FLEXIBLE MEMORY PROTECTION FOR MULTICORE PROCESSORS

BY

MATTHEW ROBERT JOHNSON

THESIS
Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:
Professor Sanjay J. Patel
ABSTRACT

A system’s memory access control mechanisms profoundly impact the performance, reliability, security, and composability of the software it runs. Desirable features of an access control mechanism include: the ability to grant arbitrary permissions on any region of memory to any thread in the system; zero-copy sharing between threads with no restrictions on the set of sharers, region granularity, or sharing of pointer-based data structures; time and space overheads dependent only on the fundamental complexity of the access control being requested; and well-defined, hierarchical memory region ownership and permission semantics. The virtual-memory-based access control used in most modern systems, as well as recently proposed enhancements, fall short along one or more of these dimensions.

We introduce Lumen, an access control scheme providing security, fault isolation, and efficient shared memory to any number of threads within a single address space. Lumen uses a new concurrent interval skip list (ISL) to scalably maintain and query a set of memory region descriptors containing permissions information. We describe a permissions and ownership system for memory regions that allows safe delegation of privileges between protection domains for sandboxing and software reusability purposes. We describe the LumenCache, which caches region permissions to avoid most ISL lookups. We discuss applications of Lumen in debugging, security, and reliability, and extensions to use Lumen for prefetching, profiling, and user-facing memory management features. Lumen offers scalable, flexible memory access control and the ability to trade off security, system reliability, and performance in a way not possible with existing solutions.
I would like to thank the Department of Electrical and Computer Engineering and Dan Vivoli for supporting this work through the ECE Distinguished Fellowship and Dan Vivoli Endowed Fellowship, respectively.

I would like to thank my advisor, Professor Sanjay Patel, and Professor Steve Lumetta for their guidance and invaluable assistance in developing the ideas leading to this thesis. The members of the Rigel group, especially John Kelm, Danny Johnson, Neal Crago, Bill Tuohy, and Matt Frank, shaped my intuition for computer architecture and helped to develop the infrastructure with which I encountered the problems addressed by this work. Danny kindly indulged me in many long conversations about the finer points of LUMEN’s implementation tradeoffs, software interface, and applications.

My friends, especially Erin Lawton and Bethany Krebs, were a valuable source of encouragement, perspective, and home-cooked meals throughout this process. My parents Steve and LuAnn Johnson, to whom I owe nearly all my desirable qualities, have been steadfast in supporting me in my academic and personal endeavors. Finally, my wife Katya, to whom I owe the remainder of my desirable qualities, has served as a remarkable example of warmth, love, dedication, and patience.

Thank you all; without you, this thesis would not have been possible.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ABBREVIATIONS</td>
<td>v</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER 2 BACKGROUND</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Goals and Requirements</td>
<td>5</td>
</tr>
<tr>
<td>CHAPTER 3 LUMEN</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Data Structure</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Permissions</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Concurrency</td>
<td>17</td>
</tr>
<tr>
<td>3.4 Hardware/Software Interface</td>
<td>20</td>
</tr>
<tr>
<td>3.5 Caching</td>
<td>21</td>
</tr>
<tr>
<td>CHAPTER 4 EVALUATION</td>
<td>30</td>
</tr>
<tr>
<td>4.1 Address Space Complexity</td>
<td>30</td>
</tr>
<tr>
<td>4.2 ISL Characterization</td>
<td>30</td>
</tr>
<tr>
<td>4.3 System Performance</td>
<td>34</td>
</tr>
<tr>
<td>CHAPTER 5 LUMEN APPLICATIONS AND EXTENSIONS</td>
<td>38</td>
</tr>
<tr>
<td>CHAPTER 6 RELATED WORK</td>
<td>40</td>
</tr>
<tr>
<td>CHAPTER 7 CONCLUSION</td>
<td>44</td>
</tr>
<tr>
<td>APPENDIX A INTERVAL SKIP LIST STABBING QUERY RESULT ORDER</td>
<td>45</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>49</td>
</tr>
</tbody>
</table>
## LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>Access Control List</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASan</td>
<td>Address Sanitizer</td>
</tr>
<tr>
<td>CAM</td>
<td>Content Addressable Memory</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complimentary Metal-Oxide-Semiconductor</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random Access Memory</td>
</tr>
<tr>
<td>EBR</td>
<td>Epoch-Based Reclamation</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-process Communication</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>ISL</td>
<td>Interval Skip List</td>
</tr>
<tr>
<td>MLPT</td>
<td>Multi-Level Permissions Trie</td>
</tr>
<tr>
<td>MMP</td>
<td>Mondrian Memory Protection</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>NEBR</td>
<td>New Epoch-Based Reclamation</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PLB</td>
<td>Permissions Lookaside Buffer</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>PTE</td>
<td>Page Table Entry</td>
</tr>
<tr>
<td>QPI</td>
<td>QuickPath Interconnect</td>
</tr>
<tr>
<td>RMB</td>
<td>Range-Matching Block</td>
</tr>
<tr>
<td>RMC</td>
<td>Range-Matching Cell</td>
</tr>
<tr>
<td>RWX</td>
<td>Read-Write-Execute</td>
</tr>
<tr>
<td>SASOS</td>
<td>Single Address Space Operating System</td>
</tr>
<tr>
<td>SMR</td>
<td>Scalable Memory Reclamation</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>TCAM</td>
<td>Ternary Content Addressable Memory</td>
</tr>
<tr>
<td>TLB</td>
<td>Translation Lookaside Buffer</td>
</tr>
<tr>
<td>TSCache</td>
<td>Thread Set Cache</td>
</tr>
<tr>
<td>TSet</td>
<td>Thread Set</td>
</tr>
</tbody>
</table>
Memory access control has a profound impact on the performance, reliability, and flexibility of complex applications, particularly when data is shared among threads. The de facto mechanism for access control enforcement, virtual memory, forces a dichotomy between multi-threaded and multi-process models. Threads in the same process can share data at any granularity without copying, but no mechanisms exist to selectively share parts of an address space, requiring complete mutual trust between threads. Threads in different processes can use shared memory regions for selective zero-copy sharing, but cannot transparently share pointer-based data structures. Shared memory regions are restricted to page-granularity objects with inflexible semantics, which limits the flexibility and composability of the software that uses them.

Modern processors base access control on virtual memory mechanisms, fundamentally limiting the granularity at which data can be protected or shared and forcing programmers to compromise on memory efficiency or protection. Several flexible and fine-grained sharing and protection schemes have been proposed in the literature, but suffer from high overhead or fragile semantics. We introduce Lumen, a mechanism that supports arbitrarily fine-grained and flexible data sharing and protection between threads, and can function alongside or instead of virtual memory. Lumen performs well and provides much more flexible and composable semantics, encouraging software to make heavier use of memory protection.

The contributions of this thesis include:

1. Lumen, a novel memory protection mechanism that enables flexible, fine-grained, scalable protection

2. The first concurrent interval skip list, to our knowledge

3. The LumenCache, an interval metadata cache enabling Lumen implementations to achieve high performance at low hardware cost
4. A characterization of the *address space complexity* of modern personal, professional, and mobile computers
CHAPTER 2

BACKGROUND

The performance and programmability of a computing system is increasingly determined by its mechanisms for sharing data among threads. Hardware is becoming more parallel to extract increased system throughput in spite of plateauing single-thread performance. Programs are increasingly multithreaded in order to provide user interface responsiveness via concurrency, and to maximize performance via parallelism. More sophisticated user needs and an industry trend toward virtual server consolidation mean that modern machines are also running an increasing number of concurrent processes. Many of these processes also share data with one another, including library code and common memory-mapped files to conserve system memory, and shared memory segments for inter-process communication (IPC). A mechanism for sharing data between threads should therefore provide high performance and scalability while preserving process isolation and imposing as few programming model constraints as possible.

The memory management unit (MMU), translation lookaside buffers (TLB), and page tables on nearly all modern microprocessors combine to provide three key features for application developers and users: memory permissions, virtualization of physical memory, and memory mapping. These three features ensure isolation between multiple processes, enable compilation without regard for the target’s physical memory capacity or organization, and enable efficient and safe memory-mapped I/O and IPC. However, page-granularity virtual memory presents a semantic gap between the programmer’s protection and sharing goals and the implementation of those goals by the operating system and hardware [1].

The vast majority of multitasking systems today provide process isolation, shared memory between processes, and memory-mapped I/O using the virtual memory page table. Since the hardware implementation of these mechanisms is so similar even across processor architectures, the mecha-
nisms can be exposed to software through standard application programmer interfaces (APIs). Most UNIX-based operating systems use the `mmap()` and `mprotect()` to allocate and manage permissions on memory regions, respectively. While these low-level primitives are sufficient to implement shared memory regions between processes, most of these operating systems also provide higher level APIs conforming to the POSIX or System V specifications. Windows provides `VirtualAlloc()` and `VirtualProtect()` as analogs to `mmap()` and `mprotect()`, respectively.

We now describe the most common hardware implementation of process isolation and shared memory regions. Each process is assigned its own virtual address space and page table, and memory belonging to other processes is inaccessible by virtue of being unaddressable. Two zero-copy mechanisms exist to share a data structure between $N$ threads of control: multithreading and operating system (OS)-managed shared memory regions. Both mechanisms have significant drawbacks under virtual memory. Placing threads in the same process gives them unlimited access to each other’s memory, which is undesirable for security and reliability. Placing threads in separate processes and placing the data structure in a shared memory region has three disadvantages:

1. Processes cannot share or set permissions on memory at a sub-page granularity.

2. Data structures with pointers cannot generally be shared transparently, since the system cannot guarantee that all processes will map the region at the same virtual address.

