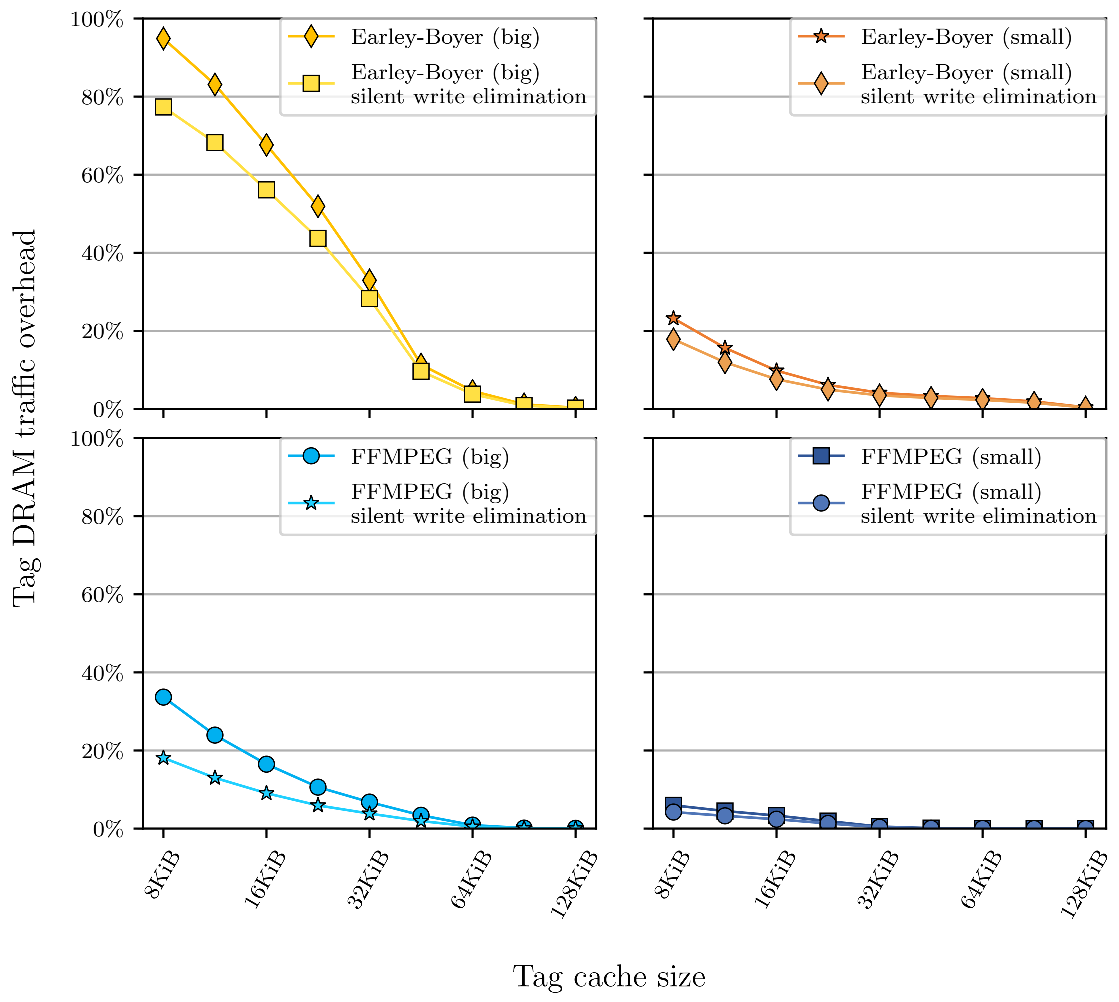


Figure 5.6: *Tag cache "redundant stores" elimination optimization. The optimization reduces up to ≈ 15% of the tag DRAM traffic overhead for a 4KiB cache in the "big" cases*

5.4 Caching pointer tags efficiently

In this section, we further reduce the overheads of tagged memory in CHERI using compression. We will first look at a static pointer density study in x86 applications, with the aim of finding a way to limit the cost of storing their associated tags, followed by a dynamic study of a simulated hierarchical tag cache leveraging the observations made by the static study.

5.4.1 Static pointer density in x86 applications

We look at a selection of x86 applications running under FreeBSD and identify how the pointers they use are distributed across the virtual address space. The intention is to consider the number of pointers found in these applications to determine the distribution of tags that would be required for each application when ported to CHERI (that is, all pointers are implemented using capabilities), to estimate how this would be seen by the tag storage mechanism.

This experiment was conducted under FreeBSD since there exists a FreeBSD port to CHERI (CheriBSD), and applications available on FreeBSD are more likely to be easily ported to CHERI.

A snapshot of an application's address space together with its memory mappings contains the information required to identify the memory locations that hold a pointer and those that do not. It is possible to keep track of the number of contiguous memory chunks holding pointers for a given chunk granularity. This information can help characterise the distribution of tags across application memory, and ultimately motivate optimisations to the tag storage mechanism.

Gathering snapshots

A "coredump" is a binary file in the ELF format, and contains different segments corresponding to the various memory mapped regions of the application at the time of taking the snapshot. Under FreeBSD, the `gcore` utility takes the processid of a desired running application and creates a coredump of that running process. FreeBSD's version of `gcore` also dumps additional metadata segments in the coredump alongside the process's data, in particular, some virtual memory mapping information.

It is possible to manually pause a long running application and take a snapshot with `gcore` for later analysis. For short lived applications, this is not a practical approach. Generating coredumps programmatically can be done with minor modifications to the sources of a program. For this, I added a call to `fork()`, immediately followed by a call to `abort()` when in the child process. This allows the parent process to continue its execution while a clone of the the process is terminated generating a coredump as a side effect. This second approach is particularly useful for benchmarks that run much faster on a modern x86 machine than on our CHERI prototype.

Thanks to these to methods, I gathered coredumps for the benchmarks presented in **Section 4.5**.

coredump-scanner

I developed "coredump-scanner", a tool that analyses the content of a coredump and extract pointer information. It is capable of understanding the virtual memory mapping information reported by FreeBSD and identifying values in the dumped memory that look like pointers based on this information. This gives a static estimate of an upper bound of the number of pointers used. As with the previously presented dynamic study, false positives should be rare, as it is unlikely that non-pointer values will match a valid 64-bit address. Additionally, coredump-scanner remembers how many blocks of memory (for different power-of-two sizes) held at least one pointer. This information gives an idea of how the pointers are clustered in the application under study. We collect this information for groups of size 8B that can hold a single pointer, to groups of size 64KiB that can hold 8192 pointers, with all power-of-two sizes in between. The following section explores the results of coredump-scanner.

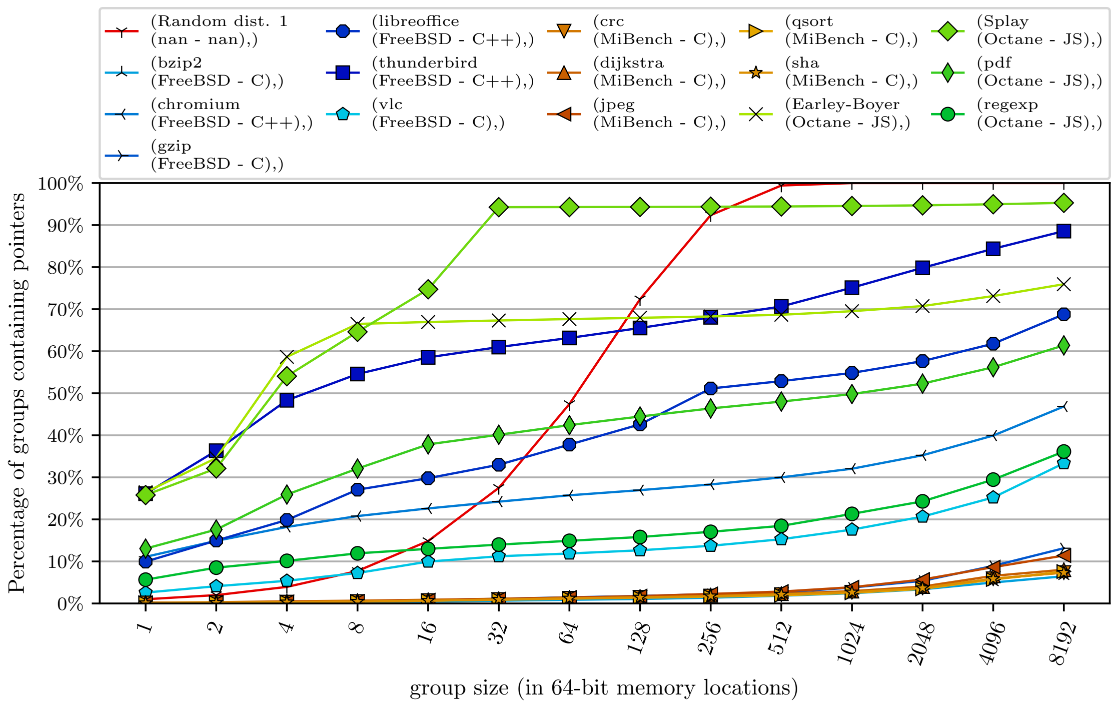


Figure 5.7: *Measured pointer densities vs. a 1% uniform random distribution. A 1% uniform random pointer distribution exhibits more groups containing pointers than even an application with a 30% pointer density for groups between* ≈ 256 *and* ≈ 512 *pointers (and groups of only* ≈ 64 *pointers for applications with around 10% pointers)*

5.4.2 Exploring compression of pointer tags

To assess compressibility of the tag working set of a program, we are interested in the distribution of pointers across memory, and in particular, the share of groups that contain at least one pointer for various group sizes. **Figure 5.7** reports, for a selection of applications, the share of memory groups that contain pointers for increasing group sizes, where the group size is reported in number of 64-bit memory locations, that is memory locations capable of holding a pointer (these would be 128-bit or 256-bit memory locations for CHERI). For many of the studied C-language applications, fewer than 10% of the memory contain pointers, even for large groups. For these applications, more than 90% of the tags could theoretically be eliminated from the tags working set, and the reach of a tag cache would be amplified by a factor of more than 10 above the natural amplification factor of 64 (128 or 256 for CHERI). For applications in higher-level languages, such as JavaScript and C++, there is a much higher concentration of pointers, with some approaching as much as 30% of all memory, and up to around 95% for very large groups.

Nevertheless, even in pointer-rich applications, pointers tend to cluster together, i.e., the grouped pointer densities are lower than for a uniform random distribution. **Figure 5.7** includes a synthetic uniform random distribution with 1% of the address space being pointers that surpasses all the application samples even for large groups. We see that for group sizes above 32, low pointer density applications all have fewer pointer groups than the 1% pointers uniform random distribution. For groups sizes of 256 and above, all applications have fewer pointer groups than the 1% pointers uniform random distribution. This result suggests that using a hierarchical structure rather than a flat tag table would improve cacheability by eliminating contiguous groups of tags from the tag working set.

A hierarchical approach - low pointer density

Let us study one specific low pointer density application. **Figure 5.8** shows an example of the results returned for the "Bitcount" mibench benchmark.

