SpaceJMP: Programming with Multiple Virtual Address Spaces

Izzat El Hajj[3,4,*], Alexander Merritt[2,4,*], Gerd Zellweger[1,4,*], Dejan Milojicic[4],
Reto Achermann[1], Paolo Faraboschi[4], Wen-mei Hwu[3], Timothy Roscoe[1], and Karsten Schwan[2]

[1]Department of Computer Science, ETH Zürich [2]Georgia Institute of Technology
[3]University of Illinois at Urbana-Champaign [4]Hewlett Packard Labs

[*]Co-primary authors

Abstract

Memory-centric computing demands careful organization of the virtual address space, but traditional methods for doing so are inflexible and inefficient. If an application wishes to address larger physical memory than virtual address bits allow, if it wishes to maintain pointer-based data structures beyond process lifetimes, or if it wishes to share large amounts of memory across simultaneously executing processes, legacy interfaces for managing the address space are cumbersome and often incur excessive overheads.

We propose a new operating system design that promotes virtual address spaces to first-class citizens, enabling process threads to attach to, detach from, and switch between multiple virtual address spaces. Our work enables data-centric applications to utilize vast physical memory beyond the virtual range, represent persistent pointer-rich data structures without special pointer representations, and share large amounts of memory between processes efficiently.

We describe our prototype implementations in the DragonFly BSD and Barrelfish operating systems. We also present programming semantics and a compiler transformation to detect unsafe pointer usage. We demonstrate the benefits of our work on data-intensive applications such as the GUPS benchmark, the SAMTools genomics workflow, and the Redis key-value store.

1. Introduction

The volume of data processed by applications is increasing dramatically, and the amount of physical memory in machines is growing to meet this demand. However, effectively using this memory poses challenges for programmers. The main challenges tackled in this paper are addressing more physical memory than the size of a virtual space, maintaining pointer-based data structures across process lifetimes, and sharing very large memory objects between processes.

We expect that main memory capacity will soon exceed the virtual address space size supported by CPUs today (typically 256 TiB with 48 virtual address bits). This is made particularly likely with non-volatile memory (NVM) devices – having larger capacity than DRAM – expected to appear in memory systems by 2020. While some vendors have already increased the number of virtual address (VA) bits in their processors, adding VA bits has implications on performance, power, production, and estate cost that make it undesirable for low-end and low-power processors. Adding VA bits also implies longer TLB miss latency, which is already a bottleneck in current systems (up to 50% overhead in scientific apps [55]). Processors supporting virtual address spaces smaller than available physical memory require large data structures to be partitioned across multiple processes or these structures to be mapped in and out of virtual memory.

Representing pointer-based data structures beyond process lifetimes requires data serialization, which incurs a large performance overhead, or the use of special pointer representations, which results in awkward programming techniques. Sharing and storing pointers in their original form across process lifetimes requires guaranteed acquisition of specific VA locations. Providing this guarantee is not always feasible, and when feasible, may necessitate mapping datasets residing at conflicting VA locations in and out of the virtual space.

Sharing data across simultaneously executing processes requires special communication with data-serving processes (e.g., using sockets), impacting programmability and incurring communication channel overheads. Sharing data via traditional shared memory requires tedious communication and synchronization between all client processes for growing the shared region or guaranteeing consistency on writes.

To address all these challenges, we present SpaceJMP, a set of APIs, OS mechanisms, and compiler techniques that constitute a new way to manage memory. SpaceJMP applications create and manage Virtual Address Spaces (VASes) as first-class objects, independent of processes. Decoupling VASes from processes enables a single process to activate multiple VASes, such that threads of that process can switch

ASPLOS '16 April 02 - 06, 2016, Atlanta, GA, USA

ACM 978-1-4503-4091-5/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2872362.2872366

between these VASes in a lightweight manner. It also enables a single VAS to be activated by multiple processes or to exist in the system alone in a self-contained manner.

SpaceJMP solves the problem of insufficient VA bits by allowing a process to place data in multiple address spaces. If a process runs out of virtual memory, it does not need to modify data mappings or create new processes. It simply creates more address spaces and switches between them. SpaceJMP solves the problem of managing pointer-based data structures because SpaceJMP processes can always guarantee the availability of a VA location by creating a new address space. SpaceJMP solves the problem of data sharing by allowing processes to switch into a shared address space. Clients thus need not communicate with a server process or synchronize with each other on shared region management, and synchronization on writes can be lightweight.

SpaceJMP builds on a wealth of historical techniques in memory management and virtualization but occupies a novel "sweet spot" for modern data-centric applications: it maps well to existing hardware while delivering more flexibility and performance for large data-centric applications compared with current OS facilities.