3. The shared region is represented in each process’ page table, resulting in increased storage overhead, region modification latency, and TLB pressure.

The first limitation can be worked around to some extent by laboriously splitting program data into regions with common sharing properties and permissions. Pointer swizzling has been used in databases and distributed shared memory systems to share pointer-based data structures [2]. Both techniques carry a large performance and complexity cost.

Page-granularity protection forces a fundamental tradeoff between performance and protection. We use a cycle-accurate, execution-driven simula-
tor [3] to evaluate the TLB behavior of a suite of data- and task-parallel benchmarks [4]. Figures 2.1(a) and 2.1(c) characterize many-core L1 TLB performance versus TLB size and page size, respectively. The small page sizes desired for low fragmentation and fine-grained protection require correspondingly large TLBs to achieve the desired miss rate of less than one percent.

2.1 Goals and Requirements

The high-level goals of Lumen are to enable fine-grained memory protection and sharing, to achieve high performance and scalability, and place as few constraints on application programmers as possible in terms of data layout and code structure. Each of these goals leads to a small set of core design requirements that guide our proof-of-concept implementation.

In order to achieve fine-grained memory protection and sharing, Lumen breaks from the page frame-centric model imposed by virtual memory and specifies memory region protection and sharing properties explicitly, a model more similar to segments. Memory segmentation, a model first used on the Burroughs B5000 [5], divides memory into semantically separate segments, and each memory operation is expressed as a segment identifier and a byte offset within the segment. Segmentation can be used to virtualize memory by relocating segments transparently within a larger address space, and to prevent a thread from addressing memory outside its assigned segments. However, unlike traditional segment implementations, Lumen must support the dynamic creation, destruction, and modification of arbitrarily nested segments. The binding between a memory instruction and the segment it refers to must therefore happen at runtime, as opposed to binding memory instructions to segment registers at compile time. Segment-agnostic memory instructions and pointers have the additional benefit of enabling segment-agnostic code to be used on data from many segments [6].

Abstractly, we conceptualize permissions and sharing metadata as a hierarchical collection of intervals; each interval consists of an address range and a function encoding the permissions each thread in the system has on that address range. To simplify queries, page tables and existing fine-grained segment schemes [7, 8] flatten the interval metadata along one or more di-
Figure 2.1: Average TLB behavior of 10 parallel workloads on a simulated many-core processor. Large TLBs or large pages can provide high performance, but impose high hardware and fragmentation costs, respectively.
mensions; that is, they store a single set of permissions per address and do not necessarily preserve the high-level structure of the permissions information. This flattening has two disadvantages. First, it introduces duplication which leads to performance and storage overhead; for example, $N$ processes sharing an $M$-page region under virtual memory lead to $MN$-way translation and permissions duplication. Second, flattened representations typically throw away information about the interval itself after its data is replicated, losing the ability to add and remove intervals dynamically with consistent semantics. For example, consider a case where a thread has read-write permissions on the interval $[0, 100]$, temporarily marks $[30, 70]$ as read-only, then removes the read-only interval. Ideally, when the interval is removed, $[30, 70]$ would automatically become read-write again, but flattened representations discard information about the previous permissions. As a result, applications must either explicitly track hierarchical permissions information themselves or compromise security and reliability by adopting more permissive protection policies.

Another disadvantage of virtual memory-based protection is that it forces a strict dichotomy between multi-threaded and multi-process models. Threads in the same process enjoy flexible, zero-copy data sharing, but must grant one another full permissions to the entire address space. Threads in different processes have tighter control over permissions, but must communicate either through slow IPC channels or shared memory regions which may be mapped at different virtual addresses, making it difficult to share pointer-based data structures. Our goal with Lumen is to enable a thread to grant access to shared memory regions on a region-by-region, thread-by-thread basis. The ability to get the best of the multi-threading and multi-process programming models will allow developers to use shared memory and memory protection more pervasively, improving the overall performance and reliability of future computer systems.

While virtual memory offers fixed performance and fixed protection granularity for a given page size, Lumen allows application developers and users to trade off access control granularity and performance. The tradeoff chosen can vary between applications, or for a single application over its development lifecycle; the developer can use very fine-grained protection as a debugging mechanism, then improve performance over time by doing coarser-grained checking as the program becomes more reliable. Users can also perform this
tradeoff to fit their security, performance, and reliability needs if applications expose mechanisms to increase or reduce the strictness of LUMEN intervals. In the limit, a system running completely trusted software may grant all threads full permission to the full address space with a single interval; the interval metadata can be trivially cached, and LUMEN’s runtime overhead goes to zero.
In this chapter, we describe the design and implementation of Lumen, a flexible, efficient data structure for storing and querying memory interval metadata, including permissions.

3.1 Data Structure

Lumen must support efficient queries and updates on a set of intervals representing memory regions, stored as <StartByteAddress, EndByteAddress, Metadata> tuples. The metadata must include read-write-execute permissions for all threads in the system, but may include other information, as discussed in Section 5. We call each tuple an interval descriptor. The data structure used to index interval descriptors must efficiently find all intervals that intersect a query interval corresponding to the bytes accessed by a memory operation.

Many data structures have been proposed for indexing intervals efficiently, including the interval tree [9], the segment tree [10], and the radix priority search tree [11]. For Lumen, we impose the additional constraint that the data structure be dynamic; to be useful in a running system, it must support efficient interval insertion and deletion at runtime. The best known bounds for the general case of interval intersection queries on a dynamic dataset are $O(n \, lg(n))$ space and $O(lg^2(n) + k)$ time, where $n$ is the total number of intervals and $k$ is the number of matching intervals returned by the query [12]. However, for Lumen, the query interval's size is bounded by the maximum load or store width of the machine, $W$. We can take advantage of this fact to reduce the time bound to $O(lg(n) + k + W)$.

While several tree-based structures achieve the optimal time and space bounds, trees must be kept balanced to maintain logarithmic search time.
These balancing operations are often global in scope, reducing concurrency by locking large portions of the tree. We therefore focus our attention on data structures based on the skip list [13]. A skip list stores a set of sorted items using a linked list; each node contains a key and an array of at least one forward pointer. The first forward pointer, called level 0, implements a linked list of all nodes in sorted order. Any additional pointers are linked together by level and constitute shortcuts to nodes further down the list, reducing query time from $O(n)$ to $O(\lg n)$. Skip lists maintain balance probabilistically and without global operations by assigning a random number of forward pointers to each node at insertion time. To find a key $K$, start at the top level of the header node and follow all forward pointers at that level until the forward pointer sink node’s key is greater than $K$. Once such a pointer has been found, drop down one level and repeat the process until key $K$ is found or a key greater than $K$ is found on level 0, indicating that $K$ is not stored in the skip list.

Figure 3.1: LUMEN data structures for a set of 5 intervals. We highlight all data related to interval $a$. (a) Intervals must be hierarchical to give each address implicit, unambiguous permissions based on its enclosing intervals. (b) The ISL data structure contains a node for each distinct interval endpoint and interval markers on relevant nodes and forward pointers. (c) Interval markers point to descriptors, linked lists of permissions tuples encoding ownership information, maximum permissions, and current permissions for one or more thread sets. (d) A thread set is encoded as a chunked linked list of contiguous thread ID ranges (e.g., TSet 2 contains threads 2 and 9–16).
Hanson and Johnson introduced the interval skip list (ISL) [14], an extension to the skip list for finding all intervals that intersect a query point. Lumen uses an ISL, as shown in Figure 3.1b, and modifies the query procedure to allow a small query interval with the aforementioned optimal time and space bounds.

An ISL is a skip list with a node for each unique interval endpoint and a set of interval markers associated with each node and forward pointer; a marker is simply a pointer to an interval descriptor. We describe the layout of each interval descriptor in Section 3.2. Lumen’s ISL stores intervals on the set of integer byte addresses, so each node’s key corresponds to the byte address of the endpoint of one or more intervals. We denote an interval with endpoints $A$ and $B$ as $[A, B]$, and a forward pointer between two nodes with keys $X$ and $Y$ as $[X, Y]$. To insert an interval $I = [A, B]$, we place markers for $I$ on all forward pointers $[X, Y]$ in the ISL such that $I$ contains $[X, Y]$ and no other forward pointer exists that lies within $I$ and contains $[X, Y]$. Hanson and Johnson call these two conditions containment and maximality. For each marked forward pointer $[X, Y]$, we also place a marker on nodes $X$ and $Y$ if they are within $I$. To find all intervals containing a point $K$, we traverse the nodes and forward pointers as if searching for $K$ in a skip list. Whenever we drop from level $i$ to level $i-1$, we add all markers on the level $i$ forward pointer of the current node to the list of intersecting intervals. At the end of the search, if we find node $K$, we add its markers to the list; otherwise, we add the markers on the level 0 pointer of the last node with key less than $K$. The containment and maximality conditions guarantee that the list of intervals intersecting point $K$ is complete and contains no duplicates. This point-interval intersection query is known as a stabbing query, and is shown in the first part of Algorithm 1.
For **Lumen** to be an efficient protection mechanism for modern systems, the ISL must be able to check permissions on a small contiguous byte range without performing individual stabbing queries for each byte. Such a facility is necessary to support word-wide memory accesses and per-byte permissions. We present an extension to the ISL that allows intersection queries with small query intervals. For a **Lumen** query with an interval \([K, K+W]\), we search for \(K\) using the stabbing query procedure outlined above, then continue traversing along level 0 until we find a key greater than or equal to \(K+W\), picking up markers on each node and all its forward pointers as we go. We call this second phase of the query the *conflict phase*. We compute permissions exactly for the first byte in the interval during the stabbing query phase; for subsequent bytes, we merely check to see if any enclosing intervals have permissions that contradict the first byte’s (i.e., a needed permission is set to ‘1’ in the first byte and to ‘0’ on one or more subsequent bytes). This more relaxed check during the conflict phase is necessary for the interval intersection query to be performed in constant space since, unlike in the stabbing query phase, we may encounter duplicate intervals and intervals are not guaranteed to be ordered by size.