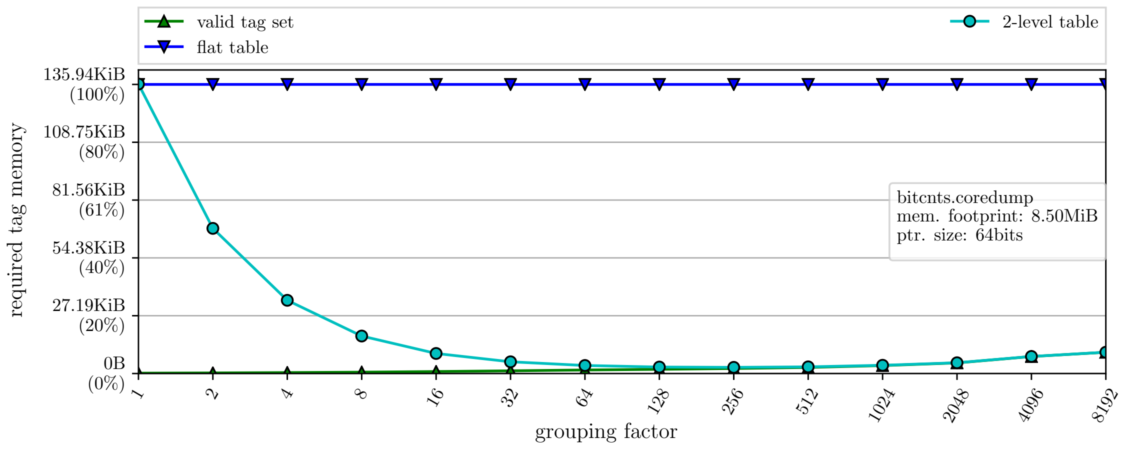


Figure 5.8: *Example result from coredump-scanner - "Bitcount", 64-bit pointers. As the grouping factor increases, the required space for the "2 lvl table" drops significantly. It is indistinguishable from the "valid tag set" curve for grouping factors greater than 64*

The x axis gives the various grouping factors that were sampled by coredump-scanner, that is the size of the memory groups considered. We use the term "grouping factor" to refer to how many bits at a given level of the hierarchy are grouped behind a bit at the level above, as presented in **Figure 5.9**. In this approach, a 0 root level bit means that there are no tags set in the group of memory locations hidden below it, and therefore that group does not need to be cached. A 1 root level bit means that there is at least one tag set in the group of memory locations below it, and therefore that group needs to be cached. This enables us to exploit the sparsity of the leaf level table to achieve a smaller tag cache footprint.

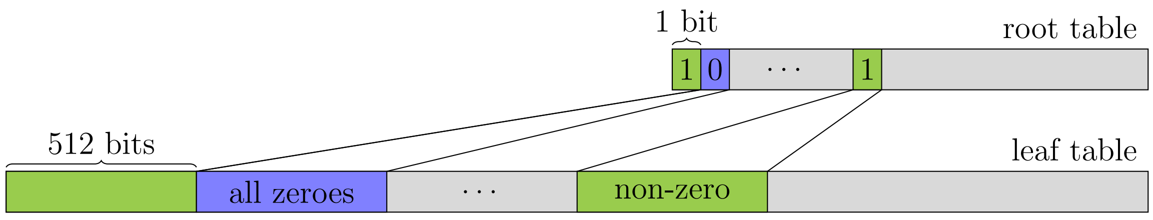


Figure 5.9: *Example of a hierarchical table structure with 2 levels and a grouping factor of 512*

Figure 5.8's y axis gives the size of the "required tag memory", i.e. the number of tag bits that would have to be cached to cover the whole working set of the application. This information is reported both as an absolute amount of memory and as a percentage of the all memory locations capable of holding a pointer.

Three curves are displayed, showing the various caching requirements necessary for different approaches. Upwards triangles mark the "valid tag set", that is the raw information of how many groups of a given size (the grouping factor) where at least a pointer was encountered by coredump-scanner (the same information that was reported on **Figure 5.7**). This curve corresponds to the ideal case where only valid tag bits would be cached, and no cache space would be wasted with cleared tags. We see that "Bitcount" uses very few pointers (close to 0% for a grouping factor of 1, and barely reaching 10% when pushing the grouping factor to 8192), suggesting very low cache space requirements when using suitable compression (a few kilobytes).

The downwards triangles curve is the "flat table" case, representative of the current CHERI approach to caching tags. All memory locations are tracked with this technique which does not use a grouping factor. 100% of all memory locations represents 135.94KiB worth of tags for 64-bit pointers for the 8.50MiB overall working set of "Bitcount". This is of course relatively far from the ideal cache footprint for tags seen for the "valid tag set" curve.

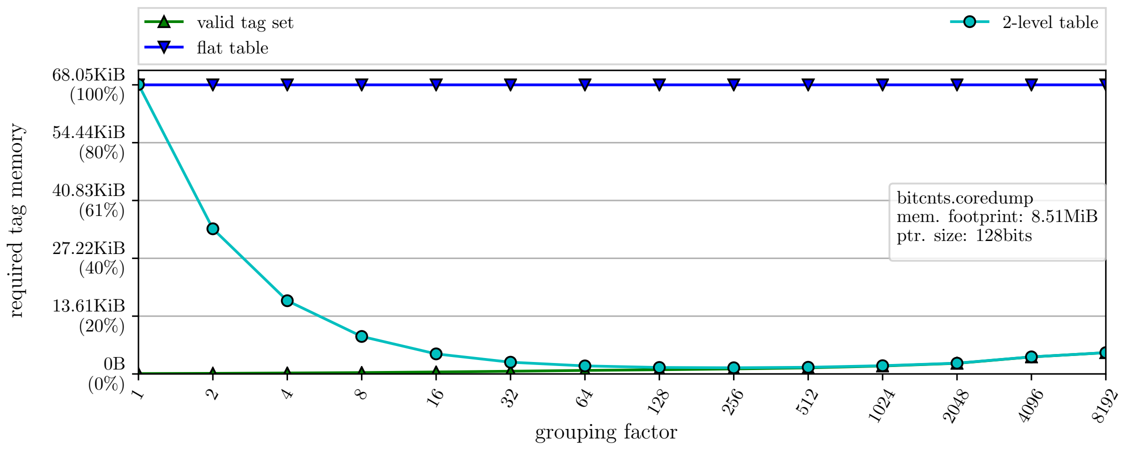
To approach the ideal curve, let us consider a 2-level hierarchical structure, where each bit at the root level masks a number of tag bits at the leaf level given by the grouping factor. This gives us the "2-level table"(based on the mechanism described in **Figure 5.9**) curve with circles.

The results from coredump-scanner can be used to estimate the cache footprint of such a hierarchical tag table by building the leaf table, and counting leaf groups containing pointers. The total tag working set is the entirety of the root table and the set of leaf groups containing pointers. Note that while enabling a reduced tag cache working set, the overall table structure consisting of the full flat table and the extra root level is fully allocated in DRAM in this technique.

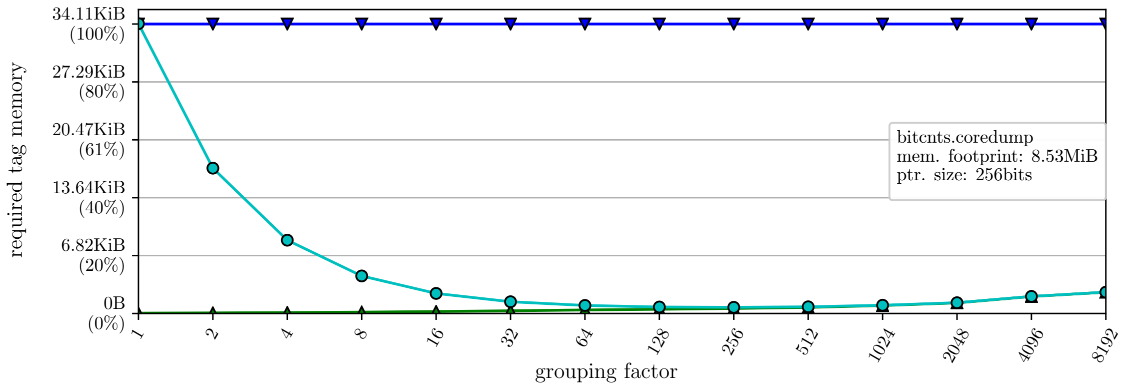
We can see in **Figure 5.8** that as the grouping factor increases, the required space for the "2-level table" curve drops significantly. It finally almost merges with the "valid tag set" curve for grouping factors greater than 64.

Figures 5.10a and **5.10b** give results for artificially grown pointer sizes (respectively 128 bits and 256 bits) to approximate the costs to consider when using CHERI capabilities. We see that the total memory footprint grows with the size of pointers. As expected, we also observe that the tag memory footprint is reduced, as fewer locations capable of holding pointers are now contained in the total memory footprint of the application.

One of the arguments made in **Chapter 4** is that in order for CHERI capabilities to be adopted in realistic industrial uses, we require them to go down from 256 bits to 128 bits. We see here the clear downside of this: doubling the amount of required capability validity tag bits. The newly described hierarchical approach greatly mitigates this problem. For the "Bitcount" benchmark, cache footprint for the 2 level table approach drops below a few kilobytes for grouping factors greater than 16, effectively making the 128-bit pointer case as attractive as the 256-bit



(a) *"Bitcount" pointer density, 128-bit pointers*



(b) *"Bitcount" pointer density, 256-bit pointers*

Figure 5.10: *Approximations of **Figure 5.8** for 128-bit and 256-bit pointers*

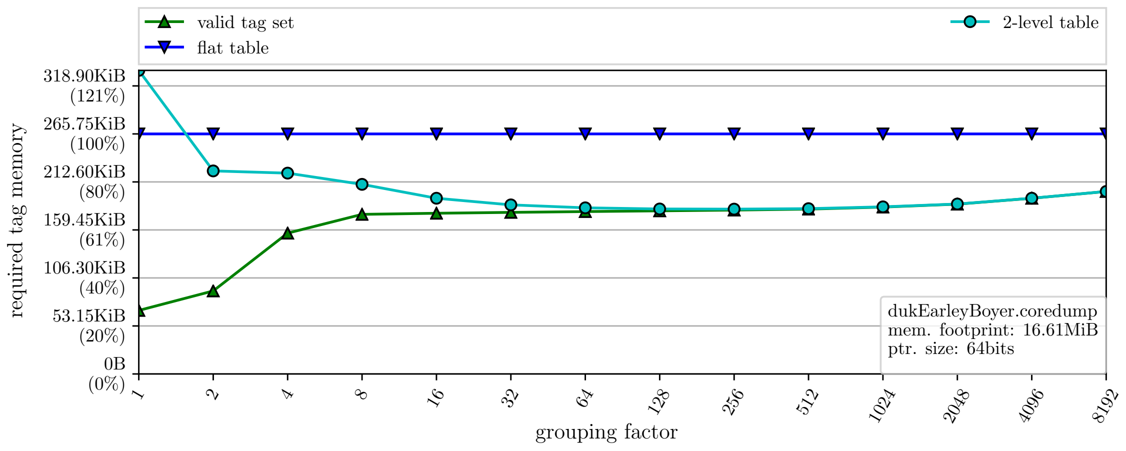
pointer case from a tag caching perspective.

The grouping factor in the hierarchical approach conceptually acts as an extra amplification factor on top of the natural amplification factor for the tag cache, i.e. for very low pointer density applications, root level bits will effectively have a negligible footprint in the tag cache and each cover the equivalent in data memory of the natural amplification factor multiplied by the grouping factor. This ideal combined amplification factor is only approached by low pointer density applications as leaf nodes will still need caching in the presence of pointers.

