Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-based Security

Albert Kwon
kwyoung@seas.upenn.edu
Department of ESE
200 S. 33rd Street
Philadelphia, PA 19104
United States

Udit Dhawan
udit@seas.upenn.edu
Department of ESE
200 S. 33rd Street
Philadelphia, PA 19104
United States

Jonathan M. Smith
jms@cis.upenn.edu
Department of CIS
3330 Walnut Street
Philadelphia, PA 19104
United States

Thomas F. Knight, Jr.
tk@ginkgobioworks.com
Gingko Bioworks
27 Drydock Ave.
Boston, MA 02210
United States

André DeHon
andre@acm.org
Department of ESE
200 S. 33rd Street
Philadelphia, PA 19104
United States

ABSTRACT

Referencing outside the bounds of an array or buffer is a common source of bugs and security vulnerabilities in today's software. We can enforce spatial safety and eliminate these violations by inseparably associating bounds with every pointer (*fat pointer*) and checking these bounds on every memory access. By further adding hardware-managed tags to the pointer, we make them unforgeable. This, in turn, allows the pointers to be used as *capabilities* to facilitate fine-grained access control and fast security domain crossing. Dedicated checking hardware runs in parallel with the processor's normal datapath so that the checks do not slow down processor operation (0% runtime overhead). To achieve the safety of fat pointers without increasing program state, we compactly encode approximate base and bound pointers along with exact address pointers for a 46b address space into one 64-bit word with a worst-case memory overhead of 3%. We develop gate-level implementations of the logic for updating and validating these compact fat pointers and show that the hardware requirements are low and the critical paths for common operations are smaller than processor ALU operations. Specifically, we show that the fat-pointer check and update operations can run in a 4 ns clock cycle on a Virtex 6 (40nm) implementation while only using 1100 6-LUTs or about the area of a double-precision, floating-point adder.

CCS'13, November 4–8, 2013, Berlin, Germany.

ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516713.

Categories and Subject Descriptors

B.8.0 [**Hardware**]: Performance and Reliability—*General*; C.1.3 [**Computer System Organization**]: Processor Architectures—*Capability Architectures*

Keywords

Processor; security; spatial confinement; fat pointer; capabilities; memory safety

1. INTRODUCTION

Computer systems security is a major and increasing concern, not least because of the increasing dependence on such systems for activities in daily life, such as communication, shopping and banking. Today's computers reflect biases in system design and organization that are often decades out of date—based on silicon budgets where orders of magnitude fewer transistors were available than today, and based on threat models where very few machines were networked and the data they held was of limited value. For example, today's systems only provide coarse-grained separation between processes using virtual memory contexts. Switching between virtual memory contexts is expensive, encouraging the use of coarse-grained contexts with no internal separation of privileges within the contexts. A virtual memory context typically holds a very large number of objects with no boundary to separate the access of one object from another, either intentionally or accidentally. The lack of fine-grained controls on object access enables unintended use or modification of data, such as buffer overflows. Despite long familiarity with their existence and exploitation [3], buffer overflow attacks remain a persistent source of security holes.

Leaving the burden of spatial safety enforcement to individual programmers no longer appears to be a viable approach to software security. If even a single programmer leaves a potential spatial safety violation in a program, that is an open attack vector for the program.

The alternative is to design our systems to automatically prevent spatial safety errors, protecting against silent corruption of the system or violations of program semantics. This can be done at the language level with bounds checked arrays (*e.g.* Java), at the compiler or runtime level by maintaining object base and bounds information, or at the hardware level. Many recent systems explore the use of *fat pointers* that extend the pointer representation with base and bounds information so that the runtime or hardware can prevent spatial safety violations (Sec. 2.1). Software level solutions typically come with high runtime overheads (50–120%) or weak protection guarantees limiting ubiquitous adoption.

Object capability systems [14, 17, 45, 34, 28, 40] are attractive for both their potential to enable programmer control of access rights and their support for least privilege. Using hardware supported tags [36, 18, 34, 23, 20] (Sec. 3.1), we can make fat pointers unforgeable so that they can serve as a basis for the *descriptors* used in hardware-based capability systems.

In this paper we assume that the safety benefits of fat pointers [27, 33, 31] and the security benefits of capabilities [29, 42] are well established from prior work, but that their costs have typically been considered too high for ubiquitous use in the past [19, 10]. To that end, we address how they might be efficiently implemented in hardware. Specifically, we explore (1) compact representations for fat pointers that limit their impact on memory footprint (3% worst case), (2) parallel hardware support for bounds checking that guarantees there is no runtime overhead for checking, and (3) hardware enforcement and management that allow the fat pointers to serve as object capabilities. That is, we spend hardware to eliminate the overheads of spatial safety checking. We evaluate the area and delay complexity of the hardware fat-pointer operations using an FPGA implementation.

Our novel contributions include:

- Design and evaluation of a new, compact fat-pointer encoding and implementation (BIMA) that has significantly lower gate depth (small cycle time) operations than previous work (*e.g.* [7]) while simultaneously retaining: (1) compact representation in memory, (2) low memory loss due to fragmentation (<3%), and (3) precise spatial bounds detection.
- Hardware that enforces the BIMA bounds checking and update, making the fat pointers unforgeable and non-bypassable. This allows the 64b pointer to serve as an object capability, achieving significant savings over schemes that required three full 64b words to encode a capability address, base, and bound (*e.g.* [43]).
- Pipeline organization that allows the BIMA encoding to run just as fast as the baseline processor without spatial safety checking.

2. BACKGROUND

2.1 Fat Pointer

Many modern systems have explored the maintenance and checking of explicit object base and bounds in order to maintain spatial safety. That is, rather than simply representing a pointer with its address, the system includes the base and bound address in the pointer representation, for a total of three words. Since this makes the pointer larger, it is often referred to as a *fat-pointer* scheme (*e.g.* [27], [33]). By checking against the base and bound during memory operations, the system can detect any spatial safety violations and prevent them from occurring:

```
if ((ptr.A >= ptr.base)  && (ptr.A <= ptr.bound))
    perform load or store
else
    jump to error handler
```

The Secure Virtual Architecture (SVA) [11] lists fat pointers as a potential future direction for further performance improvement of OS kernel safety enforcement. Many schemes have introduced the pointers in the compiler when running on conventional hardware. These incur significant runtime and space overheads. Examples include: PAriCheck (9.5% average memory overhead and 49% runtime overhead on SPEC2000) [50], Baggy Bounds (worst-case 100% memory overhead; 15% average memory overhead and 60% runtime overhead on SPEC2000) [2], SoftBound (worst-case 200% memory overhead; 64% average memory overhead and 67% runtime overhead on SPEC and Olden benchmarks) [31], and CRED (26–130% runtime overhead) [39]. Other software schemes sacrifice guaranteed protection against all out-of-bound references in order to improve performance, such as Lightweight Bounds Checks (8.5% average memory overhead and 23% runtime overhead on SPEC2000) [21]. Furthermore, since these schemes depend on software maintenance and checking of guards, they are not suitable for use as capabilities since they only assist with voluntary confinement rather than providing mandatory access confinement.

HardBound is a hardware approach that attempts to maintain data structure layout compatibility by placing the bound information in a shadow space and reduces runtime overhead to 10–20% [15] but has a worst-case memory overhead of 200%. Moreover, the hardbound design is described only down to the micro-architectural level, providing no quantification of added gate count or necessary gate delay. Intel has recently announced a hardware-assisted approach for runtime memory bounds management [1] that appears very similar to HardBound.