If any byte in the interval does not have the required permissions to satisfy the read, write, or execute access, **Lumen** throws a segmentation fault or equivalent error to the accessing program.

Since loads and stores under **Lumen** seldom cross protection interval boundaries, there will often be no nodes or markers between \(K\) and \(K+W\), but if there are, markers picked up in the conflict phase may contain duplicates. We address this complication in Section 3.5.2.

Algorithm 1 describes the procedure for finding all intervals in an ISL intersecting a small query interval.

To insert a node \(N\) with \texttt{level} forward pointers:

- Search for \(N.key\) and build up a list \(\texttt{update[level-1:0]}\), where \(\texttt{update[i]}\) is the node that will point to \(N\) at level \(i\)

- Remove all markers on forward pointers that will point to \(N\), place them in a set \(S\)

- Update forward pointers in and out of \(N\)

- For each interval \(I = [X, Y]\) in \(S\), place markers for \(I\) along the search
Algorithm 1 Find all intervals intersecting \([K, K + W]\) in an ISL \(isl\):

\[
\begin{align*}
 n &\leftarrow isl.\text{header} \ \{\text{Stabbing Query Phase}\} \\
 &\text{for } l = isl.\text{maxlevel} \text{ to } 1 \text{ do} \\
 &\quad \text{while } n.\text{next}[l].\text{key} \leq K \text{ do} \\
 &\quad \quad n \leftarrow n.\text{next}[l] \\
 &\quad \quad m \leftarrow m \cup n.\text{markers}[l] \\
 &\quad \text{while } n.\text{next}[0].\text{key} \leq K \text{ do} \\
 &\quad \quad n \leftarrow n.\text{next}[0] \\
 &\quad \text{while } n.\text{key} < K + W \ \{\text{Conflict Phase}\} \text{ do} \\
 &\quad \quad m \leftarrow m \cup n.\text{nodemarkers} \cup n.\text{markers}[(n.\text{level} - 1) : 0] \\
 &\quad \text{for } i = 0 \text{ to } n.\text{level} - 1 \text{ do} \\
 &\quad \quad m \leftarrow m \cup n.\text{node} \\
 &\quad \text{if } n.\text{key} == K + W \text{ then} \\
 &\quad \quad m \leftarrow m \cup n.\text{nodemarkers} \\
 &\quad \text{else} \\
 &\quad \quad m \leftarrow m \cup n.\text{markers}[0] \\
 &\text{return } m
\end{align*}
\]

path from \(\text{max}(\text{update}[\text{level} - 1].\text{key}, X.\text{key})\) to \(\text{min}(\text{update}[\text{level} - 1].\text{next}[\text{level} - 1].\text{key}, Y.\text{key})\)

To insert an interval \(I = [X, Y]\):

- Insert \(X\), if not present
- Insert \(Y\), if not present
- Place markers for \(I\) along the search path from \(X\) to \(Y\)

We refer the reader to Hanson and Johnson’s paper [14] for the full node and interval insertion and deletion procedures.

In this work, we augment the query, insertion, and removal procedures to support concurrent accesses efficiently, as described in Section 3.3.
3.2 Permissions

While Lumen allows the attachment of arbitrary metadata with any semantics to a memory interval, we describe a permissions scheme that improves flexibility and composability for programmers over the per-process RWX bits used in page table-based memory protection.

An ISL can store multiple intervals covering a single address; separate intervals should be used to represent semantically meaningful memory regions, like a thread’s stack, a memory-mapped file region, or a member of a C struct. These regions tend to be hierarchical; that is, the protection intervals we want to insert tend to be subintervals of one another, where an interval $J = [C, D]$ is a subinterval of $I = [A, B]$ iff $A \leq C \leq D \leq B$. Furthermore, it is difficult to reconcile conflicting permissions on non-hierarchical intervals. Consider two overlapping intervals, $X = [0, 12]$ with read-write permissions and $Y = [8, 20]$ with no permissions; without a complex and fragile notion of per-interval priority, the permissions on address 10 cannot be determined. We therefore enforce the condition that an interval can only be inserted into the ISL if it contains, is contained by, or does not overlap with every other interval in the ISL. More formally, for any two intervals $a = [L_a, R_a]$ and $b = [L_b, R_b]$, $L_b \in [L_a, R_a] \iff R_b \in [L_a, R_a]$. We call this condition the pyramid constraint, a reference to the pyramid-like shapes in the abstract representation of the ISL as shown in Figure 3.1a, with smaller intervals above larger ones. With this constraint, the intervals covering a given address can be sorted by size; smaller, and thus more specific, intervals take precedence over larger ones, regardless of the order in which they were inserted.

To further decouple intervals from one another, we allow an interval to specify partial permissions. That is, we allow an interval to defer to the next largest enclosing interval for one or more RWX permissions, rather than require each interval to set all permissions explicitly. We encode RWX permissions with 2 bits each and allow three states: ‘1’, ‘0’, and ‘X’ (defer). As an example, if an interval $I = [20, 30]$ has permissions 1X0 and $J = [10, 40]$ has permissions 011, $I$ defers to $J$ for write permissions and address 10–40 have permissions 110. This ability to specify partial permissions is analogous to inheritance in object-oriented programming, and allows separation of concerns between intervals; libraries and applications can manage
their intervals largely in isolation, with predictable results even when they share data. In contrast, under page-based protection, the flattened nature of the permissions data necessitates close cooperation between all entities that manage permissions on a given address; this required cooperation limits the programmer’s ability to compose multiple permissions-aware libraries or applications. Partial permissions can also improve the performance of permission management operations. If many small intervals defer to a common enclosing interval for a certain permission, simply modifying that permission on the parent will implicitly propagate the change to all subintervals. Beyond just a performance increase, it allows the permissions for large memory regions to be changed atomically, a new capability with interesting implications for security, reliability, and synchronization primitives in parallel systems.

To ensure system reliability and security in the face of untrusted or buggy software, a shared memory mechanism must allow a set of threads to delegate permissions on a memory region to other threads on a fine-grained basis while maintaining ultimate control of the region. The virtualization extensions found on many modern microprocessors enable a limited form of such delegation between host and guest operating systems [15]. Existing shared memory APIs either provide no ownership semantics and assume fully cooperative threads (e.g., anonymous `mmap()`) or treat memory regions as files and provide semantics akin to UNIX filesystem permissions or access control lists (ACLs). Since the virtual memory hardware does not support either permission model natively, they are enforced in software by the operating system, requiring expensive page table scans and TLB shootdowns when permissions are changed. Additionally, under OS-managed permissions schemes, a memory region is treated as a single file, so permissions cannot be changed dynamically on part of a region. In **Lumen**, we support a form of the stronger model, ACLs, natively in each interval’s metadata, providing more fine-grained, flexible, and performant protection than is available today.

Each interval’s descriptor contains an ACL, specified as a set of `<ThreadSetID, OwnershipLevel[1:0], MaxPermissions[5:0], CurrentPermissions[5:0], CreatorLevel[1:0]>` tuples, as shown in Figure 3.1c. In our implementation, we encode this set as a linked list of 64-bit values and leave the investigation of a more dense encoding to future work. Thread sets have no access to an address unless they are specified in the ACL of an interval covering that
address. Each thread set in the ACL has a set of maximum RWX permissions and a set of current permissions, which it can modify freely as long as they do not exceed the maximum permissions. We rank the three permissions states from greatest to least as ‘1’, ‘X’, ‘0’; if any element of an RWX permission set \( P_1 \) is higher than the corresponding element of \( P_2 \) in that ranking, \( P_1 \) is said to exceed \( P_2 \). If a thread is a member of multiple thread sets specified in a single interval ACL, the thread can assume the most lenient of all associated RWX permissions. The maximum permissions are set when the permissions tuple is inserted. The OwnershipLevel field establishes a hierarchy of rights holders, with higher-numbered levels denoting fewer rights. Only a single thread set can be at ownership level 0 for a given address; this thread set is called the owner and must be specified in the largest interval enclosing the address. A thread set with ownership level \( L \) and maximum permissions \( P \) on an interval \( I \) can perform the following actions:

- Set current permissions \( \leq P \) on \( I \) and access addresses in the interval with those permissions
- Add subintervals of \( I \) satisfying the pyramid constraint
- Add permissions tuples for itself or other thread sets on \( I \) or its subintervals, setting \( \text{MaxPermissions} \leq P \) and \( \text{CreatorLevel} \geq L \)
- Modify or remove permissions tuples with \( \text{MaxPermissions} \leq P \) and \( \text{CreatorLevel} \geq L \) on \( I \) or its subintervals
- Remove \( I \) or a subinterval of \( I \) if the interval to be removed has no subintervals and no attached permissions tuples

These rules allow permissions to be delegated easily by any thread, while preventing direct or indirect privilege escalation and protecting permissions granted by a thread from being modified by less privileged threads. In our implementation, we allocate two ownership level bits per permission tuple, allowing four distinct levels. While two levels are sufficient to implement the usual kernel/userspace ownership hierarchy, additional levels give applications and libraries the new ability to safely delegate control over regions of memory to non-cooperative or even malicious threads.
Thread Sets  Our implementation allows up to $2^{16}$ threads and $2^{48}$ thread sets in the system. Thread set $2^{48} - 1$ is reserved as a wildcard; an interval descriptor may refer to this thread set to apply permissions globally, with the caveat that the wildcard thread set cannot have ownership level 0. When a thread is spawned and assigned a thread ID $t$, it is also assigned a default thread set with ID $t$, with itself as the only member. The thread may then use thread set $t$ immediately for private data or add other threads to it; the latter case simulates a multithreaded process in a traditional operating system. Thread set $2^{16}$ is reserved for the operating system. Thread set membership is encoded in two data structures. Each thread has a chunked linked list of thread sets of which it is a member. Each thread set has a chunked linked list of the contiguous thread ID ranges of its members, as shown in Figure 3.1d. Encoding a thread set’s members as contiguous ranges optimizes for the common case of sharing occurring within tight cliques of threads, as is the case in today’s multithreaded processes; these cliques should attempt to allocate contiguous thread IDs from the operating system for maximum encoding efficiency.