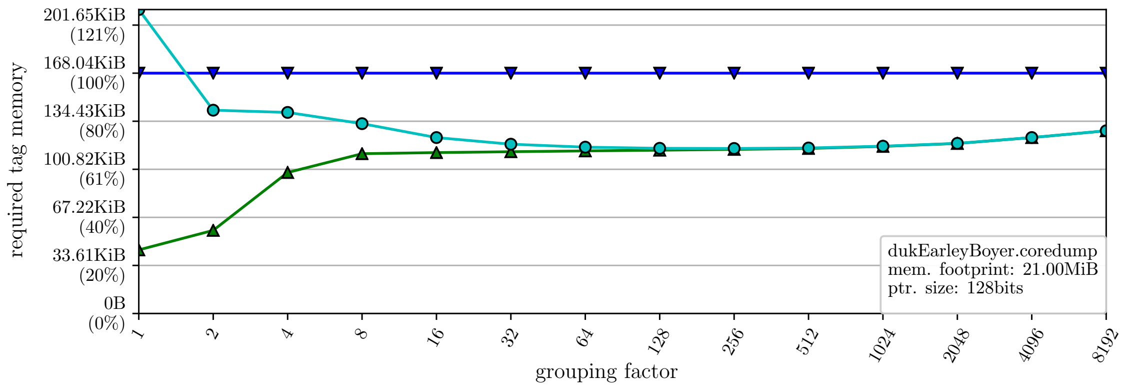
A hierarchical approach - high pointer density

The "Bitcount" benchmark is representative of applications that use very few pointers. On the other hand, "Earley-Boyer" is more representative of a pointer heavy application. **Figures 5.11a** to **5.11c** present the coredump-scanner results for the Earley-Boyer benchmark.

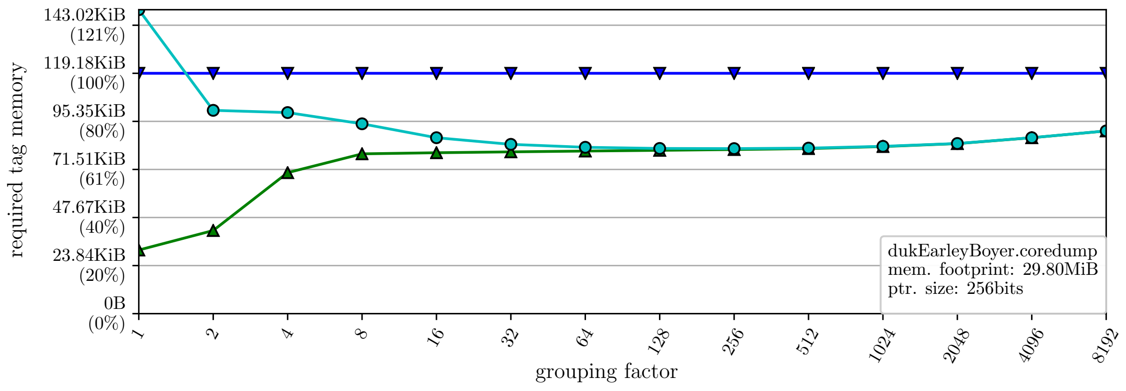
Using the 2-level table technique with a grouping factor of 128, we see that the 128-bit pointer case will have a tag cache footprint of ≈120KiB where "Bitcount" only required a few KiB. The 2-level table curve merges with the ideal curve for a grouping factor around 128. This graph show that for a pointer heavy application, a



(a) *"Earley-Boyer" pointer density, 64-bit pointers*



(b) *"Earley-Boyer" pointer density, 128-bit pointers*



(c) *"Earley-Boyer" pointer density, 256-bit pointers*

Figure 5.11: *"Earley-Boyer" pointer densities. For grouping factors greater than 64, the "2-lvl table" curve is indistinguishable from the "valid tag set" curve*

2-level table approach would still present some benefits over the flat table approach.

A Multi-level approach

Conceptually, better amplification factors in the tag cache can be achieved by hiding more bits behind one higher level bit. Using multiple levels in the hierarchical structure can help achieve that.

Figure 5.12 shows the required tag memory for a 3-level table for the "Bitcount" results with a 128-bit pointer size, with a variety of root and middle level grouping factors combinations. Configurations of grouping factors leading to a valley in the surface being drawn are the most desirable.

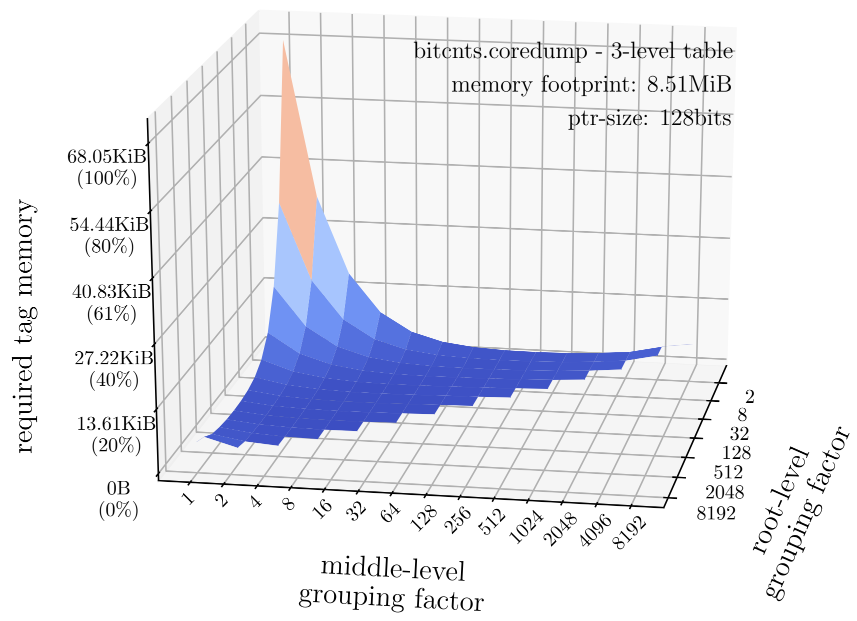


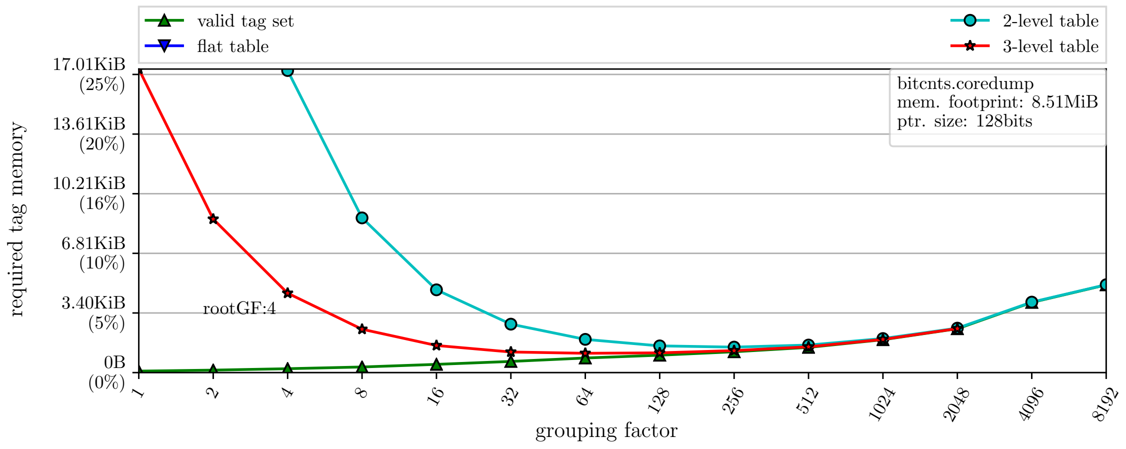
Figure 5.12: *A 3-level table for "Bitcount". Valleys in the surface are configuration of grouping factors leading to lower tag footprint*

We look at specific slices of the surface graph for specific root grouping factors in **Figures 5.13a** to **5.13c**. These figures show a new curve with stars, labelled "3-level table". The root grouping factor is written next to the starred curve. The middle level grouping factor, or final leaf granularity, is shown on the x axis.

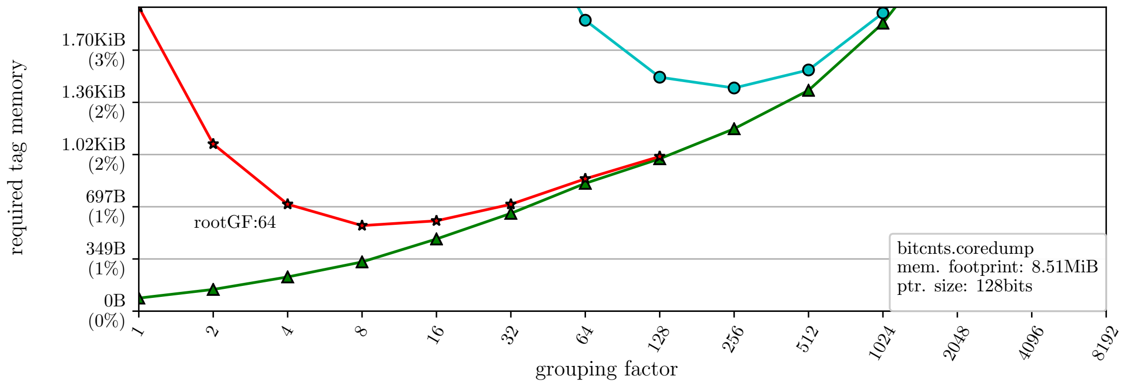
Note that there are fewer data points for the 3-level table as we consume some of the data returned by coredump-scanner to account for the root-level grouping factor on top of the middle level[4].

We can see in **Figures 5.13a** to **5.13c** that for low pointer density applications, the 3-level approach converges with the ideal case faster than the 2-level approach does. For higher pointer density applications, **Figures 5.14** and **5.15a** to **5.15c** do not show as much improvement as the root grouping factor increases. For the low pointer density case, the ratio between the optimal point on each curve is 76% with

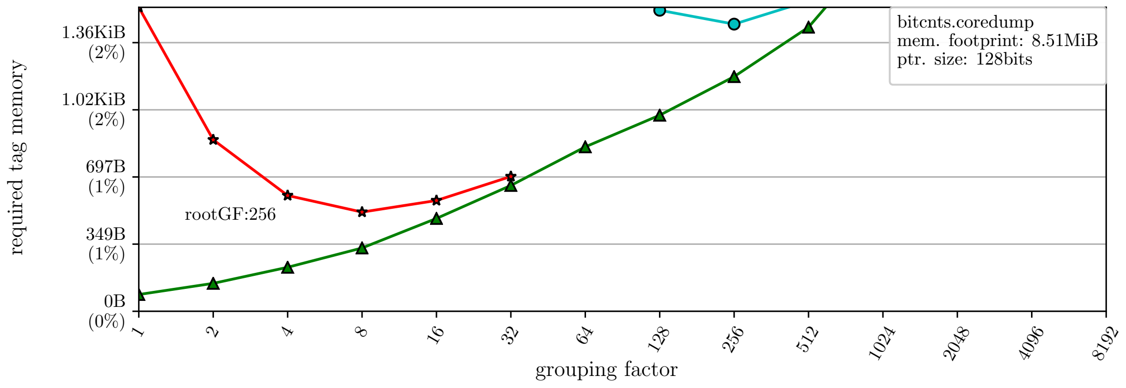
[4]For example, on **Figure 5.13a**, as the coarsest granularity sampled by coredump-scanner is for groups of 8192 memory locations, the coarsest data point that can be represented for a 3-level table with a root grouping factor of 4 is $\frac{8192}{4} = 2048$. Similarly, we get fewer data points for greater root grouping factors. This however is not a significant issue as we can see that the 3-level table curve converges rapidly with the valid tag set.