We make the following contributions:

1. We present SpaceJMP (Section 3): the OS facilities provided for processes to each create, structure, compose, and access multiple virtual address spaces, together with the compiler support for detecting unsafe program behavior.
2. We describe and compare (Section 4) implementations of SpaceJMP for the 64-bit x86 architecture in two different OSes: DragonFly BSD and Barrelfish.
3. We empirically show (Section 5) how SpaceJMP improves performance with microbenchmarks and three data-centric applications: the GUPS benchmark, the SAMTools genomics workflow, and the Redis key-value store.

2. Motivation

Our primary motivations are the challenges we have already mentioned: insufficient VA bits, managing pointer-based data structures, and sharing large amounts of memory – discussed in Sections 2.1, 2.2, and 2.3 respectively.

2.1 Insufficient Virtual Address Bits

Non-Volatile Memory (NVM) technologies, besides persistence, promise better scaling and lower power than DRAM, which will enable large-scale, densely packed memory systems with much larger capacity than today's DRAM-based computers. Combined with high-radix optical switches [76], these future memory systems will appear to the processing elements as single, petabyte-scale "load-store domains" [33].

One implication of this trend is that the physical memory accessible from a CPU will exceed the size that VA bits can address. While almost all modern processor architectures support 64-bit addressing, CPU implementations pass fewer bits to the virtual memory translation unit because of power and performance implications. Most CPUs today are limited to 48 virtual address bits (i.e., 256 TiB) and 44-46 physical address bits (16-64 TiB), and we expect a slow growth in these numbers.

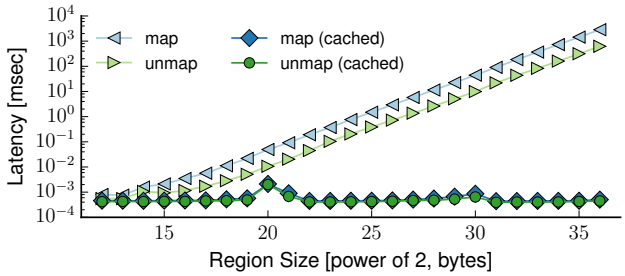


Figure 1: Page table construction (`mmap`) and removal (`munmap`) costs in Linux, using 4KiB pages. Does not include page zeroing costs.

The challenge presented is how to support applications that want to address large physical memories without paying the cost of increasing processor address bits across the board. One solution is partitioning physical memory across multiple OS processes which would incur unnecessary inter-process communication overhead and is tedious to program. Another solution is mapping memory partitions in and out of the VAS, which has overheads discussed in Section 2.4.

2.2 Maintaining Pointer-based Data Structures

Maintaining pointer-based data structures beyond process lifetimes without serialization overhead has motivated emerging persistent memory programming models to adopt region-based programming paradigms [15, 21, 78]. While these approaches provide a more natural means for applications to interact with data, they present the challenge of how to represent pointers meaningfully across processes.

One solution is the use of special pointers, but this hinders programmability. Another solution is requiring memory regions to be mapped to fixed virtual addresses by all users. This solution creates degenerate scenarios where memory is mapped in and out when memory regions overlap, the drawbacks of which are discussed in Section 2.4.

2.3 Sharing Large Memory

Collaborative environments where multiple processes share large amounts of data require clean mechanisms for processes to access and manage shared data efficiently and safely. One approach uses a client-server model whereby client processes communicate with key-value stores [24, 63, 67] that manage and serve the data. This approach requires code to be rewritten to use specific interfaces and incurs communication overhead. Another approach is using shared memory regions, but current OS mechanisms for doing so have many limitations discussed in Section 2.4.

2.4 Problems with Legacy Methods

Legacy methods for interacting with memory are based on interfaces exposed over a *file* abstraction. As systems become increasingly memory centric, these interfaces introduce performance bottlenecks that limit scalability [35], and their complexity and usability create challenges for programmers.

Changing memory maps in the critical path of an application has significant performance implications, and scaling to large memories further exacerbates the problem. Using file-based methods such as `mmap` is slow and not scalable [20]. Figure 1 shows that constructing page tables for a 1 GiB region using 4 KiB page takes about 5 ms; for 64 GiB the cost is about 2 seconds. Frequent use of such interfaces quickly becomes an expensive operation, especially when applications must access all pages within a region.

In addition to performance implications, the flexibility of legacy interfaces is also a major concern. Memory-centric computing demands careful organization of the virtual address space, but interfaces such as `mmap` only give limited control. Some systems do not support creation of address regions at specific offsets. In Linux, for example, `mmap` does not safely abort if a request is made to open a region of memory over an existing region; it simply writes over it. Moreover, in current systems, memory sharing is tied to coarse-grained protection bits configured on the backing file, such as user-group, or access control lists (ACL). This requires a translation between different security models which may want to operate at different granularities. That along with linkers that dynamically relocate libraries or OSes that randomize the stack allocation prevents applications from sharing memory effectively.