3.3 Concurrency

If we are to encourage libraries and applications to make more aggressive use of permissions mechanisms, the mechanisms must perform well in the highly parallel systems being built and designed today. In practice, this means that LUMEN should follow the trend in modern operating systems of favoring fine-grained locking and lock-free synchronization in shared data structures wherever possible.

We experimented with several synchronization strategies for LUMEN, and settled on a hybrid of fine-grained locking and lock-free operations. Lock-free synchronization primitives on modern processors provide excellent scalability by shrinking critical sections to a single cycle in many cases. These primitives are usually implemented by using cache coherence mechanisms to prevent intervening accesses during the atomic operation, or by serializing operations through dedicated functional units located at a central location such as the last level cache. Therefore, though hardware atomic operations provide much higher throughput than small critical sections protected by locks, each indi-
Individual atomic operation is significantly slower than its non-atomic equivalent. In data structures with complex invariants like the ISL, the high-level operation to be performed concurrently can be complex, and although it can be broken down into a sequence of atomic operations in some cases, such a sequence may in fact be slower and more error-prone than lock-based synchronization, particularly in situations with low contention. Several concurrent skip lists have been proposed in the literature [16, 17, 18], but skip lists are fundamentally easier to make lock-free than interval skip lists. In particular, the forward pointers in a skip list are purely performance hints, and a lock-free skip list may lose forward pointers without impacting correctness; indeed, the implementation used in Java’s ConcurrentSkipListMap class acknowledges this explicitly [16]. In contrast, precise forward pointer and marker placement is critical to ISL correctness, and inserting or removing an interval requires a large number of markers to logically update consistently and atomically. We thus place read-write locks on each node to prevent ISL modifications from conflicting and to prevent queries from observing inconsistent state. These locks allow either one writer or an arbitrary number of readers to access a node simultaneously. All operations acquire locks in increasing order of keys, or left to right in the abstract ISL representation in Figure 3.1a; this strategy prevents deadlock, and is similar to Herlihy et al.’s suggestion to define such a total order on skip list node locks [19].

When inserting or deleting a node X, the scope of potentially affected ISL nodes and edges is from the source node of the top edge into X to the sink node of the top edge out of X. We write-lock all nodes in this range while updating pointers and markers surrounding the node of interest. When a node X is deleted, all nodes pointing to X are write-locked, so no other threads can hold references to X and X’s memory can be reclaimed immediately. Figure 3.2 shows the results of 1 billion trials of random ISLs of several sizes. The mean operation locks 3.3 nodes for a 4-node ISL and 21.9 nodes for a 2-million-node ISL. The 95th and 99th percentile operations level off at 29 and 145 locks, respectively, at 2 million nodes. Statistically, insertions and deletions seldom lock a significant portion of the ISL, and absolute overhead is low even for the very complex ISLs found in debugging situations. Node insertions and deletions will block some reads from proceeding past them, but reads can bypass the write-locked nodes if they traverse a shortcut pointer at a higher level than the node being inserted or removed. In general, due to
the height distribution of nodes, write-locked nodes seldom block concurrent reads to unrelated addresses.

Queries and the marker insertion phase of interval insertion proceed using hand-over-hand read locking, without acquiring any node write locks. To traverse an ISL edge from source to sink, start by read-holding source.lock, read-acquire sink.lock, traverse the edge, and read-release source.lock.

Bit 0 of an interval first permissions tuple’s next pointer is used as a flag, manipulated with atomic compare-and-swap, denoting whether (0) or not (1) the interval is logically inserted into the ISL. The flag is reset at the very end of interval insertions and set at the very beginning of interval deletions. Queries ignore intervals if their flag is set.

To optimize the ISL for the common case, queries, we store the markers on each forward pointer and node as a singly linked list, where readers can proceed without any locking and writers must acquire a spinlock. Since readers are lock-free, a marker’s memory cannot be freed until no readers have references to it. This problem is called scalable memory reclamation (SMR) in the context of parallel systems, and is common to nearly all lock-free data structures. While many solutions exist, we use a variant of Fraser’s epoch-based reclamation (EBR) [20] termed new EBR (NEBR) by Hart et al [21]. All

Figure 3.2: Mean, 95th percentile, and 99th percentile number of nodes locked for ISL node insertions and deletions vs. ISL size.
SMR solutions reclaim memory when it can be proven that no thread holds a reference to that memory. In EBR, a global epoch counter is incremented periodically, and participating threads observe the epoch number when they begin a critical section which may acquire a reference to the memory covered by EBR. Each thread maintains a global variable recording the last observed epoch number; if a piece of memory is removed from the data structure during epoch \( X \), it may be reclaimed once all threads have observed a subsequent epoch \( E > X \). The developer of an EBR-based data structure may vary the way the epoch counter is incremented and the size of the EBR-protected code section to trade off performance and reclamation latency. NEBR is a coarse-grained variant of EBR for data structures where a single high-level operation encompasses many small critical sections; in LUMEN, rather than observing the epoch for each marker list reference, a thread only observes the epoch once per high-level query, insertion, or removal operation, improving performance significantly.

3.4 Hardware/Software Interface

At a high-level, LUMEN operates similarly to page-based protection; it intercepts all memory accesses at the core to verify that the issuing thread has the required permissions on the bytes being accessed. Insufficient permissions will cause a signal to be sent asynchronously to the issuing thread, which can attempt to recover or exit.

LUMEN can be employed in two ways: a single system-wide ISL covering all kernel and user threads in a single address space operating system (SASOS), or on top of virtual memory mechanisms with a separate ISL for each virtual address space. When the ISL is initialized, a single interval is inserted giving the controlling thread set ownership level 0 and full permissions on the entire address space. The memory used for the ISL itself is part of the address space. The operating system can set permissions on these allocation arenas to either force all ISL operations to be mediated by the operating system or trusted hardware walker, or allow some operations to proceed fully in userspace, according to the system’s performance and reliability goals. Table 3.1 summarizes the differences between the SASOS and per-address-space implementations.
Table 3.1: Properties of single address space operating system and per-virtual-address-space LUMEN implementations. Implementations with one ISL per virtual address space may trade off performance for reliability by allowing some ISL modifications or queries to occur in userspace.

<table>
<thead>
<tr>
<th></th>
<th>SASOS</th>
<th>ISL Per Virtual Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISL initialized at</td>
<td>boot</td>
<td>process creation</td>
</tr>
<tr>
<td>Controlling TSet</td>
<td>OS</td>
<td>parent thread</td>
</tr>
<tr>
<td>Permissions for other threads on ISL memory</td>
<td>None</td>
<td>None or read-only</td>
</tr>
<tr>
<td>ISL modified by</td>
<td>OS or HW walker</td>
<td>Any thread or HW walker</td>
</tr>
<tr>
<td>ISL queried by</td>
<td>OS or HW walker</td>
<td>Any thread or HW walker</td>
</tr>
</tbody>
</table>

LUMEN does not require any ISA extensions to operate efficiently. In systems with low security and reliability needs, trusted software, or a trusted compiler, the ability to tag memory instructions as requiring or not requiring ISL lookup would increase performance. Table 3.2 outlines our proof of concept API.

3.5 Caching

Programs often exhibit very high temporal locality in the memory regions they access [22, 23]. Just as TLBs as necessary for performant virtual memory, for LUMEN to be usable in a high-performance system, permissions must be cached at the core to minimize per-memory-access latency. Caching permissions also speeds up those operations which do have to traverse the ISL by reducing contention. In this section, we introduce the LUMENCACHE, a novel hardware cache for interval metadata.

The LUMENCACHE consists of two content-associative memories (CAMs): an interval cache and a thread set membership cache (TSCache). Figure 3.3 illustrates the structure of the interval cache.

The interval cache is logically an array of comparators. A naïve implementation using static CMOS comparators would likely suffice for a small interval cache, but would carry significant overhead for large numbers of entries because traditional comparators are large and ill-suited to a dense array implementation. We therefore construct the interval cache using a variant of
Table 3.2: The Lumen API may be implemented as system calls or a library, depending on the desired level of performance and protection.

```c
struct PermissionsTuple { uint64 tsid : 48, creator_level : 2, own_level : 2, max_perms : 6, cur_perms : 6 }

IntervalHandle insert_interval(void *low_addr, void *high_addr, PermissionsTuple perms[])
Inserts an interval with permissions for the specified TSets; returns a pointer to the interval desc., or NULL on failure

int remove_interval(IntervalHandle handle)
Removes the given interval; returns 0 on success, -1 on failure

int modify_interval_perms(IntervalHandle handle, PermissionsTuple perms[])
Modifies (if present) or adds (if not) permissions tuples for the specified TSets; returns -1 if any TSets failed

remove_interval_perms(IntervalHandle handle, uint64_t thread_sets[])
Removes permissions tuples for the specified TSets; returns -1 if any TSets failed

IntervalHandle[] get_intersecting_intervals(void *low_addr, void *high_addr)
Get the set of intervals which intersect [low_addr, high_addr] and have a permissions tuple pertaining to the issuing thread

int get_perms(void *low_addr, void *high_addr)
Get the final computed permissions for [low_addr, high_addr] for the issuing thread; returns -1 if permissions are inconsistent

void set_permfault_handler(void (*handler)(void *addr, int requested_perms, int computed_perms))
For per-VA-space Lumen implementation, set the address of the handler used for permission faults

SASOS implementations of Lumen use the existing signal handling infrastructure and deliver permissions faults via SIGSEGV
```

Table 3.3: Area complexity of CMOS comparators. Complexity includes one 6T SRAM cell per bit. Complexity does not include buffers used to improve timing, but buffers should add a small, relatively equal overhead to all designs. TG stands for transmission gate, and PT stands for pass transistor. PG and PG* are specialized propagate-generate circuits [24].