Apart from the explicit fat-pointer approach for memory safety, some tagging mechanisms have been proposed that use metadata to perform spatial checks [12, 9]. The most lightweight version of these, uses a few extra bits per word to limit accidental spatial violations but not guarantee protection, while the more complete require over 100% area overhead and can have over 100% runtime overhead.

In contrast, our scheme has a worst-case 3% memory loss due to fragmentation (Sec. 4.5) and no runtime overhead (Sec. 5) while providing spatial safety semantics similar to HardBound and hardware enforcement that makes it suitable for supporting capabilities. Furthermore, we provide a gate-level design that allows us to quantify gate count and gate delays.

While most of the prior work was performed on x86 architectures, we will be using a RISC architecture as our baseline. As a crude estimate of the work performed by the fat-pointer checking with dedicated hardware, we identify the instruction sequences required to provide the same protection as our fat-pointer scheme in Tab. 1 and use instruction trace simulations for the SPEC2006 benchmarks (App. A) to calculate the impact on dynamic instruction count (Fig. 1). This is an overestimate in that a good compiler will optimize away some of these checks as redundant (*e.g.* [32]). Nonetheless, this illustrates the work performed by our fat-

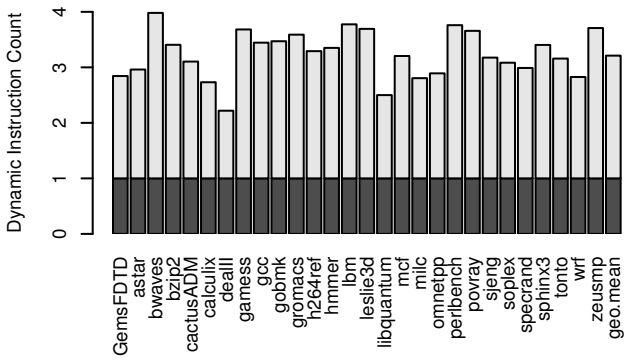


Figure 1: Dynamic instruction count for an ALPHA architecture performing spatial checks in software

pointer hardware and further motivates its need. In Fig. 1 we observe that there is an average of 220% overhead in the number of dynamic instructions executed for non-fat pointer hardware ISAs.

2.2 Object Capabilities

Typical isolation models for users of a shared computing resource include sharing disciplines for time (such as real-time or best-effort scheduling) and memory (such as read-write-execute access controls on objects in processor memory or secondary storage). Models for protection have included storage keys [25] associated with fixed size blocks of memory (*e.g.*, 2048 bytes on the System/360), programmer-defined contiguous regions of memory such as segments (*e.g.*, Burroughs descriptors [36]), and fixed-sized pages of memory comprising a protected virtual address space. Today, most virtual memory systems are implemented using several large "segments", *e.g.*, for a heap, stack and instructions, with an eye to both protection (*e.g.*, non-executable data segments) and sharing (*e.g.*, shared read-only segments). These segments are demand-paged, combining segmentation and paging in the style of Multics [35], primarily to exploit locality of reference [13]. Large segments and pages provide a coarse-grained form of hardware access control, but this coarse granularity (*e.g.*, for code, data and stack in UNIX) comes with security and performance consequences. First, there is no protection of individual objects contained within the segment, and second, the sizes of segments and pages are unrelated to the sizes of individual objects. Furthermore, switching security contexts is an expensive operation that discourages programmers from providing fine-grained separation.

Capability-based computer architectures provide hardware support for fine-grained access control. One example, the Cambridge CAP Computer [44], provided a set of "base-limit" registers that point to and delimit a segment of memory. These were augmented with bits indicating permitted accesses, *e.g.*, read, write and/or execute. A second example, HYDRA/C.mmp [46] implemented capabilities as a combination pointer and access rights, with access rights defined by a richer type system than that of the CAP Computer. A third and final example, the Intel 432 [37] provided a complex "Object Descriptor" with access rights and a structure incorporating a base address for the object and the lengths of both data and access parts. A data operand is a two-part value with an object selector (which can be interpreted as a base address) and a displacement. The displacement serves the role of a bound, giving a complex implementation of a base and bound. The basic protection mechanism for all three of these systems consists of a set of rights, and a base and bound pointer that is exercisable with appropriate rights. A major performance cost associated with these approaches is the multiple levels of indirection and table lookups for memory references; Colwell [10] provides considerable analysis of these overheads.

Other hardware systems have implemented similar constructs, *e.g.*, the 64-bit guarded pointers [8] of Carter, *et al.*, avoid translation tables by encoding permission bits and segment sizes in the pointer itself. This allows lightweight (cycle-by-cycle) context switches, allowing instructions from different threads with different security domains to be mixed in a processor pipeline and encouraging more fine-grained separation of privilege domains. Cambridge's CHERI [43] is a recent hardware capability architecture that has the goals of supporting a *hybrid capability model* as explored in Capsicum [42]. CHERI uses a 256b fat pointer as a capability including a full 64b address, 64b base, and 64b length. Our design improves on this greatly, and shows how the address, base, and bounds can be composed into a single 64b word for a 46b address space.

3. SIMPLIFIED SAFE PROCESSOR

We originally developed this fat-pointer scheme for the SAFE Processor [16]. Since the SAFE Processor includes a number of orthogonal safety innovations, we extract a reduced version of the SAFE processor, the *Simplified SAFE Processor* (`SAFElite`), to provide concrete context for the fat-pointer logic. Nonetheless, the fat-pointer implementation we describe, should be a useful component in a wide range of processor architectures.

The `SAFElite` is a simple 64b RISC processor with a four-stage pipeline as shown in Fig. 2. The `SAFElite` is a clean-slate ISA design that is not concerned with legacy binary compatibility. We use the `SAFElite` as a single-address space machine, with all applications and system services running in the same address space, exploiting the fat pointer object capabilities for privilege separation. Memory is structured into *segments*, a contiguous set of words addressable by a fat pointer. Memory in the `SAFElite` is garbage collected, thereby avoiding temporal safety hazards.

3.1 Hardware Types

The SAFE processor uses capabilities to constrain malicious activities at the hardware level. To support this, on top of a base RISC processor, we incorporate hardware mechanisms that allow us to enforce critical semantics to ensure a secured computation. First, we add a notion of native hardware types in the form of tags (*e.g.* [36, 44, 20]) to the words in the processor, such as *Integer* for integer words, *Instruction* for instructions, and *Pointers* for words that reference memory segments. These hardware types are added on top of the actual data payload. For the current discussion, let us assume that we have 8 bits allocated for the hardware types in `SAFElite` (making our memory and data path 72 bits wide). The hardware types could be larger or smaller depending on number of types desired, but even at 8 bits (256 types), `SAFElite` only incurs a total overhead of at most 16% ($\frac{72}{64} \times 1.03$; 3% worst-case memory fragmentation is explained in Sec. 4.5). Alternate implementations might prefer to avoid adding hardware type tag bits on top

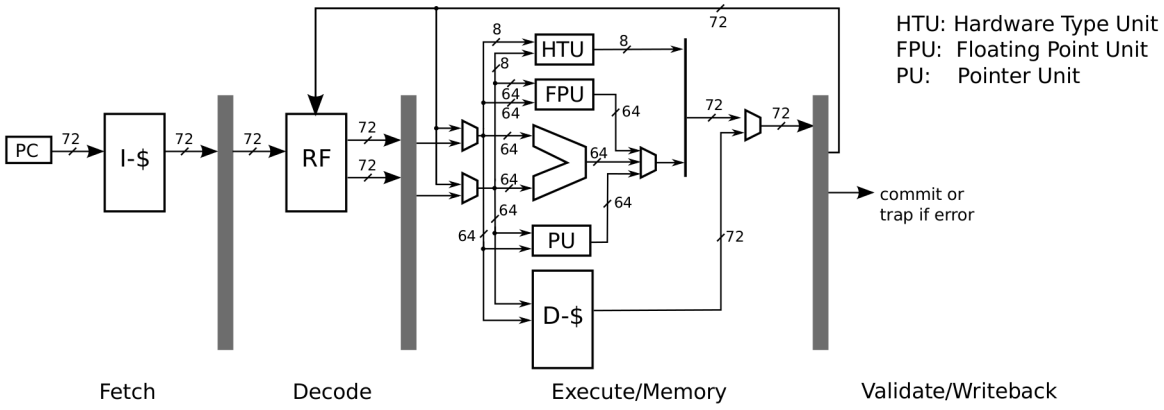


Figure 2: Simplified SAFE Processor Pipeline

of the 64b word. Some designs store metadata in a shadow space (*e.g.* [15]). Another alternative is to borrow bits from the base 64b word for hardware type.