3. Design

We now describe the design of SpaceJMP and define the terminology used in this paper. SpaceJMP provides two key abstractions: *lockable segments* encapsulate sharing of in-memory data, and *virtual address spaces* represent sets of non-overlapping segments. We also describe semantics for safe programming of segments and address spaces.

3.1 Lockable Segments

In SpaceJMP, all data and code used by the system exist within *segments*. The term has been used to refer to many different concepts over the years; in SpaceJMP, a segment should be thought of as an extension of the model used in Unix to hold code, data, stack, etc.: a segment is a single, contiguous area of virtual memory containing code and data, with a fixed virtual start address and size, together with metadata to describe how to access the content in memory. With every segment we store the backing physical frames, the mapping from its virtual addresses to physical frames and the associated access rights.

For safe access to regions of memory, all SpaceJMP segments can be *lockable*. In order to switch into an address space, the OS must acquire a reader/writer lock on each lockable segment in that address space.

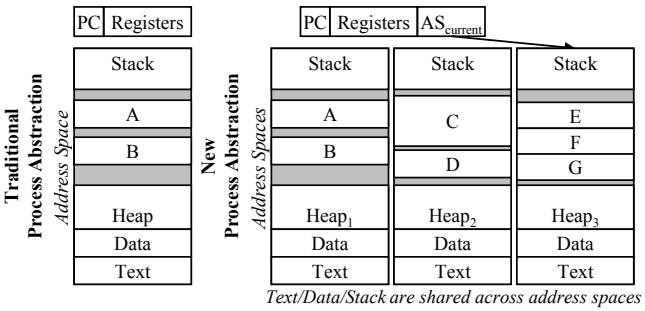


Figure 2: Contrasting SpaceJMP and Unix.

```
VAS API – for applications.
vas_find(name) → vid
vas_detach(vh)
vas_create(name,perms) → vid
vas_clone(vid) → vid
vas_attach(vid) → vh
vas_switch(vh)
vas_ctl(cmd,vid[,arg])

Segment API – for library developers.
seg_find(name) → sid
seg_attach(vh,sid)
seg_detach(vh,sid)
seg_ctl(sid,cmd[,arg])
seg_attach(vid,sid)
seg_detach(vid,sid)
seg_clone(sid) → sid
seg_alloc(name,base,size,perms) → sid
```

Figure 3: SpaceJMP interface.

Each lock acquisition is tied to the access permissions for its corresponding segment: if the segment is mapped read-only, the lock will be acquired in a shared mode, supporting multiple readers (i.e., multiple reading address spaces) and no writers. Conversely, if the segment is mapped writable, the lock is acquired exclusively, ensuring that only one client at a time is allowed in an address space with that segment mapped.

Lockable segments are the unit of data sharing and protection provided in SpaceJMP. Together with address space switching, they provide a fast and secure way to guarantee safe and concurrent access to memory regions to many independent clients.

3.2 Multiple Virtual Address Spaces

In most contemporary OSes, a process or kernel thread is associated with a single virtual address space (VAS), assigned when the execution context is created. In contrast, SpaceJMP virtual address spaces are first-class OS objects, created and manipulated independently of threads or processes. The contrast between SpaceJMP and traditional Unix-like (or other) OSes is shown in Figure 2. SpaceJMP can emulate regular Unix processes (and does, as we discuss in the next section), but provides a way to rapidly switch sections of mappings to make different sets of segments accessible.

The SpaceJMP API is shown in Figure 3. A process in SpaceJMP can *create*, *delete*, and *enumerate* VASes. A VAS can then be *attached* by one or more processes (via `vas_attach`), and a process can *switch* its context between

```
type *t; vasid_t vid; vhandle_t vh; segid_t sid;
```

// Example use of segment API.

```
va=0xC0DE; sz=(1UL«35);
vid=vas_create("v0",660);
sid=seg_alloc("s0",va,sz,660);
seg_attach(vid, sid);
```

// Example use of VAS API.

```
vid=vas_find("v0");
vh=vas_attach(vid);
vas_switch(vh);
t=malloc(...); *t = 42;
```

Figure 4: Example SpaceJMP usage.

different VASes to which it is attached at any point during its execution. A VAS can also continue to exist beyond the lifetime of its creating process (e.g., it may be attached to other processes).

The average user only needs to worry about manipulating VASes. Advanced users and library developers may directly manipulate segments within the VAS. Heap allocators, for instance, must manage the segments that service the allocations (see Section 4). Segments can be *attached* to, or *detached* from a VAS. In practice, some segments (such as global OS mappings, code segments, and thread stacks) are widely shared between VASes attached to the same process. `seg_attach` and `seg_detach` allow installing segments that are process-specific by using the `vid` handle (`vh`) or globally for all processes attached to a VAS using the `vid`.