(a) *"Bitcount" adding a 3-level table, root grouping factor 4*



(b) *"Bitcount" adding a 3-level table, root grouping factor 64*



(c) *"Bitcount" adding a 3-level table, root grouping factor 256*

Figure 5.13: *The leaf granularity on the x axis is the root level grouping factor for the 2-level table, and the middle level grouping factor for the 3-level table. With a root grouping factor of 4 the "3-level table" converges with the "valid tag set" faster than the "2-level table" curve, and even faster for greater root grouping factors of 64 and 256*

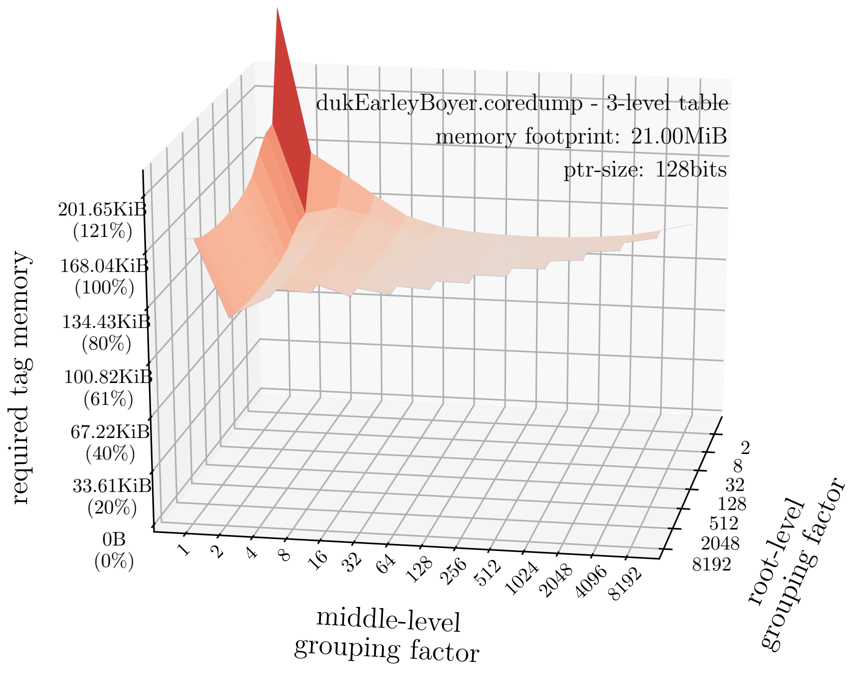
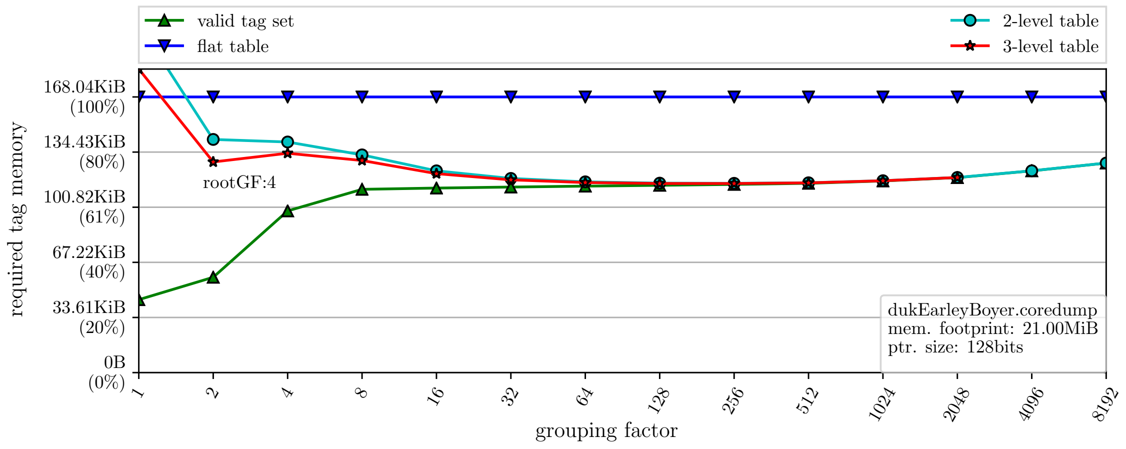


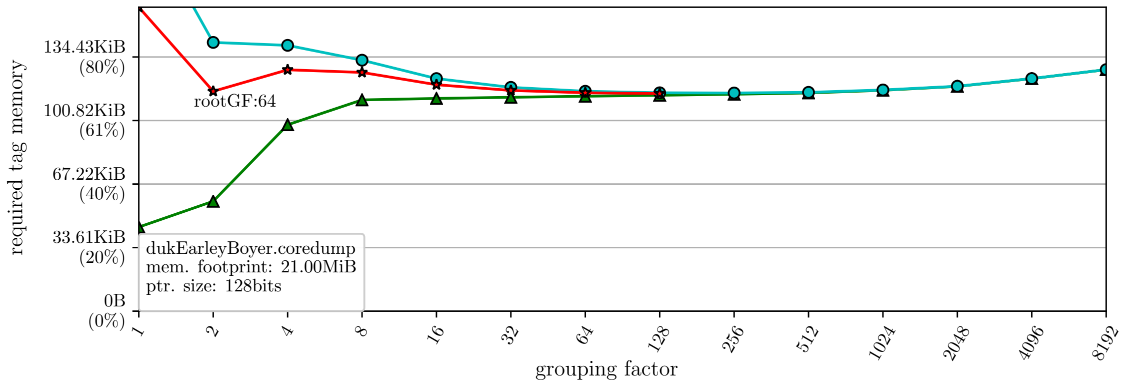
Figure 5.14: *A 3-level table for "Earley-Boyer". Valleys in the surface are configuration of grouping factors leading to lower tag footprint*

a root grouping factor of 4, 38% for a 64 root grouping factor, and 37% for a 256 root grouping factor. For the high pointer density application, the optimal 3-level footprint only reaches between 99% and 100% of the 2-level one. This is consistent with the fact that for pointer dense cases, all leaf nodes are required to be cached. Overall, the vast majority of the benefits provided by the multi-level table approach are present already in the 2-level table[5]. The next sections only consider results for 2-level tables.

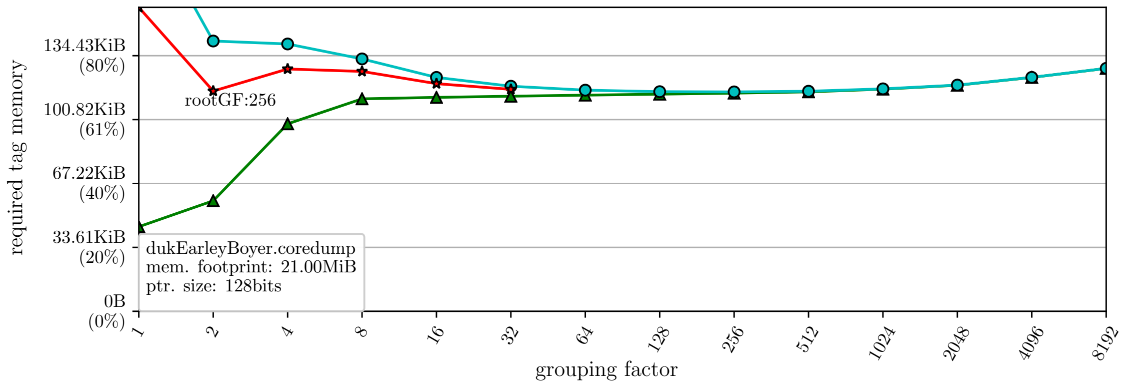
[5]This is especially true when looking at leaf granularities close to realistic cache line sizes (grouping factors around 256/512). It is difficult to take advantage of grouping factors smaller than the caching granularity as full cache blocks will be fetched regardless.



(a) *"Earley-Boyer" adding a 3-level table, root grouping factor 4*



(b) *"Earley-Boyer" adding a 3-level table, root grouping factor 64*



(c) *"Earley-Boyer" adding a 3-level table, root grouping factor 256*

Figure 5.15: *The leaf granularity on the x axis is the root level grouping factor for the 2-level table, and the middle level grouping factor for the 3-level table. With a root grouping factor of 4 the "3-level table" converges with the "valid tag set" faster than the "2-level table" curve. Increased root grouping factors of 64 and 256 don't have as significant an impact as they do for low pointer density applications*

5.4.3 A hierarchical tag cache simulator

Having initially explored static memory layout, we now explore dynamic hierarchical tag cache behaviour.

I augmented the python tag-cache simulator presented in **Section 5.3.1** with the ability to use the hierarchical tag table technique. I replay the DRAM traces already used for the dynamic study on the new simulator, using several different tag-cache line sizes and grouping factors for a two-level tag table, and obtain the results reported in **Figure 5.16**.

The first observation is that it is possible through the hierarchical table approach to push the tag DRAM traffic overheads to below 5% for even the high pointer density workloads (Earley-Boyer), and to almost completely eliminate them in the low pointer density workloads (FFMPEG). In general, we observe that "conventional" line sizes (i.e. 512 bits or 64 bytes) happen to perform best, and this for a grouping factor around the line size itself. Intuitively, this can be explained by the fact that eliminating leaf groups from the tag working set at a granularity smaller than the cache block size is not possible, as full cache blocks will be fetched regardless of

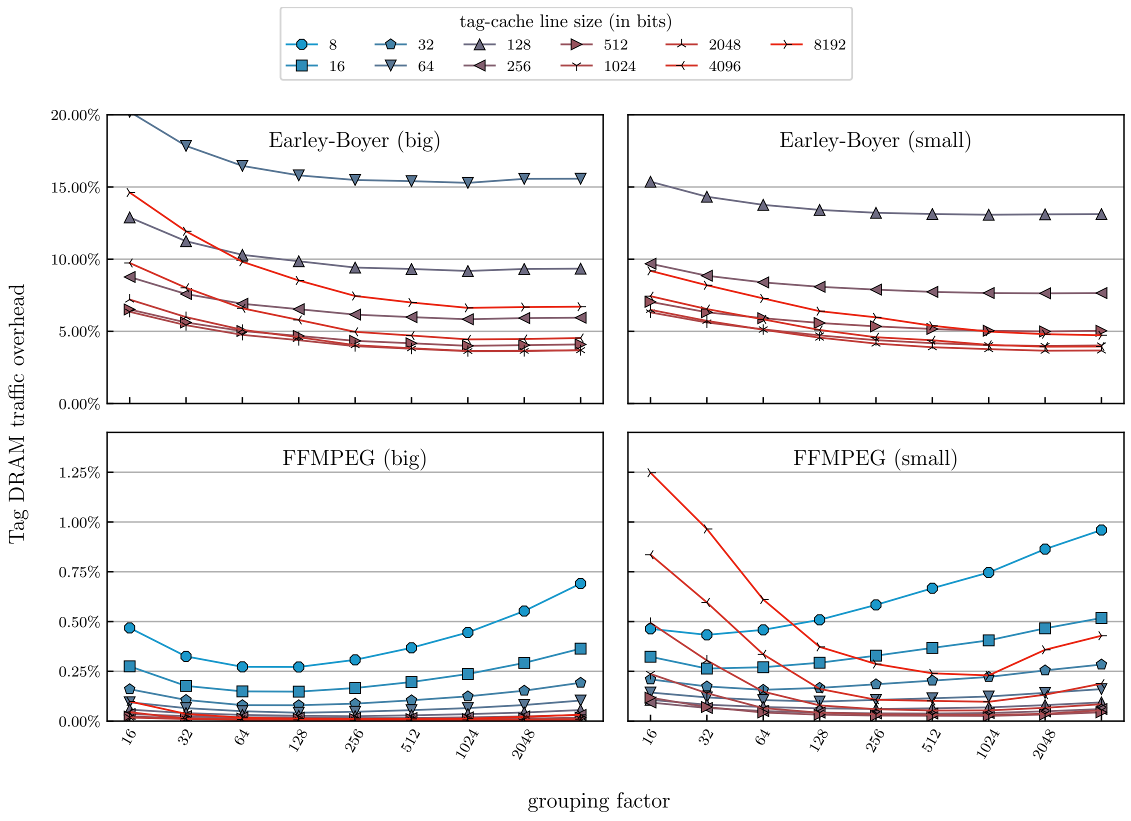


Figure 5.16: *Tag DRAM traffic overheads for a 2-level table and varying tag-cache line sizes, as the grouping factor increases. For the pointer heavy case, a typical line size of 512 bits performs best for a grouping factor of 512, with DRAM overheads dropping below 5%*

fine-grained knowledge in root level bits[6].