Types limit how data can be used. Using these hardware types, the `SAFElite` avoids several attacks such as malicious code injection attacks that try to use input data that would be typed as an *Integer* as processor instructions. On `SAFElite` only a word with *Instruction* hardware type will be executed as an instruction. To support this, we add a *Hardware Type Unit* (HTU) to the `SAFElite` (see Fig. 2) to validate its entire operation with respect to the instruction and input operands (such as only an *Integer* can be added to an *Integer*). The Hardware Type Unit also assigns the hardware type to the result of a computation, if any.

Although 8b of hardware type allows up to 256 unique hardware types, for this paper we concentrate on the following hardware types:

- **Pointer:** hardware type on any word that is a well-formed and valid pointer to a segment of memory; pointers can be further subdivided by access, including Read-Only and Execute-Only Pointers.
- **Integer:** hardware type on a word that is used as a 64b bit vector
- **Out-of-Bounds-Pointer:** hardware type on a word that was previously a pointer, but has now preceded its base or exceeded its bound.
- **Out-of-Bounds-Memory-Location:** hardware type on the memory locations that seem to be referenced by a valid pointer but are not within bounds, as these lie beyond the precise bound of the segment (see Sec. 4.4)
- **Error:** SAFE and `SAFElite` allow us to invalidate an operation if it violates the intended semantics (*e.g.* [24]); in that case, we assign this hardware type on the operation's result. In general we may have more errors than the two Out-of-Bounds cases identified above, but we will not differentiate them further in this paper.
- **Other:** the set of hardware types on words that are not a pointer, integer, or error. We will not differentiate these further in this paper.

The hardware types allow the garbage collector to distinguish pointers from non-pointers, thereby facilitating garbage collection (GC). One hardware type is used to represent GC forwarding pointers.

In a conventional processor all the pointer arithmetic is achieved using regular arithmetic instructions. For example, the `addqi` instructions used in the ALPHA architecture for adding integers are also used to perform pointer arithmetic This poses a serious vulnerability that can be exploited to potentially create a memory reference to any location (*e.g.*, [38]). However, this is not possible in the `SAFElite` since the Hardware Type Unit does not allow typed pointer data to be offset by an integer using a regular `addm` (integer add modulo 2^{64}) instruction. Furthermore, the result of an `addm` instruction is an *Integer* that cannot be used as a *Pointer* for dereferencing memory. In order to perform pointer arithmetic in the `SAFElite`, we add a specialized instruction `addp` (add to pointer) for incrementing or decrementing a word with a pointer hardware type. Consequently, we add a new functional unit called the *Pointer Unit* (PU) as shown in Fig. 2 that is responsible for all the pointer-related arithmetic operations. Instructions used to access memory, namely `lw` (load word from memory) and `sw` (store word to memory), follow the same semantics as in a common processor, but instead require a pointer type argument (see Sec. 4.8).

3.2 Privileged Memory Management System

Control over creation and allocation is necessary to guarantee *unforgeability* of the pointers if they are to be used as capabilities [46]. We assume that the Memory Manager (MM) is a separate, privileged software subsystem that can create fat pointers. Only the MM has the privileges to create a new fat pointer. Ordinary code can call into the MM's allocation routine to obtain a new pointer. The full SAFE Processor supports calls into privileged domains (*gates*) that are as inexpensive as ordinary procedure calls [16].

3.3 Stack Protection

To get full advantage of the fat pointer protection, it would be valuable to use separate segments for each stack frame. For non-garbage-collected systems or hybrid systems where the compiler can stack-manage call frames, this can be supported with a special `pushframe` instruction[1] at the `SAFElite`-level to create subframes from a typed stack pointer and a special `popframe` instruction to return them. In hybrid cases, the garbage collector will need to distinguish these stack subframes and treat them differently. In

[1]This is similar to the `alloca` primitive in conventional systems that is used allocate memory space on the stack rather than the heap.

the full SAFE Processor, the call stack is not directly visible to user code and is used only for procedure control.

3.4 Compatibility

While our primary design is not concerned with legacy compatibility, our hardware could be added for capability support alongside legacy code as illustrated in the CHERI hybrid capability processor [43]. CHERI uses capabilities for fine-grained separation within a virtual-memory context. Capability-oblivious legacy code can call capability-aware subroutines, and capability-aware code can use capabilities to sandbox capability-oblivious software components. Types can effectively capture the operations allowed on a capability, so our hardware-typed pointer can provide the same functionality as the full 256b CHERI capability.

3.5 FPGA Implementation

To validate the operation and characterize the performance of the `SAFElite`, we prototyped the processor on an ML605 FPGA development board [47] from Xilinx with a Virtex 6 (xc6vlx240t-2 device) [48]. A 64-bit addition on this device takes close to 2 ns. However, with operations such as variable shifts, the Integer ALU can operate only at a latency of 4 ns. A moderate sized direct-mapped cache built using Block RAMs (discrete SRAM blocks on this FPGA device) can also be accessed with a delay under 2.5 ns. Therefore, it is necessary that all functional units, including the Pointer Unit, finish their operation within a 4 ns envelope in order to avoid increasing the processor cycle time.

4. COMPACT FAT POINTER ENCODING

In this section, we derive our compact fat-pointer encoding (Secs. 4.1–4.2), address issues with approximate sizing and out-of-bounds pointers (Secs. 4.3–4.4 and 4.7), and characterize the worst-case fragmentation effects (Sec. 4.5). We roundup the operations on the fat pointers (Sec. 4.8). Finally, we show the datapaths to support the fat-pointer operations and quantify their area and delay (Secs. 4.9–4.10).

4.1 Aligned Encoding

Keeping track of base and bound potentially requires three pointers. To get a more compact fat-pointer representation, let us first assume that the size of the segment the pointer is referencing is a power of two (i.e., 2^B for some B). Furthermore, assume that the pointer is aligned on the same power of two boundary. Then the base of the pointer can be determined by replacing B bits in the LSB with 0's:

$$\texttt{base} = \texttt{A} - (\texttt{A} \,\&\, ((1 << \texttt{B}) - 1)) \tag{1}$$

Similarly, the bound can be represented by replacing the B LSBs with 1's. With the above assumptions, we can represent the pointer and its base and bound with an overhead of only $|B|$ bits, which is, at most, the log of the size of the address space. This is the scheme used for guarded pointers by Carter [8] and for the Baggy Bounds software scheme [2].