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Components (N-bit)</th>
<th>XTORS</th>
<th>XTORS (N = 64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static, Serial [25]</td>
<td>N(4NOT + XOR + 2NAND + 2TG + PT + SRAM)</td>
<td>39N</td>
<td>2496</td>
</tr>
<tr>
<td>Static, Parallel [26]</td>
<td>N(NOT + AND + SRAM) + (N − 1)(OR + 2TG) + (N − lg(N) − 1)(2TG + NOT)</td>
<td>30N − 16 lg(N) − 16</td>
<td>1868</td>
</tr>
<tr>
<td>Static, RMC [27]</td>
<td>N(2TG + 1NOT + 6PT + SRAM)</td>
<td>18N</td>
<td>1152</td>
</tr>
<tr>
<td>Dynamic, RMC [27]</td>
<td>N(9PT + SRAM)</td>
<td>15N</td>
<td>960</td>
</tr>
<tr>
<td>Static, reduced-area RMC</td>
<td>N(2TG + 1NOT + 4PT + SRAM)</td>
<td>16N</td>
<td>1024</td>
</tr>
</tbody>
</table>
Figure 3.3: Interval cache and reduced-area range-matching cell structure. Matchline drivers, buffers, and keepers vary with process technology and are not shown.
the Range-Matching Cell (RMC) proposed by Kim et al. [27]. The RMC is an efficient matchline-based comparator originally developed for port range matching in networking applications. An RMC takes roughly 80% of the area and $2 \times$ the search power of a static NAND-type ternary CAM (TCAM) cell. Table 3.3 shows that an RMC-based CAM is significantly more dense than other complementary and transmission gate-based CMOS implementations. The reduced-area static matchline RMC introduced in this work uses 45-59% fewer transistors than the other organizations. The RMC-based organizations are also more amenable to a regular array organization than the parallel and prefix-adder implementations, since each cell only connects to its immediate neighbors.

Kim et al. present two versions of the RMC, both using a pass transistor on the matchline between $ML_{in}$ and $ML_{out}$. A CAM using the dynamic matchline RMC precharges the match-line before each search, and one or more RMCs can pull the match-line low to signal a mismatch. In contrast, the static matchline RMC can actively drive $ML_{out}$ high or low if its value can be determined independently of $ML_{in}$. The static version requires three more transistors per cell and is less symmetrical, but has several advantages over the dynamic version. Dynamic logic circuits precharge a storage node during the precharge phase, then allow the pull-down network to either pull the storage node down to a logic ‘0’ or not during the evaluation phase. Two important phenomena that adversely impact dynamic logic are charge sharing and leakage [28]. Charge sharing occurs when one or more of the pull-down transistors are on; the charge on the storage node is split among the storage node capacitance and the capacitances of the ‘on’ transistors, reducing the voltage on the storage node and negatively impacting evaluation speed and/or correctness. Leakage occurs when the charge on the storage unit dissipates through parasitic resistances over time, also reducing the voltage on the storage node. Both of these phenomena have become more prominent in deep submicron technologies, making static matchlines much more attractive on these technology nodes. If desired, a designer can mitigate these disadvantages of dynamic logic by inserting extra keeper transistors to decrease leakage and buffer the match-line every few cells to mitigate charge sharing and increase speed. Another advantage of the static match-line scheme is that it only charges a segment of the match-line if the cell matches, avoiding the energy cost of unconditionally precharging the entire line for every
search. Indeed, Kim et al. found that a 256x16-bit static match-line CAM was 33% larger and 20% slower than the most optimized dynamic version, but also used 35% less energy per search. For the small array sizes we anticipate for a first-level LumenCache, the static RMC presents a favorable tradeoff. We reduce the overhead of the static RMC over the dynamic RMC to only one transistor per cell by removing two transistors used only in the equality operation.

Though Figure 3.3 shows a static matchline, either architecture can be used; a small LumenCache can use static matchlines to reduce power, and a large LumenCache can use dynamic matchlines to minimize search time.

A block of $n$ RMCs, called a Range Matching Block (RMB), can perform $n$-bit equals (EQ), less than or equal to (LEQ), and greater than or equal to (GEQ) operations between a search word and a word stored in the per-RMC SRAM cells. Each interval cache entry consists of a pointer-sized RMB for each endpoint of the cached interval, a thread set ID, and the computed RWX permissions for that thread set. By tagging entries with thread set IDs, we allow threads in the same protection domain to share entries and avoid flushing the interval cache at context switches.

Since threads may belong to multiple thread sets at a time, we also maintain the TSCache, a small (2-8 entries) secondary cache of thread set IDs of which the running thread is a member. The thread set ID cache is kept consistent by conservatively flushing it on rare thread set membership changes.

To search the LumenCache for a memory access to bytes $[X,Y]$, configure the interval cache’s start and end RMBs to perform LEQ $X$ and GEQ $Y$ operations, respectively. If both RMBs in an entry match, the interval cache will output that entry’s set ID and permissions. We then search the TSCache to see if the requesting thread belongs to the relevant thread set. If the TSCache misses, we search the thread’s membership list. If the thread set ID is found, it is filled into the TSCache and the permissions from the interval cache are used. If not, or if the interval cache misses, we query the ISL to find the permissions for the query interval. During this query, the TSCache is used to quickly determine which covering interval descriptors are relevant to the thread. The smallest enclosing interval, calculated permissions, and applicable thread set ID are then filled into the interval cache.

All interval insertions, deletions, and permissions changes occur through the in-memory ISL, not the interval caches. To keep all interval caches con-
sistent, all ISL modifications broadcast a shootdown for the affected interval. For queries, we could further reduce interval cache area by specializing each RMB to only perform either LEQ or GEQ operations. However, by only removing the EQ capability, we still reduce area compared to the baseline RMC and can support 3-cycle flash invalidation of an arbitrary interval in the entire cache. A cached interval \([A, B]\) should be invalidated on a shootdown for interval \([C, D]\) iff 
\[
(C \leq A \land B \leq D) \lor (A \leq C \land C \leq B) \lor (A \leq D \land D \leq B).
\]
This reduction in shootdown overhead from linear time to three cycles is critical for supporting large interval caches or frequent permissions changes. Excessive shootdown traffic still increases ISL modification latency and network congestion; Villavieja et al. [29] found that TLB shootdowns caused slowdowns of up to 2× for the apache webserver at 128 cores. Therefore, system designers may choose to reduce shootdown traffic by tracking which LUMEN CACHES are caching which intervals in a directory; directories are commonly used for data caches, and Villavieja et al. proposed their use for TLBs.
Table 3.4: Estimated fully associative TLB and LUMENCACHE area

<table>
<thead>
<tr>
<th></th>
<th>Entries</th>
<th>Area (45nm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB (Fully Associative)</td>
<td>8</td>
<td>1290 μm²</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1940 μm²</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>2740 μm²</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4470 μm²</td>
</tr>
<tr>
<td>LUMENCACHE</td>
<td>4</td>
<td>800 μm²</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1600 μm²</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>3190 μm²</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>6450 μm²</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>12870 μm²</td>
</tr>
</tbody>
</table>

3.5.1 Area Estimation

Table 3.4 shows the estimated area of a range of fully associative TLBs and LUMENCACHES. TLBs are modeled as CAMs using CACTI [30], and LUMENCACHES are modeled as SRAMs and RMCs using CACTI and naïve technology scaling from the 130nm process used in the original RMC paper [27].

As LUMENCACHE entries are more complex than typical page table entries cached by a TLB, the per-entry cost is somewhat larger. For instance, a 16-entry LUMENCACHE is estimated to be between a 32- and 64-entry TLB in area. Similarly, the LUMENCACHE is expected to consume 2-3× as much power per entry as a typical TLB. However, the LUMENCACHE provides a much greater potential reach per entry — up to the size of the entire address space — and much greater flexibility in management of permissions. In a common scenario where only a small number of intervals are in use, a small LUMENCACHE can easily capture nearly every lookup, whereas a much larger TLB may still be unable to effectively capture all regions in use.
3.5.2 Hardware Refill

Though interval insertion and deletion are relatively complex, we specifically designed our ISL and concurrency strategy so that queries are simple, enabling a low-cost hardware LumenCache refill implementation, analogous to the page table walkers found in high-performance systems today. Offloading LumenCache refill to a hardware ISL walker improves energy and performance relative to a software routine, reduces instruction cache pollution, and enables multiple memory operations to proceed in parallel.