For “Earley-Boyer”, we observe that the overhead tends to decrease as the grouping factor increases up to a certain point, after which it stabilizes (and in rare cases slightly rises again). Excessive grouping factors do not seem to have a significantly detrimental effect on this application. Note that zooming in the 0% – 20% tag DRAM traffic overhead range for “Earley-Boyer” excludes the curves for the smallest line sizes from **Figure 5.16**.

The FFMPEG small case also has decreasing overheads for small grouping factors. As the line size grows past approximately 1024-bit, we can see overheads increase again. This is explained by the fact that a two level table needs cache entries for the root level bits at the same time as it needs entries for the leaf level bits, and FFMPEG small has a 32KiB cache that only contains 256 entries for 1024-bit lines, 128 for 2048-bit lines, 64 for 4096-bit lines and 32 for 8192-bit lines. This will cause an increase in conflict misses between table levels. Note that the FFMPEG graphs are zoomed between 0% and 1.4% overhead, effectively making this a negligible impact overall.

The study suggests that for a two level table, a standard cache line size of 512 bits with a grouping factor around 256 or 512 yields the best results in terms of limiting the memory traffic overheads due to tags. Such grouping factors, according to the static study from the previous section, are the ones where a two level table converges with the ideal curve of valid tag set to be cached. This means that the three level tables and deeper that converge for smaller grouping factor are not necessary to reach the optimal points for a 512-bit tag-cache line.

5.5 A hardware hierarchical tag cache implementation

I implement a Bluespec multi-level tag-cache controller that enables an FPGA evaluation of the mechanism on live workloads.

5.5.1 A generic module for exploration

In order to allow comparison between the base case and possible future exploration of different design points, I developed the multi-level tag table lookup engine in parametrized Bluespec System Verilog, allowing for tables of arbitrary depth, with arbitrary grouping factors at each depth level. It is composed of a tag lookup controller module, backed by a conventional generic cache-core module (the same used for CHERI’s L1 and L2 caches). As the lookup engine is on the path to DRAM, the extra cycle added per next level lookup of the table is negligible compared to the DRAM access time. Tag cache misses will still cost a DRAM access, but these should be infrequent as we have demonstrated that tag table accesses greatly exploit spatial locality.

The lookup engine is instantiated with a a 32KiB, 4-way associative, 1024-bit

[6]Note that in cases where pointers are interspersed with data in a cache line and only data locations are actively accessed, a grouping factor smaller than the line size can be beneficial.

	ALUTs	ALMs	Logic Registers	Block Memory bits
(128-bit) CHERI TOTAL	1.13 %	0.91 %	1.54 %	0 %
(128-bit) TAGCACHE	8.79 %	10.88 %	18.26 %	0 %
(256-bit) CHERI TOTAL	2.52 %	1.16 %	2.29 %	0 %
(256-bit) TAGCACHE	20 %	12.5 %	30.22 %	0 %

(a) *FPGA ressources usage overheads for a hierarchical tag-cache over a flat tag cache, for both 128-bit and 256-bit CHERI*

	ALUTs	ALMs	Logic Registers	Block Memory bits
128-bit CHERI flat	11.28 %	9.97 %	7.33 %	6.14 %
128-bit CHERI hierarchical	12.14 %	10.95 %	8.54 %	6.14 %
256-bit CHERI flat	10.17 %	9.85 %	6.87 %	6.12 %
256-bit CHERI hierarchical	11.91 %	10.96 %	8.74 %	6.12 %

(b) *Share of FPGA ressources used by the tag cache for 128-bit and 256-bit CHERI and both flat and hierarchical configurations*

Table 5.1: *Hierarchical tag-cache FPGA resource usage share and overheads*

line size cache. The cache-core's Block RAM can return 256 bits per cycle. To keep tag table maintenance simple, all bits for a given leaf node should be available at the same time to avoid need for complex state-machine fetching sequences of bits over multiple cycles. When zeroes are being written, the whole leaf node is tested at once, and if all bits are cleared, a zero is folded back into the root level bit, keeping the table in a consistent state at all times. Benchmarks were run on a bitfile with a two level table, using a grouping factor of 256 to best exploit the cache-core's BRAM width.

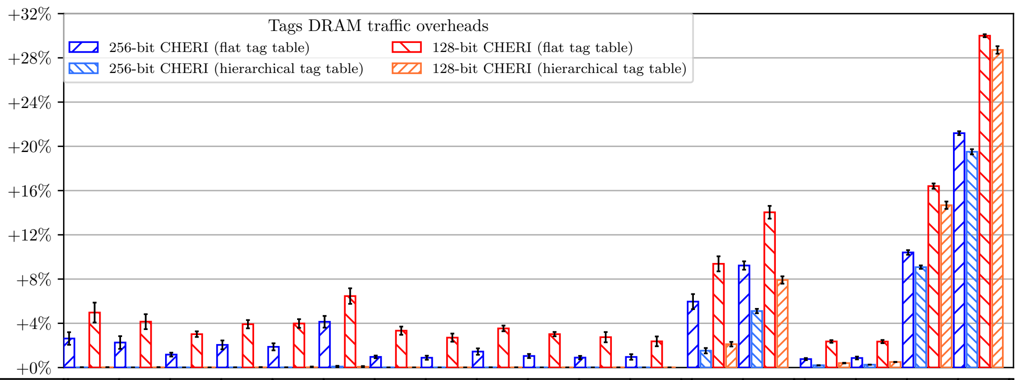
I introduce additional counters to the set of statcounters in hardware and in CheriBSD libstatcounters in order to measure the total number of accesses undertaken by the tag cache. The impact of the tag cache configuration is presented in the next paragraphs. Note that the redundant stores elimination optimisation previously introduced is not implemented in hardware (the results presented in the next paragraphs could therefore be improved with optimization work).

Table 5.1a shows that the hierarchical tag controller consumes more logic elements than the flat tag controller. These numbers are averaged across 15 syntheses of the CHERI processor, and it is important to note that the Altera Quartus FPGA toolchain is not necessarily representative of the overheads that could be seen on an ASIC implementation. The memory resources usage remains identical since the tag cache size does not change. The overhead consists of comparison logic and buffering registers to maintain the hierarchical structure. As a point of reference, **Table 5.1b** shows the share of the complete design used by the tag cache itself, for hierarchical and flat configurations. We see that in both the 128-bit and the 256-bit cases, the hierarchical tag cache share of the total resources usage is around 1% or 2% bigger than the flat configuration.

5.5.2 Running benchmarks with a hierarchical tag cache

Figure 5.17 presents the tags DRAM traffic overheads for 256-bit and 128-bit CHERI, for both a flat and a hierarchical table. For most applications, we see the overheads reduced to a fraction of a percent with a hierarchical table, drastically improving over a flat table. Note this is traffic overhead and not absolute traffic. **Figure 5.18** presents the results of the same benchmark runs as miss per thousand instructions to give a better overview of absolute numbers.

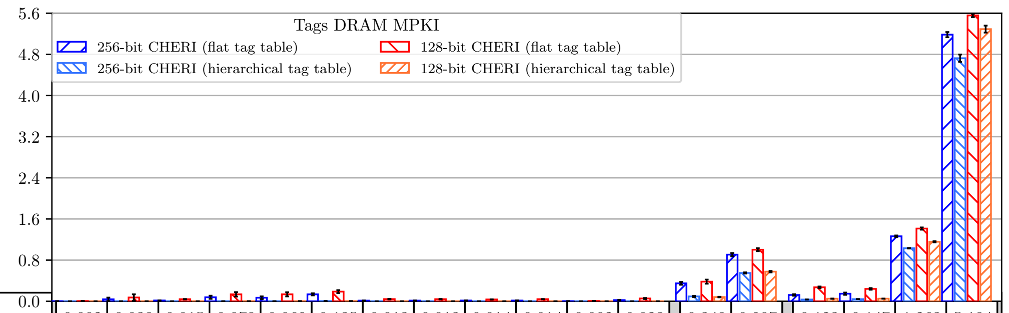
For non-pointer-heavy applications, the root level of the table is enough to serve most of the tags requests, explaining the extremely low overheads. For pointer heavy applications, some overhead is still present, but we can see that they are still reduced from the overheads present in the flat table case.



	automotive-bitcount	automotive-susan	automotive-qsort	consumer-jpeg	network-patricia	office-stringsearch	security-blowfish	security-rijndael	security-sha	telecomm-CRC32	telecomm-FFT	telecomm-adpcm	octane-earley-boyer	octane-splay	olden-treeadd	olden-perimeter	olden-mst	olden-bisort
hierarchical tag table 256-bit CHERI (%)	0.02	0.01	0.00	0.02	0.01	0.09	0.00	0.00	0.01	0.00	0.00	0.00	1.52	5.11	0.21	0.26	9.08	19.51
hierarchical tag table 128-bit CHERI (%)	0.02	0.02	0.00	0.02	0.02	0.07	0.00	0.00	0.01	0.00	0.00	0.00	2.11	7.92	0.40	0.50	14.68	28.71

Figure 5.17: *Share of the DRAM traffic due to capability validity tags. The hierarchical table consistently reduce the tags traffic overhead to well below 5% in most cases*

Overall, in this chapter, we have seen how conventional hierarchical structures for optimising storage of sparse information can be applied to tagged memory. This works particularly well in the CHERI tagged pointer case, and enable us to greatly improve cacheability of capability validity tags, making the CHERI approach viable for commercial implementation.



	automotive-bitcount	automotive-susan	automotive-qsort	consumer-jpeg	network-patricia	office-stringsearch	security-blowfish	security-rijndael	security-sha	telecomm-CRC32	telecomm-FFT	telecomm-adpcm	octane-earley-boyer	octane-splay	olden-treeadd	olden-perimeter	olden-mst	olden-bisort
flat tag table 256-bit CHERI	0.002	0.038	0.015	0.078	0.069	0.135	0.013	0.013	0.014	0.014	0.002	0.023	0.348	0.907	0.123	0.147	1.263	5.184
hierarchical tag table 256-bit CHERI	0.000	0.000	0.000	0.001	0.001	0.003	0.000	0.000	0.000	0.000	0.000	0.000	0.094	0.551	0.034	0.042	1.031	4.726
flat tag table 128-bit CHERI	0.004	0.074	0.040	0.135	0.135	0.188	0.042	0.041	0.035	0.041	0.006	0.054	0.382	1.003	0.272	0.240	1.414	5.558
hierarchical tag table 128-bit CHERI	0.000	0.000	0.000	0.001	0.001	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.082	0.576	0.050	0.051	1.156	5.291

Figure 5.18: *Tags DRAM miss per thousand instructions. The data is also presented as a table for readability.*

Chapter 6

Conclusion

In **Chapter 1** I hypothesized that it was possible to reduce the memory footprint of capabilities, and that an efficient tagged memory system could be implemented using off the shelf memory. I synthesize here how these claims were addressed, summarize the work that has been presented throughout this document, and open future research avenues.

6.1 Compressed capabilities for reduced memory footprint

I proposed a framework for pointer compression improving on ideas from underutilised existing compression schemes. In particular, it allows for more features than the Low-Fat pointer format, such as the ability to perform dynamic bounds checking, and also to enable the representation of out-of-bounds pointers, which is required for real world applications. This compression approach can be implemented in an inexpensive manner as it does not require fields to be uncompressed in order to perform computation, enabling fast load to use delay and reduced sized arithmetic for shorter critical path while avoiding large registers to store the uncompressed values.

I derive a new CHERI capability format with half the memory footprint of the original CHERI format, with very few compromises. For object sizes greater than 2^{20} bytes (that is 1MiB), the new format introduces alignment requirements. The alignment requirements of a capability are small (a few bytes) for object sizes on the order of a few MiB, and grows with the object size. Additionally, the new format does not allow pointers to venture arbitrarily far out of the allocated object bounds and still be representable, but guarantees a minimum buffer of 4KiB both above and below the object. This format is able to boot the CheriBSD operating system, both in the L3 simulation model and on FPGA, and to run a substantial corpus of C-code recompiled to use capabilities for every pointer and return address.