4.2 Floating-Point Size

This simple encoding could, however, result in large memory overhead through fragmentation: objects of size slightly larger than a power of two will waste nearly half of the allocated space. For example, for an object of size 33, we must allocate 64 words. To address this issue, instead of having

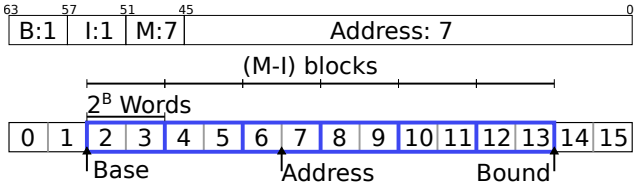


Figure 3: A typical pointer in the BIMA encoding scheme

the pointer as one large block, we can view it as a collection of several smaller blocks. If we used five bits for the number of blocks, we can allocate 17 blocks of size two for an object of size 33, forcing us to only allocate a segment one word larger than the desired allocation.

The basic floating-point size approach was introduced by Aries [7] and is analogous to floating-point numbers. In both cases, we represent a range of values much larger than the number of bits used for encoding by separating precision (mantissa) from the exponent (B in our case). This results in an exact address representation, an exact base representation that has alignment restrictions, and an approximate bound representation that may force us to conservatively overestimate (round up) the segment size.

Given the alignment restrictions, we can reconstruct the base and bound pointers with just a few additional bits beyond the current pointer address. Furthermore, the reconstruction can be performed with simple logic that places these additional bits into the appropriate bit positions (deposits) on copies of the address. To encode both base and bounds, we use a minimum field, I, for the base, and maximum field, M, for the bound that specify the $|I| = |M|$ bits above the bottom B bits of the base and bound address. When we create a pointer, we deposit $|I|$ bits above the bottom B bits of the base to I, and do the same with the bound to M. We can almost recover the base and bound by simply shifting I and M by B and replacing the bottom |I|+|B| bits in the address. We call this encoding scheme BIMA since its key fields are the block size exponent, B, the minimum bound, I, the maximum bound, M, and the address, A. Fig. 3 shows a typical pointer in the BIMA encoding scheme.

$$\begin{aligned}
\texttt{carry} &= 1 << (\texttt{B} + |\texttt{I}|) \\
\texttt{Atop} &= \left(\texttt{A} \,\&\, \overline{(\texttt{carry} - 1)}\right) && (2) \\
\texttt{Mshift} &= \texttt{M} << \texttt{B} \\
\texttt{Ishift} &= \texttt{I} << \texttt{B} \\
\texttt{D}_{\text{under}} &= (\texttt{A} >> \texttt{B})[5:0] < \texttt{I} \;? \\
&\quad\; (\texttt{carry} \mid \texttt{Atop}) - \texttt{Ishift} : \\
&\quad\; \texttt{Atop} - \texttt{Ishift} && (3) \\
\texttt{D}_{\text{over}} &= (\texttt{A} >> \texttt{B})[5:0] > \texttt{M} \;? \\
&\quad\; (\texttt{carry} \mid \texttt{Mshift}) - \texttt{Atop} : \\
&\quad\; \texttt{Mshift} - \texttt{Atop} && (4)
\end{aligned}$$

It is not as simple as a deposit because the bits **above** the I or M bits in the word may need to be incremented. Nonetheless, this is easy to detect and accommodate. There are two possible cases for this scheme: the pointer has not incurred any carries in the A field beyond $|I| + B$ bits or it has. When there have been no carries beyond the bottom $|I| + B$ bits, the relevant bits that determined I and M have

not changed. Therefore, the $|I|$ bits above the B bits will be greater than or equal to I and less than or equal to M. In this case we can compute the distance to underflow and overflow, D_{under} and D_{over}, by doing a $|I| + B$ subtraction using the bottom bits of A and the base and bound computed from simple shifts. When there is a carry, however, I could be larger than the corresponding bits in A and vice-versa for M. We can resolve this issue by adding one bit to the MSB of the smaller quantity (*i.e.*, add $2^{|I|+B}$). Eqs. 3 and 4 show the computation required.

To fit the encoding into one 64-bit word, we must use some of the bits to encode B, I, and M. For example, if we allocate 6 bits to each, that leaves us with 46b to specify the address. Since we use word addressing of 64b words, this is comparable (2^{49}=512TB) to the 40b byte addresses [26] (1TB) or 48b byte addresses (256TB) [4] currently supported by x86-64 architectures.

4.3 Out-of-Bounds

One consequence of the compact representation is that we cannot represent a pointer that points to an address that is not bounded by the base and bounds. Consequently, whenever a pointer is computed that violates its bounds, we represent it with a different hardware type, the Out-of-Bounds-Pointer. Producing an out-of-bounds pointer, itself, is not an error. It is common to increment a pointer until it exceeds the bound. The error only occurs when we attempt to use the Out-of-Bounds-Pointer as the address in a load or store operation. As a result, there is no bounds check at the time of the load, only a Hardware Type Unit check that the type of the word used as an address is still a valid, in-bounds pointer. This means that `addp` effectively both computes the new pointer and checks bounds. Note here that once a pointer goes out of bounds, it is permanently marked as an error. Any further updates on the pointer will not change the type back to Pointer.

4.4 Option to Enforce Exact Bounds

The floating-point size representation results in an approximate bound, meaning that there can be size mismatches between the pointer and the object. The segment thus will have some words that extend beyond the end of the object that will be unused and accessing these words will not be considered a violation on the basis of the compact fat-pointer bounds alone. Nonetheless, these extra words are not part of any other segment, so writes through pointers will never write into a different segment.

In systems, such as `SAFElite`, where type rules and forwarding pointers demand that we read every word before writing it, we can enforce exact bounds for all pointers by filling the extra words with type Out-of-Bounds-Memory-Location. When the processor performs a memory operation with a pointer within the fat-pointer bounds but reads an Out-of-Bounds-Memory-Location type, the hardware will flag the same error as an out-of-bounds pointer. In both of the cases of spatial violation, the processor will trap to a software handler that will resolve the error. This abstraction allows user-level applications to treat both kinds of out-of-bounds references in the same way.

4.5 Fragmentation

The BIMA encoding provides a better fit between the object and the segment than the exponential alignment and sizing (Sec. 4.1), but the approximate nature of the floating-point size representation also forces us to over-allocate memory to objects as noted above. Any object of size larger than $2^{|I|}$ may result in allocation of a segment larger than the object. Segments of odd size larger than $2^{|I|}$, for instance, will be collections of blocks of even size, and thus there would be at least one word that is wasted.

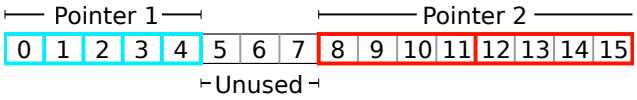


Figure 4: Example of external fragmentation

To compute the impact of the *internal fragmentation*, we note that maximizing $(M - I)$ and minimizing B will result in smaller internal fragmentation. The pointer will never be more than one 2^B block larger than the object. Thus, if we ensurethat the allocator will always use the largest $(M - I)$ and smallest B possible, the memory loss from *internal fragmentation* is less than $1/2^{|I|}$.