The ISL walker must be able to correctly compute the combined permissions of any number of intervals covering an address. Due to partial permissions and the higher priority given to smaller intervals, intervals must be processed from largest to smallest, or vice-versa.\footnote{The problem is analogous to alpha blending in computer graphics, as partially specified permissions are analogous to transparency.} The ISL as proposed by Hanson and Johnson does not guarantee that markers are picked up in any particular order, so all intervals must be sorted before their permissions can be combined. However, we find that in Lumen, intervals in the initial stabbing query will be picked up from largest to smallest as long as the marker lists at each node and forward pointer are sorted from largest to smallest. We therefore add the requirement that all markers be inserted into their respective lists in sorted order; this invariant is easy to maintain, even under concurrency, because of the write locks on marker lists. We prove that intervals are returned from largest to smallest in Appendix A. With this requirement, permissions can be combined on-the-fly during the initial stabbing query, without storing or sorting intervals. The ability to query the ISL without sorting is beneficial even for a software implementation, but is especially important for a hardware walker.

Intervals picked up in the conflict phase may contain duplicates or be out of order, but since we only look for conflicting permissions on subsequent bytes rather than computing permissions exactly, we can compare each interval’s permissions against the first byte’s one by one, without storing or sorting intervals. It is rare for a single memory access to span multiple intervals, so the conflict phase will seldom require any action.

For an $N$-bit address space, an efficient ISL walker requires $13N + \log_2 N$ bits of state ($421/838$ bits for $N = 32/64$) and a modest amount of address
Table 3.5: Required state for hardware traversal of a concurrent ISL.

<table>
<thead>
<tr>
<th># Bits</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Search Key</td>
</tr>
<tr>
<td>$N$</td>
<td>Pointer to ISL structure</td>
</tr>
<tr>
<td>$N$</td>
<td>Pointer to ISL header node</td>
</tr>
<tr>
<td>$N$</td>
<td>Pointer to per-thread epoch structure</td>
</tr>
<tr>
<td>$2N$</td>
<td>Temporary Variables</td>
</tr>
<tr>
<td>$N$</td>
<td>Pointer to current node</td>
</tr>
<tr>
<td>$\log(N)$</td>
<td>Current level</td>
</tr>
<tr>
<td>$N$</td>
<td>Current node’s key</td>
</tr>
<tr>
<td>$2N$</td>
<td>Lock addresses for hand-over-hand locking</td>
</tr>
<tr>
<td>$N$</td>
<td>Currently computed interval descriptor state (N bits in our implementation)</td>
</tr>
<tr>
<td>$2N$</td>
<td>Low and high address of last encountered marker (optional, to detect pyramid constraint violations)</td>
</tr>
</tbody>
</table>

generation and state machine logic. The required state variables are listed in Table 3.5. Due to its small overhead, the walker is suitable even for small embedded or accelerator cores where its energy and instruction cache benefits are even more important.
CHAPTER 4
EVALUATION

4.1 Address Space Complexity

To get a sense for Lumen’s performance when being used as a SASOS in a real system, we gathered detailed data on the number and nature of virtual memory intervals on a variety of real machines. We wrote a Linux kernel module to gather system-wide snapshots of user- and kernel-space memory maps by traversing kernel data structures and page tables. The results, shown in Table 4.1 and Figure 4.1, are averaged across periodic snapshots taken every minute for a period of one day. The main result is that Lumen can store the same permissions information as virtual memory in \(2.5 \times 10^{-5}\) less space, due to its hierarchical, non-redundant nature. Also of note is that 25–75% of user-space virtual memory areas system-wide are duplicates, due to common patterns like sharing code and reading common system files.

4.2 ISL Characterization

As ours is the first concurrent ISL implementation we are aware of, we characterize its insertion, deletion, and query behavior under concurrent accesses. We implemented Lumen in C, using Concurrency Kit [31] as a portable interface to platform-specific synchronization primitives, atomic operations, memory barriers, and concurrent data structures. As interval insertion may require memory allocation, we use nedmalloc [32], a fast parallel malloc() implementation with per-thread caches. Our experiments are run on a server with 4 Intel Xeon E7-4860 processors running at 2.27 GHz. Each socket has 10 cores with two-way simultaneous multithreading, for a total of 40 cores and 80 hardware threads. The server runs Redhat Enterprise Linux 6.3,
Table 4.1: Test machine configurations for a desktop (*DT*), laptop (*LT*), workstation (*WS*), cloud server (*CS*), and smartphone (*SP*), all running Linux. We characterize the address space complexity of these machines, averaged over a long period of normal use, and compare virtual memory to LUMEN in terms of storage overhead. We only count those parts of virtual memory data structures necessary to provide protections identical to LUMEN: non-leaf page table entries (PTEs), 1 byte for permissions on leaf PTEs, 9 pointers per *mm_struct*, and 3 pointers per *vm_area_struct*. The Linux kernel keeps one *mm_struct* per task and one *vm_area_struct* per distinct virtual memory area (VMA). LUMEN can provide identical protection using an average of 6.2× less memory.

<table>
<thead>
<tr>
<th>Machine</th>
<th><em>DT</em></th>
<th><em>LT</em></th>
<th><em>WS</em></th>
<th><em>CS</em></th>
<th><em>SP</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>3.2.0 64b</td>
<td>3.2.0 64b</td>
<td>2.6.32 64b</td>
<td>3.0.0 32b</td>
<td>3.0.31 32b (Android 4.1)</td>
</tr>
<tr>
<td>RAM</td>
<td>16GB</td>
<td>4GB</td>
<td>64GB</td>
<td>512MB</td>
<td>512MB</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel i7-2600K</td>
<td>Intel T7500</td>
<td>4× Intel L7555</td>
<td>Intel L5520 (Virtual)</td>
<td>ARM Cortex A8</td>
</tr>
<tr>
<td>Frequency</td>
<td>3.7 GHz</td>
<td>2.2 GHz</td>
<td>1.87 GHz</td>
<td>2.27 GHz</td>
<td>1 GHz</td>
</tr>
<tr>
<td>Cores/Threads</td>
<td>4/8</td>
<td>2/2</td>
<td>32/64</td>
<td>4/4</td>
<td>1/1</td>
</tr>
</tbody>
</table>

Characterization

| Processes | 277 | 224 | 262 | 123 | 143 |
| User/Kernel Threads | 630/105 | 608/70 | 134/213 | 169/47 | 626/92 |
| Unique Open Files | 9222 | 9471 | 723 | 678 | 979 |
| Unique/Total VMAs | 16353/52459 | 10798/40589 | 1862/3651 | 1782/5157 | 8757/16126 |
| Pg. Tbl. Prot. Storage | 35.26MB | 19.11MB | 5.15MB | 1.30MB | 3.11MB |
| LUMEN Storage | 4.65MB | 3.03MB | 525KB | 258KB | 1.263MB |
| LUMEN Strg. Savings | 7.59× | 6.30× | 9.82× | 4.77× | 2.46× |
Figure 4.1: Virtual memory areas (VMAs) across kernel- and user-space. The fundamental address space complexity varies by a factor of five between a smartphone and a desktop. Most regions are duplicates.
which in turn uses the 2.6.32 Linux kernel.

In a multithreaded experiment, threads are first distributed one per core within a single socket, then one per core on the other three sockets, then on the other thread on all 40 cores. Thus, experiments with fewer than ten threads will behave as though run on a single socket machine with single-threaded cores. Experiments with 11 to 40 threads will experience non-uniform memory access (NUMA) effects from running across multiple sockets, but will not experience the resource contention effects of multithreading. Experiments with 41 to 80 threads will experience both NUMA and multithreading effects.

In these experiments, we evaluate a userspace, pure software implementation of LUMEN with no hardware caching or ISL traversal support. Performance would be better and more consistent in a hardware-supported, kernel-mode implementation of LUMEN. For each experiment, we present the median of seven runs.

Inspired by the methodology used by Sundell and Tsigas to characterize a concurrent skip list [18], we vary the number of threads concurrently accessing the ISL, the mix of insertion, removal, and query operations these accessors perform, and the size of the ISL. In each experiment, a set of intervals is randomly generated such that each address in the address space is covered by an average of four intervals. Some intervals are inserted into the ISL in the setup phase to set the initial size of the ISL, and the remainder of the intervals are distributed equally among the accessor threads. All accessors are assigned two weights that govern the frequency of query and insert/remove operations; insertions and removals are assigned the same weight to keep the size of the ISL roughly constant throughout the experiment. The accessors then perform ISL operations in a tight loop in a randomized but valid order — an interval can only be removed after it has been inserted — according to the given weights. Note that, for performance-oriented applications of LUMEN, interval insertion and removal are rare, and the LUMENCACHE captures most stabbing queries; thus, even in highly parallel systems, the ISL will often have a small number of concurrent accessors.

Figure 4.2 shows the ISL’s performance for a workload consisting of 10% insertions, 10% removals, and 80% queries. Such an operation mix is representative of a heavyweight debugging, security, or reliability application of LUMEN, where intervals are inserted very frequently for fine-grained opera-
tions like function calls. Insertions and removals take roughly 4000–32000 cycles in the single-socket experiments; insertions take slightly longer than removals, as shown in Figure 4.2d, because they allocate memory on the heap. In the single-socket experiments, queries take 500–10000 cycles, depending on thread count and ISL size; query time grows logarithmically with ISL size as expected. The relatively small latency difference between 1, 2, 4, and 8 threads for all but the smallest ISLs suggests that Lumen’s concurrency scheme is effective in maximizing throughput under many concurrent accessors. This result is promising evidence for Lumen’s potential as a high performance memory protection scheme for highly parallel systems.

The results for 16–80 threads in Figure 4.2 shows that sharing a single ISL across sockets significantly decreases performance. For example, query latency for a 16k interval ISL only grows by 1.77× from 1 to 8 threads on a single socket, but grows dramatically once the ISL is split across two or four sockets; query latency grows by 2.46× from 8 to 16 threads, then again by 9.24× from 16 to 32 threads. These results echo previous findings that the QuickPath Interconnect (QPI) between sockets is a bottleneck for programs with fine-grained data sharing; Li et al. [33], using a similar quad-socket system, found that accessing memory on a remote socket over QPI had 2.3×–4.7× lower bandwidth and 24%–56% higher latency than local DRAM. These multi-socket effects are complex and merit closer examination in future work.