I also introduce further possible optimisations to the pointer compression framework without evaluating these in this work, but opening research avenues for fields where greater compression may be required (e.g. 32-bit processors).

6.2 Efficient tagged memory for capability validity tags

I presented a method to optimize caching of capability validity tags leveraging their sparse in-memory distribution pattern through a hierarchical tag table. I store this table in a dedicated region of DRAM, and cache all its levels in a dedicated multi-level tag cache at the end of the cache hierarchy, conceptually in the DRAM controller. This avoids consistency issues in the cache hierarchy, and acts as a tag-awareness shim to DRAM. This effectively enables us to build a CHERI SoC as exclusively composed of tag-aware modules, i.e. manipulating tags and data atomically.

The use of a hierarchical tag cache with the backing tag table stored in a dedicated region of the DRAM makes CHERI the processor with the most efficient tagged memory implementation compatible with standard memory.

6.3 Further contributions

6.3.1 Capability processor formal model

I developed a CHERI architectural model in the L3 formal specification language as an extension to Anthony Fox's existing MIPS model. It serves as a CHERI golden model and is now used to formally prove properties of the CHERI mechanism. Support for various features have later been added, such as cache modelling and cache coherence for multicore versions, and it has been used as an architectural exploration tool to develop the ideas presented in **Chapter 4**.

It is the first formal model of a capability processor capable of booting a full operating system and of running capability code, being used for formal proof of architectural features of a capability processor, fuzzing capability hardware, and to help architectural exploration.

6.3.2 Event monitoring toolkit

I designed and implemented a toolkit for precise monitoring of processor events. I implemented a Bluespec SystemVerilog module for counting arbitrary events. It allows for a value of the "ModuleEvent" type (conceptually a bundle of boolean values signifying the occurrence of specific events) to be observed every cycle, and for an arbitrary number of modules to be under observation. Each event observed is instantly accumulated in a small register, and continuously merged back in a memory that can later be queried for information on dynamic statistics of the system. It is integrated in the CHERI hardware to report memory events like cache hits and misses for all levels of caches and DRAM traffic, and is query-able through the MIPS "RDHWR" instruction.

I also implement a CheriBSD (port of the FreeBSD operating system on the CHERI architecture) library, libstatcounters, that interfaces with CHERI's new hardware counter module and allows for easy sampling of counter values. It is possible to use the library's functions in arbitrary user code after the operating system enables the counter feature. It is also possible to directly link a binary against the library without interfering with the sources, and get an automatic counter sampling at the beginning and at the end of the `main()` function.

This toolkit enables easy benchmarking on a large scale and with real hardware directly on FPGA.

6.3.3 Continuous integration for open hardware

I created and maintained several jobs within the Jenkins continuous integration framework that enables regular automatic testing of new additions to the different implementations of the CHERI processor. In particular, I worked with Theodore Markettos on the renovation of the Bluehive machine with 16 DE4 FPGA boards, and its repurposing as a Jenkins slave so that actual hardware tests can be done automatically and routinely, and so that benchmarks can be run on real hardware regularly as well.

This demonstrates that continuous integration techniques used in software projects can be applied to an open source hardware workflow, and be integrated with the software flow for great benefits in hardware-software co-design, in particular when leveraging the event monitoring toolkit.

6.4 Future work

This research aims at making CHERI a computer system that enable high performance memory safety. To conclude this work, I present other research avenues that can still be explored to further improve the performances of a CHERI system.

6.4.1 Capability assisted hardware prefetching

In CHERI, capability validity tags and data move together throughout the cache hierarchy. The capability validity tag can be regarded as a piece of metadata identifying with certainty which data is a valid pointer and which one is not. In a similar fashion to Cooksey [25], it is possible to build a prefetching mechanism built around pointer detection. In CHERI, the heuristics for virtual address detections are trivially simplified to testing a single tag bit.

I conducted a preliminary exploration of this technique in the L3 model that was augmented with caches. Using the capability validity tags as a filter to identify valid pointers yields significantly less TLB accesses than a naive approach testing for a TLB hit on every memory fetch, and does not require significant additions to the code for virtual address detection heuristics. The L3 model being an instruction level architectural model simulator, all memory subsystem side effect happen atomically with the instruction being simulated, which does not allow for useful microarchitectural results, but does demonstrate functional correctness of the mechanism. I also implemented an initial Bluespec version of a capability pre-fetcher, but did not evaluate its impact on cache traffic and overall performance. A mature mechanism would try to leverage the extra information available in a capability such as its size or its access permissions to precisely direct a prefetching policy.

6.4.2 Fast protection domain crossing

CHERI provides a means to describe protection domains with a pair of code and data capabilities sealed with the same object type. The `CCall` allows for protection domain switching by installing the unsealed code capability in PCC and the unsealed data capability in IDC. The current implementation of this instruction performs a series of tests on the capabilities it is invoked with, and triggers a software exception, passing the control flow to the kernel that can backup and clear the register file, and possibly keep track of a protection domain trusted call stack before manually unsealing the two capabilities with the `CUnseal` instruction.

I implemented a faster version of this `CCall` instruction in the L3 model. It conducts a series of tests on the capabilities it is invoked with, and performs the unsealing and installation in PCC (Program Counter Capability) and IDC (Invoked Data Capability) of the unsealed versions. It leaves the register file untouched and does not keep track of a protection domain trusted call stack. I test this instruction using a simple piece of assembly code with one protection domain having a pair of sealed capabilities to another protection domain, `CCall` into it leaving a pair of sealed capabilities to the first protection domain, and `CCall` back into the first protection domain.

This primitive opens exploration for fast protection domain crossing with a variety of different trust models.

6.4.3 Pushing the presented optimisations further

For future very large address space needs, possibly given rise to by the use of non-volatile memory for warehouse scale computing, it is conceivable to consider a 256-bit capability format leveraging the pointer compression framework with a 128-bit virtual address. On the other hand, for embedded applications like internet of things, a full 64-bit address space may not be required, and the pointer compression framework can be exploited to derive a 64-bit capability format with 32-bit virtual addresses.

Finally, the efficient tagged memory implementation that has been presented relies on the in memory pattern observed for capability validity tags. However, other kind of tags that expose sparse patterns can also leverage the same technique. The tag cache python simulator can be used to explore this technique with code pointer tags, zero memory tags, etc. Exploring this avenue could lead to cheap primitive for common operations such as zeroing memory.

Appendix A

Posits-based exponent encoding

A.1 Introduction to posits

Posits are a recent encoding for representing floating point numbers, described by John Gustafson [53]. Posits have interesting properties such as straight forward inverse operations by negating certain bits of the number represented. They are able to maximize dynamic range and accuracy. They do not have "NaN" (Not a Number values), nor do they overflow to infinity or underflow to zero the way IEEE floats do.

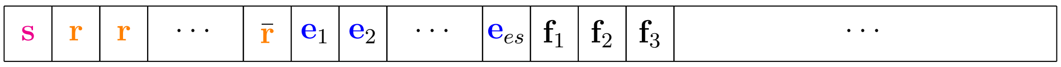


Figure A.1: *Posit numbers representation showing the sign **s**, the **regime**, the **exponent** and the **fraction** fields*

Figure A.1 presents the posits format, with the sign field $\mathbf{s}$ in magenta, the **regime** field in orange represented by $\mathbf{r}$ and $\bar{\mathbf{r}}$, the **exponent** field in blue represented by $\mathbf{e}_x$, and the **fraction** field in black represented by $\mathbf{f}_y$.

The $\mathbf{s}$ bit simply gives the sign of the encoded number. The value of the **regime** field can be determined by counting the number of consecutive $\mathbf{r}$ until the first $\bar{\mathbf{r}}$. There is a fixed number **es** of bits to encode the value of the **exponent** field (that is the value in the **es** $\mathbf{e}_x$ bits), and the remaining bits are used for the value of the **fraction** field. Posits also define the "**useed**" value to be "$2^{2^{\mathbf{es}}}$". With these fields, a posit number value is expressed as follows:

$$(-1)^{\mathbf{s}} \times \mathbf{useed}^{\mathbf{regime}} \times 2^{\mathbf{exponent}} \times (1 + \mathbf{fraction})$$

In the context of CHERI capability compression, the interesting property of posits is that they trade off **fraction** bits for **regime** bits. The **regime** bits are used to shape a fast growing power-of-two factor by which the represented number is multiplied. Effectively, posits gives small numbers a larger **fraction** field, leading to more precision than for large numbers. The **exponent** and **regime** fields can be thought of as a single factor, where a trade-off exists between bits for this factor and bits for the **fraction** field.

If we think of the $\mathbf{top_{bits}}$ and $\mathbf{base_{bits}}$ fields of **Chapter 4** as the **fraction** field, we have the possibility to trade off alignment requirements on the **top** and **base** addresses for expressiveness of the **e** exponent value. The next section derives an encoding scheme for CHERI using this principle.

A.2 Extending the posits approach with 2-bit tokens

For CHERI capability compression, unlike simple posit numbers, we are simultaneously encoding two values (the two bounds of a capability sharing the same exponent). Some modifications to the posit scheme are required. In particular, we will not consume bits one at a time from the **fraction** field when building up the **regime** field; as we want to keep the same alignment requirements for $\mathbf{top_{bits}}$ and

base$_{\textbf{bits}}$ (corresponding to the **fraction** field), we will try to consume the same number of bits from each. If we consume one bit from each field, we get two bits for the exponent (or **regime**) at a time. The **regime** field can be thought of as being composed of successive tokens, each either a *continuation token* or a *termination token* indicating the end of the **regime** field. In the original scheme, each token is a single bit. In this section, I explore an extension to the posits technique using 2-bit tokens.

Let us consider a **regime** field using 2-bit tokens. Each 2-bit token can have $2^2 = 4$ different values. Each value can either be a continuation value, encoding a *continuation token*, or a termination value, encoding a *termination token*. Let us call **num_cont_values** the number of continuation values, and **num_term_values** the number of termination values, and choose **num_cont_values** $= 2$ and **num_term_values** $= 2$.

A valid **regime** field is composed of a sequence of zero or more *continuation tokens* followed by a single *termination token*.

- The number of combinations expressible by a **regime** field composed of exactly one termination token is 2.
- The number of combinations expressible by a **regime** field composed of exactly one continuation token followed by a termination token is $2 \times 2 = 4$.
- The number of combinations expressible by a **regime** field composed of exactly two continuation tokens followed by a termination token is $2 \times 2 \times 2 = 8$.

For any **regime** field consuming up to three tokens, the total number of expressible values is the sum of the number of expressible values for one token (the single termination token string), two tokens (the one continuation token followed by one termination token string) and three tokens (the one continuation tokens followed by one termination token string), i.e. $2 + 2 \times 2 + 2 \times 2 \times 2 = 2 + 4 + 8 = 14$.

This means for example that when encoding the CHERI exponent with this technique, we could express up to 14 different exponent values in the bottom bits of the existing **top$_{\textbf{bits}}$** and **base$_{\textbf{bits}}$** fields taking at most three bits from each (a total of 6 bits), incurring at worst an 8-byte additional alignment requirement. In comparison, the initial posits encoding scheme can only express 5 different **regime** values with 6 bits (and 10 when considering that both strings of zeroes terminated by a one and strings of ones terminated by zero can be used).

A.3 Generalising to multi-bit tokens

Let us consider a **regime** field using x-bit tokens. Each x-bit token can have 2^x different values. Let us call **num_cont_values** the number of continuation values allocated for *continuation tokens*, and **num_term_values** the number of termination values allocated for *termination tokens*, such that we have **num_cont_values** $= 2^x -$ **num_term_values** (where $0 <$ **num_cont_values** $< 2^x$).