The encoding also specifies that blocks of size 2^B must be aligned, and the alignment could cause *external fragmentation*. In particular, with the simple allocation scheme that assigns blocks of memory in the order of the requests, we could have a large object following a small object. In this case, the smaller object could be using some words in the segment that would have been in the first block of the larger object. This forces the larger object to be aligned on the next boundary of 2^B, thus wasting nearly a whole block. For example, in Fig. 4, Pointer 1 is of size 5, and has $B = 0$. For the sake of illustration, assume $|I| = 2$. Pointer 2 is of size 8, and $B = 2$. Because Pointer 1 uses address 4, Pointer 2 must be aligned on the next 2^2 boundary, resulting in addresses 5, 6, and 7 being unused. Nevertheless, this can only waste at most one 2^B block. External fragmentation can therefore result in memory loss of at most $1/2^{|I|}$ of the allocated memory. Internal and external fragmentation together waste less than $1/2^{|I|-1}$ of memory. *If we choose $|I| = 6$, the maximum fraction of memory lost to fragmentation is 3%.* The external fragmentation could be reduced with a more sophisticated allocator that kept track of fragments and avoided placing small blocks on larger block alignment boundaries whenever possible. Nonetheless, the observation above shows that this is not absolutely necessary: such sophisticated alignment would *at most* eliminate worst-case external fragmentation and thereby only cut total fragmentation in half. Furthermore, note that the length of objects whose size is less than $2^{|I|}$ are always represented exactly; therefore, *for small objects there is* **no** *loss due to fragmentation.*

4.6 Decoded Bounds

The BIMA encoding demands that we decode the bounds in order to perform checks. We can avoid the delay associated with decoding bounds by storing decoded bounds (D_{under}, D_{over}) along with the pointer in the register file. Whenever the processor loads a pointer, the processor also computes the underflow and overflow distances in preparation for any necessary pointer computation. This does result in a larger register file from storing the distance values. However, as we will see in Sec. 4.9, it increases performance significantly with low overhead in area. Furthermore, since

the decoded pointers are only stored in the register file, there is no overhead in the memory.

4.7 C Compatibility

For C compatibility, it is necessary to represent Out-of-Bounds pointers that are one element past the end of the array. In cases where the pointer size is larger than the allocated object, as discussed in Sec. 4.4, our scheme already accommodates the C-style Out-of-Bounds pointers. As long as no reference is made to the Out-of-Bounds-Memory-Location, no error is flagged. If the pointer is subsequently modified so that it comes back in bounds, it can be used. For cases where the pointer size is exact, our scheme as described would not allow the pointer to be advanced to one past the end of the array and then recovered as a pointer. The simplest way to support this feature would be to allocate a pointer that is at least one word longer and fill the memory location past the real end of the object with Out-of-Bounds-Memory-Location as described in the previous section. Note that this, retains the same worst-case 3% fragmentation overhead we establish in Sec. 4.5 beyond the object-size+1 allocation made.

A slight modification to the scheme would avoid paying for any extra memory locations. With our D_{under}, D_{over} scheme (Sec. 4.6) we could allow D_{over} to become negative and then check the sign of D_{over} on a memory operation. As long as the pointer is returned to in-bounds before a memory reference is made, no error is flagged. To handle the case of a one-element-over Out-of-Bounds being written to memory, we could add a distinguished C-Out-of-Bounds-by-One pointer type and convert to this pointer type at the point of writeback to memory. This feature is not included in the detailed evaluation that follows; we believe it would add area but not impact cycle time.

With suitable compiler and linker support, Nagarakatte reports that bounds can be applied to most legacy C programs without source code changes [30].

4.8 Operation

We now roundup the pointer-related operations that must be performed with the BIMA encoding.

4.8.1 newp

The MM subsystem uses a privileged instruction `newp` to create new fat pointers. Typically, these would be allocated out of an available block of memory (*e.g.* NewSpace or CopySpace in a Garbage Collection scheme). `newp` can be used to decompose a large, unallocated segment into a collection of allocated segments.

4.8.2 offsetp

The `offsetp` instruction returns the offset of the pointer from its base. This is simply the D_{under} value in the register file, so requires no computation. Iterators can use the offset to identify the end of an array when the approximate bound (Sec. 4.2) makes the segment larger than the live data.

4.8.3 addp

Pointer arithmetic in `SAFElite` is done via an instruction called `addp`. The `addp` instruction is a three-operand (two sources, one destination) instruction where one source is the pointer we are changing, and the other is the offset. In BIMA, when we read the pointer to add the offset, we read the D_{under} and D_{over} associated with the pointer as well. We then add the offset to D_{under} to indicate that distance to underflow has changed by the offset. Similarly, we subtract the offset from D_{over}. The pointer is within bounds as long as both distances are non-negative since neither underflow or overflow has happened. If either distance is negative, we replace the hardware type of the pointer with Out-of-Bounds-Pointer.

4.8.4 sw

`SAFElite` has a two-operand store instruction, `sw`. Like many RISC architectures, there is a direct path from the register file to the memory. In the `SAFElite`, however, we check the type of the pointer in parallel with the memory access (see Fig. 2). As described before, if the address field is out of bounds, then the type of the pointer will be changed to an Out-of-Bounds-Pointer. Therefore if the word we are using as an address to store is an Out-of-Bounds-Pointer, we flag an out-of-bounds error. Otherwise, the operation is carried out as with any other processors.

When we are storing a pointer to memory, note the processor can just store the 64b BIMA pointer and ignore the decoded bounds D_{under} and D_{over}. They are redundant information and can be recomputed when loading the pointer back from the memory using the `lw` operation.

4.8.5 lw

`SAFElite` also has a load operation `lw` that is a one-source, one-destination instruction. On a `lw`, there are potentially two pointers that we need to check: the pointer we are using as the source, and the loaded value from memory that could be a pointer. The fat-pointer computation for the first pointer behaves the same as the store case: we check if the pointer is out of bounds, indicated by the Out-of-Bounds-Pointer type on the pointer.

The fat-pointer decode computation is unique to `lw`. The loaded value could be a pointer as well, and for future bounds checks from the loaded pointer, we need to determine its D_{under} and D_{over}. Therefore, we must decode the pointer before it is stored in the register file (See Fig. 6a).

4.8.6 Conventional Processor Equivalent

For the sake of comparison, Tab. 1 shows roughly the instructions required to simulate these operations on a RISC processor with no hardware support for fat pointers. Tab. 1 omits `offsetp` since it is not needed when using full 64b pointers to represent base and bound and `newp` since it is dominated by other instructions for allocation. Compiler analysis and support will often be able to remove some of the bounds checks and data movement (*e.g.* [32]).

4.9 Implementation

To evaluate the performance of the encoding scheme, we implemented a Pointer Unit (PU) for decoding and updating for both the BIMA and Aries fat pointers. The PU was designed[2] using Bluespec SystemVerilog [6] and implemented on a 40nm Xilinx Virtex 6 FPGA (xcv6vlx240t-2) [48].

4.9.1 Aries

While the BIMA scheme shares the basic strategy with Aries, our encoding is different in specific details that reduce the latency of the key operations. Instead of using a lower

[2]source code available at `http://ic.ese.upenn.edu/distributions/fatptr_ccs2013`

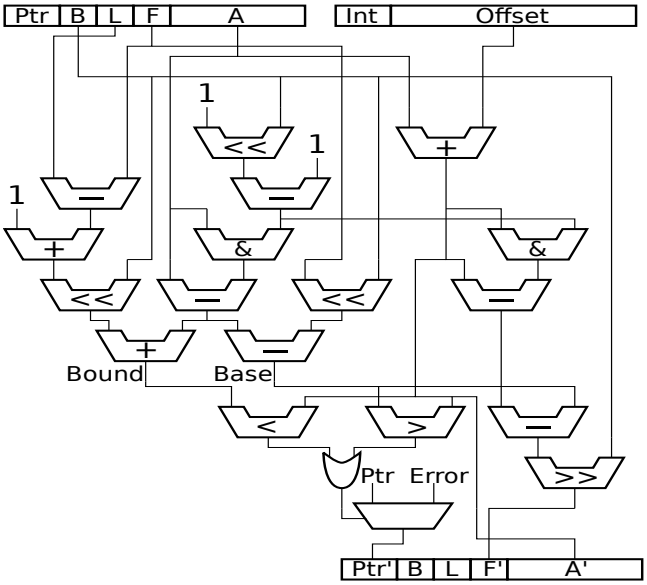


Figure 5: Data path for the Aries Encoding (`addp`)

(I) and upper (M) bound, Aries uses a length (L) and finger (F) field that effectively capture the same information. In both the BIMA and Aries cases the A field always points exactly to the current word. Since Aries does not have an I and M field, the Aries finger field records the offset of the current block form the base block in the record to allow bounds circuitry to calculate the base and bound. However, this has two problems: (1) it demands a more complicated decoding than BIMA, and (2) it demands that the finger field potentially be updated on every pointer increment. For reference, the data path for the Aries Pointer Unit is shown in Fig. 5.