Figure 4.3 shows the ISL’s performance for 0.1% insertions, 0.1% removals, and 99.8% queries. This operation mix is more representative of typical performance-oriented use cases for Lumen, where intervals are only used for per-process code, stack, heap, and mmap()ed regions.

4.3 System Performance

The total overhead imposed by Lumen on applications is a function of the underlying Lumen data structure, the effectiveness of hardware interval caching, address space complexity, and memory access patterns. The long application runtimes required to characterize Lumen for large, memory-intensive applications make cycle-accurate simulation prohibitive. We therefore characterize Lumen’s overhead by instrumenting applications running
Figure 4.2: ISL insert, remove, and query performance for 1–80 concurrent threads performing 10% insertions, 10% removals, and 80% queries.
Figure 4.3: ISL insert, remove, and query performance for 1–80 concurrent threads performing 1% insertions, 1% removals, and 98% queries.
natively on real hardware. We wrote a Pin [34] tool to maintain and query a shadow ISL for single- or multi-threaded applications, while simulating several sizes of per-core interval caches.

The pintool maintains a set of coarse-grained protection intervals, equivalent to what is provided by the baseline page-based protections, by parsing /proc/self/maps at startup and intercepting successful mmap, munmap, mremap, mprotect, and brk system calls. Intervals are inserted and removed from the ISL as appropriate to maintain an accurate map of the process’ address space. The tool instruments all memory reads and writes in the application, including all libraries, to query a simulated interval cache. If the access misses in the interval cache, the tool queries the ISL in software and fills an entry into the cache. All insertions, removals, and queries are timed to provide an accurate overhead estimate. We divide the total insertion, removal, and query time by the uninstrumented runtime of the application to determine an estimated runtime overhead of Lumen with a LumenCache.

We ran the Parsec 3.0 [35] parallel benchmark on the 80-core server, using one, two, and four threads and the simlarge inputs. We found that, when Lumen was configured to provide the same protections as the baseline operating system, even a 4-entry LumenCache yielded a hit rate of over 99.9% and a performance overhead of well under 1% for all benchmarks. Note that a 4-entry LumenCache consumes much less power and area than the per-core TLBs found in modern architectures. We then extended the pintool to a debugging scenario where Lumen also inserted and removed intervals for every malloc() and free() call; for the Parsec benchmarks, an 8-entry LumenCache sufficed to yield hit rates of over 99% and performance overheads between 0.1% and 2.5×. We leave a more detailed study of the system-level overhead of Lumen in performance-, security-, and debugging-oriented scenarios to future work.
Numerous applications for region-tracking systems in serial and parallel debugging and taint tracking have been proposed in the literature [22, 23]. We list some additional applications and possible extensions here:

**Stack Protection**  Runtime stack protection can be implemented with Lumen as follows: In each function prologue, a thread creates an interval covering the current stack frame; the interval should be read-write for the calling thread, and inaccessible to other threads, except in special circumstances where stack-allocated variables are temporarily shared among threads. At each function call, the thread sets its own stack frame interval to inaccessible before jumping to the callee, so that the callee cannot read or clobber parent stack frames. At each function return, the caller removes its stack frame interval from the ISL, jumps to the caller, and the caller re-enables read-write access on the caller’s stack frame interval.

If this is a common use case, we can augment Lumen interval descriptors to include an explicit end address, rather than encode the end address explicitly by the pattern of interval markers in the ISL. With this extension, the end address can be manipulated in a single operation, greatly accelerating permissions operations on contiguous intervals like stacks which grow and shrink frequently. Function prologues and return points would only need to modify the stack interval’s endpoint, rather than insert and remove a new interval for every function call.

**Heap Protection**  Dynamic memory allocation is prone to programmer error. Numerous tools have been developed to allow memory allocations and accesses to be monitored, but incur a high runtime and memory overhead. Lumen can be used to implement similar capabilities, such as detecting double-free, use-after-free, and buffer overflows, by creating fine-grained
regions for dynamically allocated memory, marking unallocated heap regions as inaccessible by default. Runtime virtual-memory-based tools like DUMA [36] cause memory fragmentation by using at least 2 pages for each allocation. Binary instrumentation and translation frameworks like Valgrind [37] impose a runtime overhead of 2–50×. Compile-time tools like Address Sanitizer (ASan) [38] have smaller runtime overhead, but have substantial memory overhead and cannot be used without recompiling from source. **LUMEN** can provide similar capabilities to ASan with lower overhead by leveraging the **LUMENCACHE**. Our results in Section 4 include an instrumented thread-caching heap allocator that inserts an ISL interval for every heap-allocated region.

**Prefetching and Profiling** The **LUMENCACHE** and **LUMEN** ISL can be augmented to store arbitrary metadata about memory regions, in addition to permissions. For example, extra metadata may be used to efficiently count the number of memory accesses or instruction fetches in a range of addresses, or to produce targeted hardware prefetches when certain addresses are touched. Somogyi et al. [39] discuss the prefetching benefits of correlating spatial and temporal access information, and **LUMEN** can efficiently and flexibly track spatial properties of an application’s access stream. We leave further exploration of these opportunities to future work.
Our focus on scalability when developing LUMEN is in line with a trend with a growing concern for concurrency in shared system data structures. For example, Clements et al. [40] noticed that the Linux kernel’s per-process red-black trees of virtual memory regions, guarded by a single read/write lock, limited the scalability of their parallel benchmark suite on an 80-core machine. They improved application performance by 1.7–3.4× by replacing the red-black tree with a lock-free balanced tree using read-copy-update (RCU) synchronization [41].

Single address space operating systems (SASOS) were motivated by many of the same concerns as LUMEN as well as the decreased scarcity of virtual addresses in 64-bit systems. A SASOS runs all threads in the system in a single flat address space and implements a memory management scheme to share or protect data as directed by applications. Vochteloo [42] provides an overview of SASOS memory protection schemes. Most SASOSs either use capabilities — bit-vectors carried around as proof of a thread’s permissions to an object [43] — or rely on the page table machinery in existing systems to enforce protection, with the same redundancy and granularity pitfalls as virtual memory [44]. A capability-based SASOS can either provide probabilistic protection by hiding data in a very large virtual address space, or provide complete protection using an intermediary layer of kernel-mode software to validate capabilities before granting access. LUMEN provides complete protection by mediating all accesses through the ISL, but does not use explicit capabilities.

Published SASOSs tend not to allow hierarchical permissions or sharing, and store memory metadata in simple data structures not designed for frequent concurrent accesses, such as B+ trees [42]. LUMEN’s more general representation for memory metadata allows arbitrary hierarchy, improving the performance of the ISL by reducing redundancy and allowing application and
library developers to lay out data to maximize locality and code simplicity, rather than to satisfy the memory protection system’s requirements. Further, LUMEN’s ISL has been designed explicitly for high performance under concurrent lookups and modifications, making it more suitable for modern systems with many hardware threads.

Witchel et al. [8] proposed Mondrian Memory Protection (MMP), a high-performance word-granularity permissions scheme for a single virtual or physical address space. While LUMEN grants permission on memory regions to sets of threads, MMP grants permission on memory regions to protection domains as defined by Lampson [45], which in turn map to static instructions in running code; that is, a thread is in exactly one protection domain at a time, and its protection domain depends on what code it’s executing. While LUMEN’s and MMP’s protection domain models are very different on the surface, they are isomorphic to one another, and the primary fundamental differences between the two schemes are in the data structure and permissions semantics.

MMP uses an interval-based data structure, the multi-level permissions trie (MLPT), but it is still a flattened representation, and shared regions require duplicate entries across each sharer’s MLPT, incurring the same overhead and semantic disadvantages as virtual memory based protection. Mondrix, an extension of MMP implemented in the Linux kernel, introduces group protection domains, which eliminate this MLPT redundancy if multiple protection domains require identical permissions [46]. MMP is also designed for uniprocessors and the authors make no mention of concurrent accesses. MMP and LUMEN both use a CAM-based lookaside buffer to cache permissions. MMP also caches the interval returned from a lookaside buffer access alongside the pointer-containing register that triggered the access. These sidecar registers elide lookaside buffer accesses when a pointer register value is used to access an interval more than once, but must be kept coherent when permissions change. LUMEN implementations may adopt sidecars when profitable. Whereas MMP’s storage format is dense and optimized for permissions data only, LUMEN is designed to be extensible, allowing arbitrary metadata to be attached to interval descriptors for debugging, profiling, prefetching, or other purposes.

Loki tracks permissions data for an address space by tagging every word with a 32-bit identifier that may be used to ascertain the permissions an
accessing thread has on the word [1]. The 100% storage overhead of a naïve Loki implementation can be overcome for allocating a single tag for large regions of memory with the same identifier; it also bears the composable and performance disadvantages of a flattened permissions representation. MemTracker is a debugging-oriented hardware extension that embeds a programmable state machine in the processor core [47]; separate state information is maintained for each memory word, and the programmable engine computes state transitions on every memory access. Different state machines can be used to find several classes of common memory bugs, and MemTracker performs better than MMP when configured for such a task, but is unsuitable as a security, reliability, or memory sharing mechanism. Sentry adds a metadata cache alongside the L1 data cache that can cache permissions or other metadata at a cache line granularity [7]. Since it operates at a cache line granularity, Sentry can rely on existing cache coherence machinery to keep metadata in sync across the system, and the metadata only needs to be checked once when it is loaded into the cache. However, this granularity still imposes restrictions on the size and alignment of memory regions that can be protected, the permissions metadata is still flattened, and the authors do not discuss Sentry’s performance under concurrent accesses. Table 6.1 compares LUMEN against these proposed solutions and the status quo multi-process and multi-threaded protection models; LUMEN offers the most flexibility to software and the strictest protection, and its ISL and LUMENCACHE offer high performance for a variety of workloads and system goals as shown in Chapter 4.