As before, a valid regime field is composed of a sequence of zero or more *continuation tokens* followed by a single *termination token*. In general, the number of combinations expressible by a regime field composed of exactly **num_cont_tokens** continuation tokens before a termination token is $\mathbf{num\_cont\_values}^{\mathbf{num\_cont\_tokens}} \times$ **num_term_values**.

For any regime field of up to **max_cont_tokens** continuation tokens, a termination token could terminate the string early in any of the **num_cont_tokens** first slots. As for the 2-bit token case, the total number of expressible values is the sum of the number of values expressible by each individual possible token strings given by:

$$\sum_{i=0}^{\mathbf{max\_cont\_tokens}} \left(\mathbf{num\_cont\_values}^{i} \times \mathbf{num\_term\_values}\right)$$

If we also consider a **es**-bit dedicated external exponent field ($\mathbf{es} \geq 0$) together with the x-bits tokens regime field, we can use this new encoding to express up to

$$\sum_{i=0}^{\mathbf{num\_cont\_tokens}} \left(\mathbf{num\_cont\_values}^{i} \times \mathbf{num\_term\_values} \times 2^{\mathbf{es}}\right)$$

Note the exponential growth of the range with the number of continuation tokens. Token value allocation controls the magnitude of this exponential growth; more continuation values **num_cont_values** can access greater numbers for a large **max_cont_tokens**; more termination values **num_term_values** give a wider range of accessible numbers for a small **max_cont_tokens**.

For x-bit tokens, it is possible to allocate a single value for continuation tokens and $2^x - 1$ values for termination tokens, leading to a linear growth of the exponent range's size. It is also possible to allocate a single value for termination tokens and $2^x - 1$ values for continuation tokens, leading to an extreme exponential growth of the exponent range's size. In general, token value allocation controls the growth of the exponent range with respect to the regime's size.

A.4 Applying the new scheme to CHERI

For CHERI capability compression, we use our new posits-based encoding to compress the **e** field in the $\mathbf{top_{bits}}$ and $\mathbf{base_{bits}}$ fields. **Figure A.2** shows a new CHERI encoding for the exponent. We have one regime bit $\mathbf{r_0}$ allocated outside the $\mathbf{top_{bits}}$ and $\mathbf{base_{bits}}$. This dedicated bit either gives an exponent of 0 with no additional alignment requirement on $\mathbf{top_{bits}}$ and $\mathbf{base_{bits}}$, or indicates that we must interpret the least significant bits of those fields as more regime bits, effectively encoding a bigger exponent, and imposing alignment requirements by implying a zero value of those bits in the original fields.

An example of the new CHERI format with a single dedicated regime bit is presented in **Figure A.3**, with the $\mathbf{top_{bits}}$, $\mathbf{base_{bits}}$ and **e** mapped to a new **TopBaseExp** field.

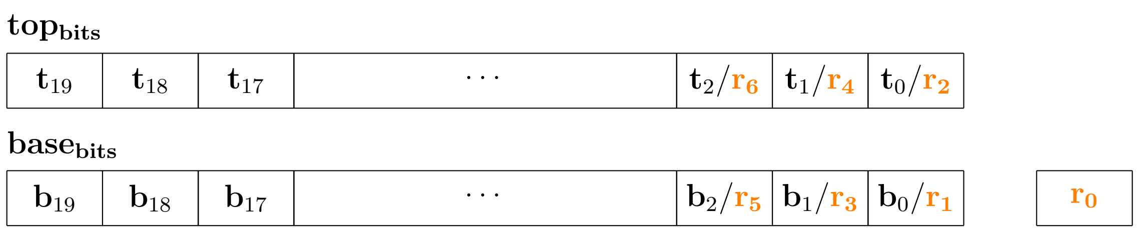
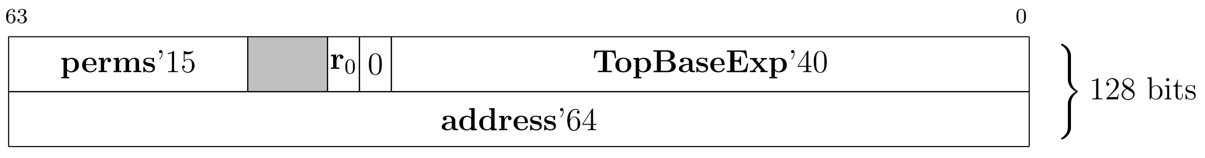
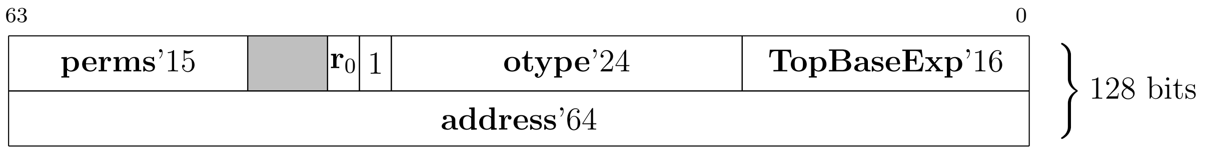


Figure A.2: *Posits based exponent encoding for CHERI with least significant bits of the* ***top_bits*** *and* ***base_bits*** *fields used as tokens to represent the* ***e*** *field*



(a) *Unsealed CHERI-128 representation of a capability with posits-based encoding of* ***e***



(b) *Sealed CHERI-128 representation of a capability with posits-based encoding of* ***e***

Figure A.3: *Posits-based 128-bit CHERI memory representation for unsealed and sealed capabilities*

A.5 Observations on the new scheme

Overall, more reserved bits are made available in this scheme, specifically 5 more bits gained from the exponent field presented in **Chapter 4** for a single dedicated regime bit.

Having a single dedicated regime bit gives full precision for the first exponent value. Having 2 dedicated regime bits would give full precision for the first two exponent values, etc. Actual use cases are likely not to require full precision for non zero **e** values.

Note that when there is a known finite number of exponent values to encode (which is the case in CHERI), the position of the last possible token in the token string is statically known (i.e. we will never count further than the maximum value and will not need to consume more tokens). This last token, by definition, does not need to encode a continuation value (i.e. for an x-bit token, all 2^x values can be used for the termination token), leaving all combinations available for useful **e** value encoding. This observation can be used to further optimize the new scheme.

References

[1] Abadi, Martín et al. ‘Control-flow Integrity’. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS ’05. New York, NY, USA: ACM, 2005, pp. 340–353.

[2] Abadi, Martín et al. ‘Control-flow Integrity Principles, Implementations, and Applications’. In: *ACM Trans. Inf. Syst. Secur.* 13.1 (Nov. 2009), 4:1–4:40.

[3] Ackerman, William B. and Plummer, William W. ‘An Implementation of a Multiprocessing Computer System’. In: *Proceedings of the First ACM Symposium on Operating System Principles*. SOSP ’67. New York, NY, USA: ACM, 1967, pp. 5.1–5.10.

[4] Akritidis, P. et al. ‘Preventing Memory Error Exploits with WIT’. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. May 2008, pp. 263–277.

[5] ALTERA (now part of Intel). *Quartus® Prime Standard Edition Handbook Volume 1: Design and Synthesis*. Vol. 1. Aug. 2017. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/qts-qps-handbook.pdf (visited on 25/08/2017).

[6] Alves, Tiago and Felton, Don. *TrustZone: Integrated Hardware and Software Security-Enabling Trusted Computing in Embedded Systems (July 2004)*.

[7] ARM. *ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*.

[8] Arora, Divya et al. ‘Enhancing Security Through Hardware-assisted Run-time Validation of Program Data Properties’. In: CODES+ISSS ’05. New York, NY, USA: ACM, 2005, pp. 190–195.

[9] Arvind. ‘Bluespec and Haskell’. In: *Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-specific Languages*. FPCDSL ’13. New York, NY, USA: ACM, 2013, pp. 1–2.

[10] Bletsch, Tyler et al. ‘Jump-oriented Programming: A New Class of Code-reuse Attack’. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’11. New York, NY, USA: ACM, 2011, pp. 30–40.

[11] Brown, Jeremy et al. *A capability representation with embedded address and nearly-exact object bounds*. Tech. rep. Project Aries Technical Memo 5, http://www. ai. mit. edu/projects/aries/Documents/Memos/ARIES-05. pdf, 2000.

[12] Campbell, Brian and Stark, Ian. ‘Extracting Behaviour from an Executable Instruction Set Model’. In: *Proceedings of the 16th Conference on Formal Methods in Computer - Aided Design (FMCAD 2016)*. FMCAD Inc, 2016, pp. 33–40.

[13] Carlisle, Martin C. and Rogers, Anne. ‘Software Caching and Computation Migration in Olden’. In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPOPP ’95. New York, NY, USA: ACM, 1995, pp. 29–38.

[14] Carter, Nicholas P., Keckler, Stephen W. and Dally, William J. ‘Hardware Support for Fast Capability-based Addressing’. In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS VI. New York, NY, USA: ACM, 1994, pp. 319–327.

[15] Carvalho, M. et al. ‘Heartbleed 101’. In: *IEEE Security Privacy* 12.4 (July 2014), pp. 63–67.

[16] Checkoway, Stephen et al. ‘Return-oriented Programming Without Returns’. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security.* CCS ’10. New York, NY, USA: ACM, 2010, pp. 559–572.

[17] Chen, Ping et al. ‘DROP: Detecting Return-Oriented Programming Malicious Code’. en. In: *Information Systems Security.* Springer, Berlin, Heidelberg, Dec. 2009, pp. 163–177.

[18] Chen, S. et al. ‘Defeating memory corruption attacks via pointer taintedness detection’. In: *2005 International Conference on Dependable Systems and Networks (DSN’05).* June 2005, pp. 378–387.

[19] Chen, Shuo et al. ‘Non-Control-Data Attacks Are Realistic Threats’. In: *USENIX Security’05.* 2005.

[20] Chew, Monica and Song, Dawn. *Mitigating buffer overflows by operating system randomization.* Tech. rep. 2002.

[21] Chisnall, David et al. ‘Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine’. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS ’15. New York, NY, USA: ACM, 2015, pp. 117–130.

[22] Chisnall, David et al. ‘CHERI JNI: Sinking the Java Security Model into the C’. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS ’17. New York, NY, USA: ACM, 2017, pp. 569–583.

[23] Clark, B. E. and Corrigan, M. J. ‘Application System/400 Performance Characteristics’. In: *IBM Syst. J.* 28.3 (July 1989), pp. 407–423.

[24] Colwell, Robert P., Gehringer, Edward F. and Jensen, E. Douglas. ‘Performance Effects of Architectural Complexity in the Intel 432’. In: *ACM Trans. Comput. Syst.* 6.3 (Aug. 1988), pp. 296–339.

[25] Cooksey, Robert, Jourdan, Stephan and Grunwald, Dirk. ‘A Stateless, Content-directed Data Prefetching Mechanism’. In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS X. New York, NY, USA: ACM, 2002, pp. 279–290.

[26] Crandall, J. R. and Chong, F. T. ‘Minos: Control Data Attack Prevention Orthogonal to Memory Model’. In: *37th International Symposium on Microarchitecture, 2004. MICRO-37 2004.* Dec. 2004, pp. 221–232.

[27] Crispan Cowan et al. ‘StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks’. In: *Proceedings of the 7th USENIX Security Symposium*. San Antonio, Texas, Jan. 1998.

[28] *CVE-2014-0160*. MITRE, CVE-ID CVE-2014-0160. Dec. 2013. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 (visited on 19/05/2017).

[29] *CVE-2015-1538*. MITRE, CVE-ID CVE-2015-1538. Feb. 2015. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1538 (visited on 19/05/2017).

[30] *CVE-2017-0144*. MITRE, CVE-ID CVE-2017-0144. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0144 (visited on 27/07/2017).