4.9.2 BIMA Decode

The decode data path for the BIMA scheme is shown in Fig. 6a. Compared to the Aries encoding data path, the depth is smaller, and there are fewer variable shifts, resulting in significantly shorter delay. For $|B| = |L| = |I| = 6$, the delay is less than 4.8 ns up to 64b addresses. The delay comparison with Aries is shown in Fig. 9. This is over a 40% reduction in delay from Aries encoding on average. The delay and area of the BIMA decode is summarized in Figs. 7a and 8. Nonetheless, at 4.2 ns, the 46b decode is slightly longer than our target cycle time of 4 ns. The decoded bounds (Sec. 4.6) avoid the need to perform this decoding for any operation other than `lw`. In Sec. 5, we show that we can split the decode needed for `lw` across pipeline stages to prevent this operation from limiting the frequency of the processor or creating any new stall conditions.

4.9.3 BIMA Update

The update operation in the BIMA scheme is even simpler than the decode operation. As shown in Fig. 6b, the BIMA update data path consist only of additions, comparison, and a multiplexer. The delay of update is independent of size of I and B and is less than 4 ns for any number of address bits below 64 as shown in Fig. 7b, meaning the update operation will not degrade processor clock frequency. At $|A|$=46, BIMA update delay is less than half the Aries update delay (Fig. 9). Aries must check and update the F field on ever update operation. We note here that because decode and update are part of different instructions in BIMA, the delay of the two do not add; the delay of decode is relevant only for `lw`, while delay of update is only relevant for `addp`. Fig. 7b also summarizes the area for the update operation.

4.9.4 Area

The total area required by the BIMA scheme is the sum of the area of decode and update; for a 64b fat pointer with $|A|$=46, the pointer unit uses 1114 LUTs.[3] This is larger than the 956 LUTs required by the Aries encoding. Nonetheless, the total area is still comparable to the floating point adder used in `SAFElite`, which takes 940 LUTs [49].

4.9.5 Other Implementation Technologies

We include the FPGA delay and area comparison to provide a concrete point of comparison for the complexity of the key operations in the fat-pointer encoding schemes. Nonetheless, similar effects and benefits would occur in ASIC or custom implementation technology. Simply looking at Figs. 5, 6a, and 6b, we can see that the depth of operations is lower for the BIMA operations than the Aries implementation. This makes it clear the delay would be smaller in any implementation. Note that the slowest operation in each datapath is the variable shift. The Aries datapath requires two variable shifts while the BIMA decode requires only one and the BIMA update requires none. Furthermore, note that the slowest operation in the ALU is the variable shift. This alone is enough to explain (1) why the BIMA update, which requires no variable shifts, is faster than the ALU, (2) why the BIMA decode is only slightly slower than the ALU, and (3) why the Aries decode with two variable shifts is over twice the delay of the ALU. Consequently, these general timing relations will hold for custom implementations as well.

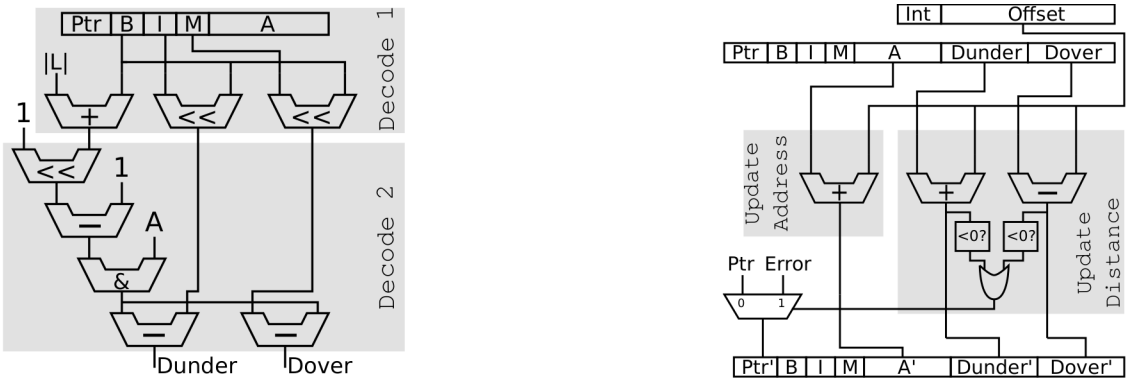
4.10 Decoded Bounds Implementation

In the baseline case where we do not add any hardware type metadata to the words, the register file width is 64b. Then, we can fit an entire 32-entry register file in two Virtex-6 BRAMs[4] organized as 512×64b, dual-ported memories. The base `SAFElite` uses two BRAMs to support one write and two read ports on 64b words. However, when we add an 8b hardware type metadata, the register file width is now 72b, and this forces us to use two BRAMs per read port or a total of 4 BRAMs, even though we are not using the entire width offered by two BRAMs. On top of this, when we add the decoded lower and upper bounds, we further increase the width of our register file by twice the width of our address size. Therefore, when the address size is 46b, the register file is $(72 + 2 \times 46 =)$ 164b wide. The register file now requires 3 BRAMs per read port for a total of 6 BRAMs. If this were a full custom implementation, we would simply expand the 64b register file width to 164b, or a 156% overhead.

5. PIPELINING AND BYPASSING

The `SAFElite` has four pipeline stages as shown in Fig. 2. Without the fat-pointer computation, the processor runs at 4 ns clock cycle with full bypass of results to prevent stalls

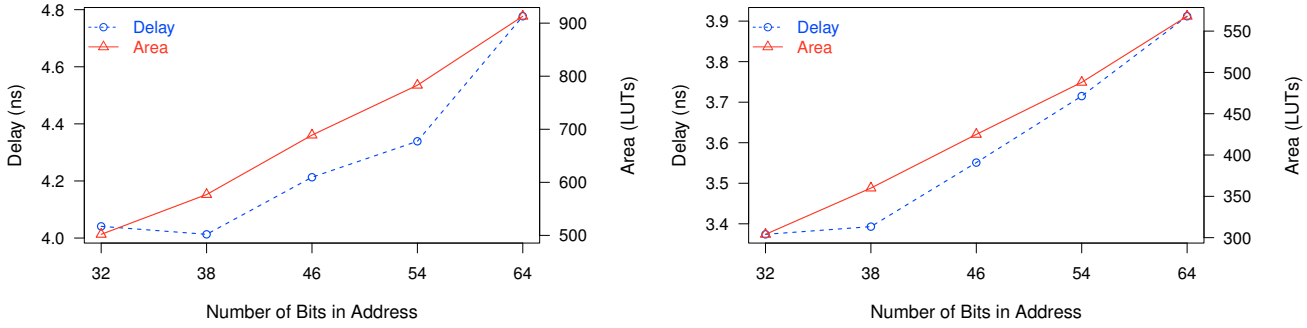
[3]The Virtex 6 FPGA employs 6-input Look-Up Tables for logic. We refer to them simply as LUTs henceforth.
[4]BRAM=Block RAM, SRAM blocks in a Virtex-6 FPGA