The Intel iAPX 432 embodied several bold ideas in hardware support for safe, high-level programming, including a large number of segments (up to
2^{24}) and capability-based addressing [48]. The iAPX 432’s philosophy of allocating distinct program objects to different, arbitrary-length segments to provide stricter protection than virtual memory is in line with LUMEN’s goals. However, flat, statically defined segments carry all the disadvantages discussed in Chapter 2.

Ternary CAMs (TCAMs) have been used to implement variable-page-size TLBs [49] and MMP’s Permissions Lookaside Buffer (PLB). A group of TCAM cells can match any power-of-2 sized, naturally aligned block. Since LUMEN allows intervals of any size and alignment to impose minimal constraints on software, a TCAM-only LUMENCACHE entry could store the largest naturally aligned subset of the interval, as is done in the PLB. However, consider the interval \([0x01 -- 0x2E]\) (\([00000001_2 -- 00101110_2]\)). The largest naturally aligned subset of that interval is \(0x10 -- 0x1F\), which could be represented by a TCAM entry storing \(0001XXXX_2\); the subset only covers \(\frac{16}{46} \approx 35\%\) of the interval, whereas 2 8-bit RMBs can cover the entire interval. By incorporating range-matching cells into the LUMENCACHE, we increase its reach per entry by up to \(3 \times\) over TCAM-only structures like the PLB. For a given workload, however, the distribution of interval alignments may lend itself to a LUMENCACHE consisting of only TCAM entries, some TCAM and some RMB entries, or hybrid entries with some TCAM bits and some RMB bits.
CHAPTER 7

CONCLUSION

LUMEN is an effective technique for providing fine-grained arbitrary protection to regions of memory within a single address space. It removes the need to trade off protection granularity and performance, and the dichotomy between multi-thread and multi-process threading models forced by conventional virtual memory based protection. LUMEN intervals may be stored, modified, and retrieved efficiently, and overhead can be reduced further via the LUMENCACHE. Future systems can use LUMEN to store other types of memory region metadata for performance, profiling, debugging.
APPENDIX A

INTERVAL SKIP LIST STABBING QUERY
RESULT ORDER

As discussed in Section 3.2, Lumen’s semantics dictate that, when an address is covered by multiple intervals, properties specified by smaller intervals take precedence over those specified by larger intervals. By preventing arbitrarily overlapping intervals, which are permitted in the ISL as described by Hanson and Johnson, Lumen’s pyramid constraint dictates a unique, unambiguous precedence ordering for all intervals covering a given address, decreasing in priority from smallest to largest. This constraint greatly simplifies the reduction operation needed to compute the properties of an address, allowing it to use a single register in the case of simple, fixed width metadata like access permissions. The traversal procedure iterates over the covering intervals from largest to smallest, storing the metadata for each interval in turn into the same register; later, smaller intervals will override the metadata set by earlier, larger intervals if they conflict.

A naïve implementation would therefore need to store all covering intervals while the ISL is traversed, sort them by size, then perform the reduction. This implied storage and computation overhead is troublesome for a software implementation of Lumen, but even more so for a hardware ISL walker, since the overhead is unbounded in the general case. A hardware walker would need to either allocate a fixed amount of storage and trap to a software implementation for queries with many covering intervals, or store intervals in main memory; either option significantly decreases the utility of a hardware walker and the viability of Lumen as a mainstream memory protection scheme.

Fortunately, by exploiting the additional information about interval placement due to the pyramid constraint, we can modify the interval insertion procedure slightly to guarantee that intervals will be visited in sorted order in Lumen. This guarantee allows constant-space traversals, and admits a simple hardware implementation that can handle all queries. We show that
Figure A.1: The pyramid constraint and interval sorting constraint ensure that an ISL stabbing query will return the intervals enclosing an address from largest to smallest. Suppose that a stabbing query on address 17 returns 5 covering intervals $a$–$e$. We show that the pyramid constraint implies that $a \supset d$, $b \supset d$, $c \supset d$, and $d \supset e$ (Case 1). We impose the interval sorting constraint to require that $a \supset c$ and $b \supset c$ (Case 2), and $a \supset b$ (Case 3). These containment relationships constitute a total order on all intervals returned by a stabbing query.

this guarantee is satisfied for the three cases shown in Figure A.1.

First, we show that interval markers from a forward pointer covering a given interval are larger than intervals picked up on any subsequent forward pointers covering a different interval. This situation is labeled as Case 1 in Figure A.1. More formally, consider a query for address $Q$ returning a set of intervals $X$. We can show that if $Q$ inserts into $X$ an interval $a = [a.L, a.R]$ on a forward pointer $FP_m$, then later inserts an interval $b = [b.L, b.R]$ on a subsequent forward pointer $FP_n$ covering an interval $FP_n.I = [FP_n.I.L, FP_n.I.R]$, where $FP_m$ and $FP_n$ cover different intervals, then $a \supset b$ (i.e., $L_a \leq L_b \leq R_b \leq R_a \land a \neq b$).

**Definition** Let $X$ be set the set of intervals covering an address $Q$ returned by an ISL query. The elements of $X$ are in the order they were encountered by the query. Let $a$ and $b$ be two of the returned intervals, $a = X_i = [a.L, a.R], b = X_j = [b.L, b.R], i < j$. The markers for $a$ and $b$ were attached to forward pointers $FP_m$ and $FP_n$, respectively. $a \in FP_m, b \in FP_n, FP_m.I = [FP_m.I.L, FP_m.I.R], FP_n.I = [FP_n.I.L, FP_n.I.R]$. 

**Lemma A.0.1.** Suppose $FP_m.I \neq FP_n.I$. Then $|a| > |b|$. 

46
Proof. 1. \( a \in X, b \in X \implies Q \in a, Q \in b \) because ISL queries only return intervals containing the query point.

2. \( Q \in a, Q \in b \implies a \cap b \neq \emptyset; a \) and \( b \) overlap.

3. \( i < j \land m \neq n \implies FP_m.level > FP_n.level \) because ISL queries traverse levels from highest to lowest, and never pick up markers from more than one forward pointer on a given level.

4. \( a \in FP_m \implies a \supseteq FP_m.I \) because of the ISL containment constraint.

5. \( i < j \implies FP_m.I.L \leq FP_n.I.L \) because ISL queries traverse nodes in nondecreasing key order.

6. \( FP_n.I.L \leq FP_m.I.R \) because if \( FP_n.I.L > FP_m.I.R \), then \( FP_m \) would have been traversed and \( a \) would not have been picked up.

7. \( FP_m.I.L \leq FP_n.I.L \leq FP_n.I.R \implies FP_m.I \cap FP_n.I \neq \emptyset, \neg(FP_n.I \supset FP_m.I) \).

8. \( FP_m.I \cap FP_n.I \neq \emptyset \land \neg(FP_n.I \supset FP_m.I) \land FP_m.I \neq FP_n.I \implies FP_m.I \supset FP_n.I \), because the structure of ISL nodes and pointers implies that the intervals covered by two forward pointers either do not overlap, are equal, or one is a proper subset of the other, similar to the pyramid constraint we impose on intervals under Lumen.

9. \( b \in FP_n \land FP_m \supset FP_n \implies \neg(b \supseteq FP_m.I) \implies \neg(b \in FP_m) \) because of the ISL maximality constraint.

10. \( a \supseteq FP_m.I \land \neg(b \supseteq FP_m.I) \implies \neg(b \supseteq a) \).

11. \( a \cap b \neq \emptyset \land \neg(b \supseteq a) \implies a \supset b \) due to the pyramid constraint; if two intervals overlap, one must be a proper superset of the other.

12. \( a \supset b \implies ((a.R - a.L) > (b.R - b.L)) \). \( \blacksquare \)

We have shown that LUMEN’s pyramid constraint and the ISL containment and maximality conditions guarantee that interval markers from forward pointers covering different intervals are returned by queries in sorted order, from largest to smallest. A pair of returned intervals may also be
from different forward pointers covering the same interval or from the same forward pointer, as shown in cases 2 and 3 of Figure A.1, respectively. We extend the property of Lemma A.0.1 to these other two cases by requiring that intervals across all forward pointers covering the same interval be inserted in sorted order. With this requirement, which we call the interval sorting constraint (ISC), the intervals in the latter two cases will be traversed by the query and inserted into the output array $X$ in sorted order. The ISC is easily implemented under our proposed concurrency scheme, since all forward pointers covering the same interval will have the same left and right endpoint nodes, and are thus covered by a single read-write lock. The constraint adds time overhead to insertions by forcing them to traverse marker lists to find the appropriate insertion point rather than insert at the beginning of the list. In our implementation where marker lists are linked lists, this overhead is linear, but may be reduced to logarithmic by implementing array-, tree-, or skip list-based marker lists. Forward pointers tend to have few interval markers, and multiple forward pointers covering the same interval are relatively rare, significantly mitigating the performance impact of the ISC. Through the pyramid constraint, ISL invariants, and ISC, we can guarantee that all intervals returned by a query will be sorted from smallest to largest, allowing a hardware ISL walker to perform the permissions reduction operation in constant space. Enabling a simple hardware walker implementation allows the computational overhead and instruction cache impact of ISL traversal to be offloaded from the processor pipeline, and makes LUMEN’s flexibility benefits available to mainstream systems where LUMEN’s performance must be comparable to virtual memory-based protection to be viable.
REFERENCES