[31] Dalton, Michael, Kannan, Hari and Kozyrakis, Christos. ‘Raksha: A Flexible Information Flow Architecture for Software Security’. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA ’07. New York, NY, USA: ACM, 2007, pp. 482–493.

[32] Davi, Lucas, Sadeghi, Ahmad-Reza and Winandy, Marcel. ‘Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-oriented Programming Attacks’. In: *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*. STC ’09. New York, NY, USA: ACM, 2009, pp. 49–54.

[33] Denning, Peter J. ‘Virtual Memory’. In: *ACM Comput. Surv.* 2.3 (Sept. 1970), pp. 153–189.

[34] Denning, Peter J. ‘Virtual Memory’. In: *ACM Comput. Surv.* 28.1 (Mar. 1996), pp. 213–216.

[35] Dennis, Jack B. and Van Horn, Earl C. ‘Programming Semantics for Multiprogrammed Computations’. In: *Commun. ACM* 9.3 (Mar. 1966), pp. 143–155.

[36] Devices, A Micro. *AMD64 architecture programmer’s manual volume 2: System programming*. September, 2006.

[37] Durumeric, Zakir et al. ‘The Matter of Heartbleed’. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC ’14. New York, NY, USA: ACM, 2014, pp. 475–488.

[38] Etoh, H. and Yoda, K. *Protecting from stack-smashing attacks*. June 2001. http://www.trl.ibm.com/projects/security/ssp (visited on 30/06/2001).

[39] Fotheringham, John. ‘Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store’. In: *Commun. ACM* 4.10 (Oct. 1961), pp. 435–436.

[40] Fox, Anthony. *L3*. http://www.cl.cam.ac.uk/~acjf3/l3/l3.pdf (visited on 25/08/2017).

[41] Fox, Anthony C. J. ‘Directions in ISA Specification’. In: *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*. Aug. 2012.

[42] Greathouse, Joseph L. et al. 'A Case for Unlimited Watchpoints'. In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 159–172.

[43] Guthaus, M. R. et al. 'MiBench: A free, commercially representative embedded benchmark suite'. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. Dec. 2001, pp. 3–14.

[44] Hansen, Paul M. et al. 'A Performance Evaluation of the Intel iAPX 432'. In: *SIGARCH Comput. Archit. News* 10.4 (June 1982), pp. 17–26.

[45] IBM. *PowerPC Operating Environment Architecture Book III.*

[46] IMAGINATION TECHNOLOGIES. *MIPS® Architecture For Programmers Volume III: MIPS64® / microMIPS64TM Privileged Resource Architecture.*

[47] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide.*

[48] *Introduction to Intel® Memory Protection Extensions — Intel® Software.* https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions (visited on 08/12/2017).

[49] Jaleel, Aamer. 'Memory characterization of workloads using instrumentation-driven simulation'. In: *Web Copy: http://www. glue. umd. edu/ajaleel/workload* (2010).

[50] jenkinsci-docs@googlegroups.com. *Jenkins User Handbook.* https://jenkins.io/user-handbook.pdf (visited on 25/08/2017).

[51] Jim, Trevor et al. 'Cyclone: A Safe Dialect of C'. In: *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference.* ATEC '02. Berkeley, CA, USA: USENIX Association, 2002, pp. 275–288.

[52] Joannou, Alexandre et al. 'Efficient Tagged Memory'. In: *35th IEEE International Conference on Computer Design*. Nov. 2017.

[53] *John Gustafson presents: Beyond Floating Point - Next Generation Computer Arithmetic*. Feb. 2017. https://insidehpc.com/2017/02/john-gustafson-presents-beyond-floating-point-next-generation-computer-arithmetic (visited on 25/08/2017).

[54] Jones, Richard W. M. and Kelly, Paul H. J. 'Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs'. In: *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97).* Linköping University Electronic Press; Linköpings universitet, 1997, pp. 13–26.

[55] *jsrun: Experimental JavaScript interpreter for FreeBSD*. original-date: 2016-01-25T12:36:07Z. Jan. 2016. https://github.com/CTSRD-CHERI/jsrun (visited on 21/06/2017).

[56] Kilburn, T. et al. 'One-Level Storage System'. In: *IRE Transactions on Electronic Computers* EC-11.2 (Apr. 1962), pp. 223–235.

[57] Kwon, Albert et al. 'Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security'. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. New York, NY, USA: ACM, 2013, pp. 721–732.

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

[59] Mayer, Alastair J. W. 'The Architecture of the Burroughs B5000: 20 Years Later and Still Ahead of the Times?' In: *SIGARCH Comput. Archit. News* 10.4 (June 1982), pp. 3–10.

[60] McKeen, Frank et al. 'Innovative instructions and software model for isolated execution.' In: *HASP@ ISCA* 10 (2013).

[61] Molina, Carlos, González, Antonio and Tubella, Jordi. 'Reducing memory traffic via redundant store instructions'. en. In: *High-Performance Computing and Networking*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Apr. 1999, pp. 1246–1249.

[62] Moore, S. W. et al. 'Bluehive - A field-programable custom computing machine for extreme-scale real-time neural network simulation'. In: *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. Apr. 2012, pp. 133–140.

[63] Nagarakatte, Santosh et al. 'CETS: Compiler Enforced Temporal Safety for C'. In: *Proceedings of the 2010 International Symposium on Memory Management*. ISMM '10. New York, NY, USA: ACM, 2010, pp. 31–40.

[64] Nagarakatte, Santosh et al. 'SoftBound: Highly Compatible and Complete Spatial Memory Safety for C'. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. New York, NY, USA: ACM, 2009, pp. 245–258.

[65] Necula, George C., McPeak, Scott and Weimer, Westley. 'CCured: Type-safe Retrofitting of Legacy Code'. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. New York, NY, USA: ACM, 2002, pp. 128–139.

[66] Needham, R. M. and Walker, R. D.H. 'The Cambridge CAP Computer and Its Protection System'. In: *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*. SOSP '77. New York, NY, USA: ACM, 1977, pp. 1–10.

[67] Nethercote, Nicholas and Seward, Julian. 'Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation'. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[68] Nienhuis, Kyndylan. *Draft: Nonforgeability of capabilities in CHERI (unpublished)*. Tech. rep. Oct. 2016.

[69] Nikhil, R. 'Bluespec System Verilog: efficient, correct RTL from high level specifications'. In: *Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings.* June 2004, pp. 69–70.

[70] Nikhil, Rishiyur S and Czeck, Kathy R. *BSV by Example*. 2010.

[71] Olatunji Ruwase and Monica S. Lam. *A Practical Dynamic Buffer Overflow Detector*. Tech. rep. Feb. 2004.

[72] Qualcomm Technologies, Inc. *Pointer Authentication on ARMv8.3 - Design and Analysis of the New Software Security Instructions*. Tech. rep. Jan. 2017.

[73] Rogers, Anne et al. 'Supporting Dynamic Data Structures on Distributed-memory Machines'. In: *ACM Trans. Program. Lang. Syst.* 17.2 (Mar. 1995), pp. 233–263.

[74] Rosenband, Daniel L. and Arvind. 'Modular Scheduling of Guarded Atomic Actions'. In: *Proceedings of the 41st Annual Design Automation Conference*. DAC '04. New York, NY, USA: ACM, 2004, pp. 55–60.

[75] Saltzer, J. H. and Schroeder, M. D. 'The protection of information in computer systems'. In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308.

[76] Shacham, Hovav. 'The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)'. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561.

[77] Shapiro, Jonathan S., Smith, Jonathan M. and Farber, David J. 'EROS: A Fast Capability System'. In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP '99. New York, NY, USA: ACM, 1999, pp. 170–185.

[78] Shioya, R. et al. 'Low-Overhead Architecture for Security Tag'. In: *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. Nov. 2009, pp. 135–142.

[79] Shrobe, Howard, DeHon, Andre and Knight, Thomas. *Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (tiara)*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE, 2009.

[80] Shrobe, Howard, Knight, Thomas and Hon, Andre de. 'TIARA: Trust Management, Intrusion-tolerance, Accountability, and Reconstitution Architecture'. In: (2007).

[81] Song, C. et al. 'HDFI: Hardware-Assisted Data-Flow Isolation'. In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 1–17.

[82] Song, Wei, Bradbury, Alex and Mullins, Robert. 'Towards General Purpose Tagged Memory'. In: *2nd RISC - V workshop* (June 2015).

[83] Spafford, Eugene H. 'The Internet Worm Program: An Analysis'. In: *SIGCOMM Comput. Commun. Rev.* 19.1 (Jan. 1989), pp. 17–57.

[84] Suh, G. Edward et al. 'Secure Program Execution via Dynamic Information Flow Tracking'. In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XI. New York, NY, USA: ACM, 2004, pp. 85–96.

[85] Szekeres, L. et al. 'Eternal War in Memory'. In: *IEEE Security Privacy* 12.3 (May 2014), pp. 45–53.

[86] Szekeres, L. et al. 'SoK: Eternal War in Memory'. In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 48–62.

[87] Tiwari, Mohit et al. ‘A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags’. In: *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 94–105.

[88] Venkataramani, G. et al. ‘FlexiTaint: A programmable accelerator for dynamic taint propagation’. In: *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. Feb. 2008, pp. 173–184.

[89] Venkataramani, G. et al. ‘MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging’. In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Feb. 2007, pp. 273–284.

[90] Watson, R. N. M. et al. ‘CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization’. In: *2015 IEEE Symposium on Security and Privacy*. May 2015, pp. 20–37.

[91] Watson, R. N. M. et al. ‘Fast Protection-Domain Crossing in the CHERI Capability-System Architecture’. In: *IEEE Micro* 36.5 (Sept. 2016), pp. 38–49.

[92] Watson, Robert N. M. et al. ‘A Taste of Capsicum: Practical Capabilities for UNIX’. In: *Commun. ACM* 55.3 (Mar. 2012), pp. 97–104.

[93] Watson, Robert N. M. et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture*. Tech. rep. UCAM-CL-TR-876. University of Cambridge, Computer Laboratory, Sept. 2015.

[94] Watson, Robert N. M. et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5)*. Tech. rep. UCAM-CL-TR-891. University of Cambridge, Computer Laboratory, June 2016.

[95] Watson, Robert N. M. et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6)*. Tech. rep. UCAM-CL-TR-907. University of Cambridge, Computer Laboratory, Apr. 2017.

[96] Watson, Robert NM et al. ‘Capsicum: Practical Capabilities for UNIX.’ In: *USENIX Security Symposium*. Vol. 46. 2010, p. 2.

[97] Witchel, Emmett, Cates, Josh and Asanović, Krste. ‘Mondrian Memory Protection’. In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 304–316.

[98] Witten, I. H. and Cleary, J. G. ‘An introduction to the architecture of the Intel iAPX 432’. In: *Software Microsystems* 2.2 (Apr. 1983), pp. 29–34.

[99] Woodruff, Jonathan D. *CHERI: A RISC capability machine for practical memory safety*. Tech. rep. UCAM-CL-TR-858. University of Cambridge, Computer Laboratory, 2014.

[100] Woodruff, Jonathan et al. ‘The CHERI Capability Model: Revisiting RISC in an Age of Risk’. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468.

[101] Yee, B. et al. ‘Native Client: A Sandbox for Portable, Untrusted x86 Native Code’. In: *2009 30th IEEE Symposium on Security and Privacy*. May 2009, pp. 79–93.

[102] Yong, Suan Hsi and Horwitz, Susan. ‘Protecting C Programs from Attacks via Invalid Pointer Dereferences’. In: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-11. New York, NY, USA: ACM, 2003, pp. 307–316.

[103] Zhou, Pin et al. ‘iWatcher: Efficient Architectural Support for Software Debugging’. In: *Proceedings of the 31st Annual International Symposium on Computer Architecture*. ISCA ’04. Washington, DC, USA: IEEE Computer Society, 2004, p. 224.