(a) BIMA Decode (`lw`)

(b) BIMA Update (`addp`)

Figure 6: Data path for BIMA scheme



(a) BIMA Decode (`lw`)

(b) BIMA Update (`addp`)

Figure 7: Area and Delay for BIMA scheme, $|B| = |I| = 6$

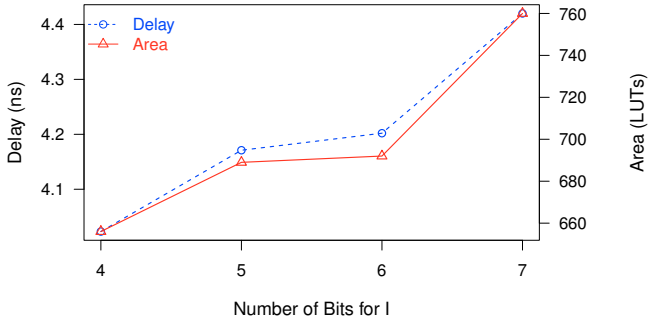


Figure 8: Area and delay for the BIMA decode (`lw`) for varying $|I|$ with $|B| = 6$, $|A| = 46$

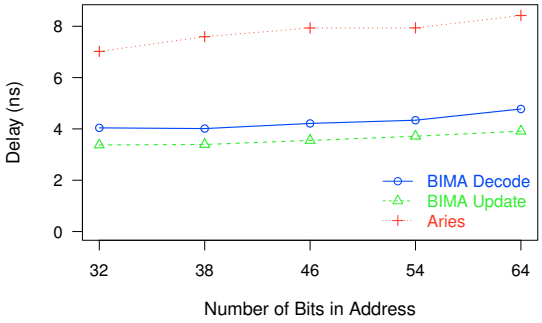


Figure 9: Delay comparison between Aries and BIMA encodings with $|I| = |L|$ with varying $|A|$

on `addp`, `lw`, and `sw` operations when the memory references hit in the L1 cache. Adding the 8 ns Aries `addp` operation to the pipeline could, however, dramatically degrade the processor's operating frequency.

The BIMA encoding also changes the pipeline due to the decoding of D_{under} and D_{over} after a load. The BIMA decode takes more than 4 ns for addresses using more than 46b, potentially increasing the cycle time. Also, if we decode the two distances in the last pipeline stage, then an `addp` operation whose argument is the pointer loaded from memory in an immediately preceding `lw` instruction (an `lw-addp` pair) requires a stall cycle: the D_{under} and D_{over} is needed for the `addp`, so we need to finish decoding the distances before `addp` can execute.

To mitigate this problem, we first split the decode data path into two stages—`Decode 1` and `Decode 2` in Fig. 6a. In our implementation, we can fetch from the data cache in less than 2.5 ns, so we use the remaining time in the Execute stage to execute the first part of decode (`Decode 1`). Then we finish the computation of D_{under} and D_{over} (`Decode 2`) in the Validate/Writeback stage. With this change, we avoid increasing the clock cycle. To remove the stall for `lw-addp`, we split the update (See Fig. 6b) into two stages as well: we keep the update address circuit (`Update Address`) in the

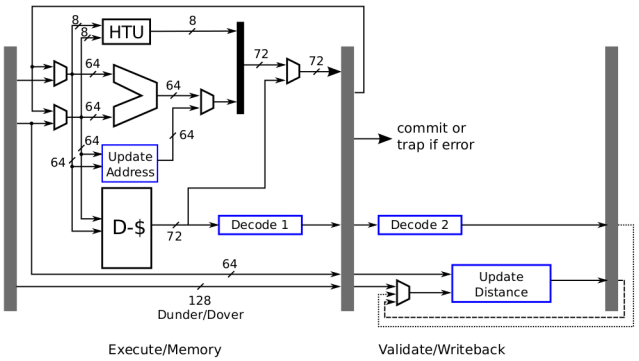


Figure 10: Last two stages of pipeline for BIMA encoding

Execute/Memory stage, and we move the update distance circuit (`Update Distance`) to the Validate/Writeback stage. Fig. 10 shows the resulting pipeline.

First we note that this change does not affect bypasses or stalls for any non-pointer computations. The number of pipeline stages stays the same, and any other unit remains unchanged. Let us now consider an `lw-addp` pair in this pipeline. The `addp` can happen immediately as the value (pointer) from memory can be bypassed in one cycle (dotted bypass in Fig. 10). While `addp` is computing the new address, `Decode 2` in the Validate/Writeback will have finished decoding the distances. Then the distances needed for `addp` will be available before the Validate/Writeback stage, and thus it will need no stall cycles. Similarly, an `addp-addp` pair does not cause any stalls either because we can bypass the D_{under} and D_{over} in the same fashion (dashed bypass in Fig. 10).

To avoid any stalls for an `addp-lw` pair, we use speculative execution. Since the target address is available by the end of the Execute stage, we bypass the updated address and load the value assuming the pointer is in bounds. When the `addp` distance update completes and the pointer used was determined to be out-of-bounds, we kill the load result, flush the pipeline, and trap to the error handler.

As a result, we maintain the same clock rate and stall conditions as the baseline processor without the Pointer Unit. That is, the BIMA encoding runs just as fast as the baseline processor without spatial safety checking (0% runtime overhead).

6. CONCLUSIONS

We have shown how it is possible to support fully mediated, fine-grained spatial safety without sacrificing performance. An appropriately low-fat pointer encoding keeps the memory impact small (64b pointers for a 46b address space) while simultaneously guaranteeing small and fast hardware decoding and small (<3%) memory fragmentation overhead. Even with an additional 8b tag on every word, the total memory overhead is under 16%. Suitable pipelining guarantees the extra work for decoding the low-fat pointer encoding does not create new stall cycles for the processor pipeline. Pointer updates are no more complex than additions, guaranteeing they do not degrade the processor clock cycle. Since software presents a large attack surface area (millions of lines of code in modern applications and kernels) with demonstrated high rates of vulnerability, there is an advantage to providing safety guarantees at the hardware level where the interface is narrower. Furthermore, as computers protect increasingly valuable assets, spending this modest number of relatively inexpensive transistors (comparable to a double-precision, floating-point adder) to increase safety and security seems a prudent investment.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. Ben Karel, Cătălin Hriţcu, and Greg Morrisett provided valuable discussion and feedback on this paper.

8. REFERENCES

[1] Introduction to Intel Memory Protection extensions. `http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`. Accessed: 2013-08-01.

[2] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium*, pages 51–66, 2009.

[3] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), November 1996.

[4] AMD Corporation. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*, revision 3.22 edition, September 2012.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[6] Bluespec, Inc. Bluespec SystemVerilog.

[7] J. Brown, J. Grossman, A. Huang, and T. F. Knight, Jr. A capability representation with embedded address and nearly-exact object bounds. Technical Report 5, MIT AI Lab, April 2000. Aries Project.

[8] N. P. Carter, S. W. Keckler, and W. J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, ASPLOS-VI, pages 319–327, 1994.

[9] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 63–65. ACM, 2006.

[10] R. P. Colwell, E. F. Gehringer, and E. D. Jensen. Performance effects of architectural complexity in the Intel 432. *ACM Trans. Comput. Syst.*, 6:296–339, August 1988.

[11] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution

environment for commodity operating systems. In *Proceedings of the Symposium on Operating Systems Principles*, October 2007.

[12] D. Y. Deng and G. E. Suh. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE Computer Society, 2012.

[13] P. J. Denning and S. C. Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, March 1972.

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

[15] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 103–114, 2008.

[16] U. Dhawan, A. Kwon, E. Kadric, C. Hriţcu, B. C. Pierce, J. M. Smith, A. DeHon, G. Malecha, G. Morrisett, T. F. Knight, Jr., A. Sutherland, T. Hawkins, A. Zyxnfryx, D. Wittenberg, P. Trei, S. Ray, and G. Sullivan. Hardware support for safety interlocks and introspection. In *SASO Workshop on Adaptive Host and Network Security*, Sept. 2012.

[17] R. S. Fabry. Capability-based Addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[18] E. A. Feustel. On the advantages of tagged architecture. *IEEE Transactions on Computers*, C-22(7):644–656, July 1973.

[19] E. F. Gehringer and J. L. Keedy. Tagged architecture: How compelling are its advantages? In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 162–170, 1985.

[20] R. Greenblatt, T. Knight, Jr., J. Holloway, D. Moon, and D. Weinreb. The LISP machine. In *Interactive Programming Environments*. McGraw-Hill, 1984.

[21] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.

[22] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

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

[24] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *34th IEEE Symposium on Security and Privacy*, pages 3–17. IEEE Computer Society Press, May 2013.

[25] IBM. *IBM System/360 Principles of Operation*. 1968.

[26] Intel Corporation. *Intel64 and IA-32 Architectures Software Developer's Manual*, August 2012.

[27] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 275–288, 2002.

[28] D. Johnson. The Intel 432: A VLSI Architecture for Fault-Tolerant Computer Systems. *Computer*, 17:40–48, August 1984.

[29] H. M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

[30] S. Nagarakatte. *Practical Low-overhead Enforcement of Memory Safety for C Programs*. PhD thesis, University of Pennsylvania, 2012.

[31] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[32] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: Compiler enforced temporal safety for C. In *Proceedings of the International Symposium on Memory Management*, pages 31–40, 2010.

[33] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[34] R. M. Needham and R. D. H. Walker. The Cambridge CAP computer and its protection system. In *Proceedings of the Symposium on Operating Systems Principles*, pages 1–10, Nov. 1977.

[35] E. I. Organick. *The MULTICS System: An Examination of Its Structure*. MIT Press, 1972.

[36] E. I. Organick. *Computer System Organization: The B5700/B6700 Series*. Academic Press, 1973.

[37] E. I. Organick. *A Programmer's View of the Intel 432 System*. McGraw-Hill, 1983.

[38] A. T. Phillips and J. S. Tan. Exploring security vulnerabilities by exploiting buffer overflow using the MIPS ISA. In *Proceedings of the SIGCSE technical symposium on Computer science education*, pages 172–176, New York, NY, USA, 2003. ACM.

[39] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[40] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: a fast capability system. In *Proceedings of the Symposium on Operating Systems Principles*, pages 170–185. ACM, 1999.

[41] R. L. Sites. Alpha AXP Architecture. *Digital Technical Journal*, 4(4):1–17, 1992. Special Issue.

[42] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, Washington, DC, August 2010.

[43] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. W. Moore, S. J. Murdoch, P. Paeps, M. Roe, and H. Saidi. CHERI: a research platform deconflating hardware virtualization and protection. In *Proc. RESoLVE*, March 2012.

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

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

[46] W. A. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[47] Xilinx, Inc. Virtex-6 FPGA ML605 Evaluation Kit.

[48] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Virtex-6 FPGA Data Sheet: DC and Switching Characteristics*, September 2011. DS512.

[49] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *LogiCORE IP Floating-Point Operator v6.0*, January 2012.

[50] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. Paricheck: an efficient pointer arithmetic checker for C programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 145–156, 2010.

APPENDIX

A. DYNAMIC INSTRUCTION IMPACT

Our main focus is to show the benefit of native hardware support to enforce spatial memory safety on every instruction. If we were to enforce the same level of checking using a conventional processor, we would need to replace a number of primitive operations with instruction sequences like those shown in Tab. 1 where in the second column we show the additional operations needed for spatial checking. In reality the compiler would add these extra instructions in the binary. However, we use instruction-trace based simulations to get a crude estimate of the impact on dynamic instruction count for such an enforcement without modifying any compiler. Our estimates illustrate how much work our investment in parallel checking hardware and compact encoding is effectively doing. These estimates are necessarily crude since the instruction sequences can be subverted below the level of the compiler, and do not account for optimizations a compiler could do to eliminate redundant checks. Also, we assume a single-issue in-order pipeline, whereas, on a superscalar processor, some of the extra instructions would fit in otherwise unused issue slots. For example, SoftBound reports only a 67% runtime overhead on 133% instruction count overhead [31].

A.1 Methodology

We evaluate the impact of the additional instructions for performing spatial checks on memory, on the programs from the SPEC2006 Benchmark Suite [22]. Instruction traces are produced using the `gem5` [5] environment simulating a 64-bit ALPHA ISA [41]. Since the ALPHA ISA used does not type pointers differently from integers, we must determine which values are actually pointers and require additional pointer operations. To do this, we first perform a pre-pass dataflow analysis that marks all registers used as addresses in load and store operations as pointers, then propagate that type information through the lifetime and use of those registers.

Table 1: Spatial Checking for Processors with no Fat Pointer Hardware Support

Primitive Operation	**ALPHA Operations**

pointer arithmetic
$d←$s+$t
(Sec. 4.8.3)
	addq $d.A, $s.A, $t
	cmpult $q,$d.A,$s.base
	cmpult $r,$s.bound,$d.A
	or $q,$q,$r
	blbs $q, bounds_error
	lda $d.base,$s.base,0
	lda $d.bound,$s.bound,0

store non-pointer mem[$s]←$t (Sec. 4.8.4)	cmpult $q,$s.A,$s.base cmpult $r,$s.bound,$s.A or $q,$q,$r blbs $q,bounds_error stq $t,0($s)
store pointer mem[$s]←$t (Sec. 4.8.4)	cmpult $q,$s.A,$s.base lda $r,$s.A,2 cmpult $r,$s.bound,$r or $q,$q,$r blbs $q,bounds_error stq $t.A,0($s.A) stq $t.base,1($s.A) stq $t.bound,2($s.A)
load non-pointer $t←mem[$s] (Sec 4.8.5)	cmpult $q,$s.A,$s.base cmpult $r,$s.bound,$s.A or $q,$q,$r blbs $q,bounds_error ldq $t,0($s)

load pointer
$t←mem[$s]
(Sec. 4.8.5)
	cmpult $q,$s.A,$s.base
	lda $r,$s.A,2
	cmpult $r,$s.bound,$r
	or $q,$q,$r
	blbs $q,bounds_error
	ldq $t.A,0($s.A)
	ldq $t.base,1($s.A)
	ldq $t.bound,2($s.A)

register target PC←$s	cmpult $q,$s.A,$s.base cmpult $r,$s.bound,$s.A or $q,$q,$r blbs $q,bounds_error jr $s.A

With all the registers annotated, we can then identify which operation must be an `addp` in our fat pointer architecture and which load and store operations are moving pointers to and from memory. Once identified, we perform a weighted sum of operations using the instruction counts from Tab. 1.

A.2 Overhead

Fig. 1 shows the estimated dynamic instructions in a non-fat pointer hardware ALPHA architecture performing spatial checks in software in comparison to our proposed architecture (see Sec. 3) with hardware support for fat pointers for a period of first 1 billion cycles. From the figure, we can observe that there is a maximum of 300% and on average, about 220% overhead in the number of dynamic instructions executed for performing spatial checks in software.