Rather than reinventing the wheel, our permission model for segments and address spaces use existing security models available in the OS. For example, in DragonFly BSD, we rely on ACLs to restrict access to segments and address spaces for processes or process groups. In Barrelfish, we use the capability system provided by the OS (Section 4.2) to control access. To modify permissions, the user can *clone* a segment or VAS and use the `seg_ctl`, or `vas_ctl`, to change the meta-data of the new object (e.g., permissions) accordingly.

A spawned process will still receive its initial VAS by the OS. The exposed functionality to the user process, however, allows it to construct and use additional VASes. A typical call sequence to create a new VAS is shown in Figure 4.

3.3 Programming Semantics for Safe Execution

Multiple VASes per process is a powerful feature for writing applications in a data-driven world. However, this feature introduces new kinds of unsafe memory access behavior that programmers must carefully avoid. We provide the user with a compiler tool to detect such unsafe behavior.

This section defines semantics for programming with SpaceJMP safely. The *common region* refers to the memory segments that a process maps to all VASes such as the stack, code, and globals. A VAS should not store a pointer that points to another VAS, except in the common region, which is shared by all VASes. Therefore any pointer not in the common region of a VAS is valid only when that VAS is active. Moreover, pointers to the common region should only be stored in the common region because they are not meaningful to other processes that may attach a VAS. This ensures that a VAS remains independent from any process.

The following rules define safe and unsafe behavior:

- If a pointer p is obtained when a VAS v is active via allocation in or loading from the non-common region:
  - Dereferencing p is safe in v and unsafe otherwise
  - Storing to p a pointer q is safe if q points to the non-common region of v and unsafe otherwise
- If a pointer p is obtained via allocation on the stack or is a global or function pointer:
  - Dereferencing and storing to p is always safe
- If a pointer p is loaded from the common region:
  - The safety properties of p must be the same as that of the originally stored pointer

Compiler support for detecting violations of these rules is discussed in Section 4.3.

4. Implementation

We built 64-bit x86 implementations of SpaceJMP for the DragonFly BSD and Barrelfish operating systems, which occupy very different points in the design space for memory management and provide a broad perspective on the implementation trade-offs for SpaceJMP. We first discuss the DragonFly BSD implementation, since it is the most mature and the platform for the majority of results in Section 5.

4.1 DragonFly BSD

DragonFly BSD [27] is a scalable, minimalistic kernel originally derived from FreeBSD4, supporting only the x86-64 architecture. Our SpaceJMP implementation includes modifications to the BSD memory subsystem and an address-space-aware implementation of the user space `malloc` interfaces.

Memory subsystem: DragonFly BSD has a nearly identical memory system to FreeBSD, both of which derive from the memory system in the Mach kernel [65]. The following descriptions therefore also apply to similar systems.

The BSD memory subsystem is based on the concept of *VM objects* [65] which abstract storage (files, swap, raw memory, etc.). Each object contains a set of physical pages that hold the associated content. A SpaceJMP segment is a wrapper around such an object, backed only by physical memory, additionally containing global identifiers (e.g., a name), and protection state. Physical pages are reserved at the time a segment is created, and are not swappable. Furthermore, a segment may contain a set of cached translations to accelerate attachment to an address space. The translations may be globally cached in the kernel if the segment is shared with other processes.

Address spaces in BSD, as in Linux, are represented by two layers: a high-level set of region descriptors (virtual offset, length, permissions), and a single instance of the architecture-specific translation structures used by the CPU. Each region descriptor references a single VM object, to

inform the page fault handler where to ask for physical pages. A SpaceJMP segment can therefore be created as a VM object and added directly to an address space, with minimal modifications made to the core OS kernel implementation.

Sharing segments with other processes is straightforward to implement as they can always be attached to an existing address space, provided it does not overlap a previously mapped region. Sharing an address space is slightly more complicated: as mentioned in Section 3, we only share segments that are also visible to other processes. Doing so allows applications to attach their own private segments (such as their code or their stack) into an address space, before they can switch into it. Those segments would typically heavily conflict as every process tends to have a stack and program code at roughly the same location. The underlying address space representation that BSD uses (the `vmspace` object) always represents a particular instance of an address space with concrete segments mapped. Therefore, sharing a `vmspace` instance directly with other processes will not work. Instead, when sharing an address space, the OS shares just the set of memory segments that comprise the VAS. A process then attaches to the VAS, creating a `vmspace`, which it then can switch into.

A slight modification of the process context structure was necessary to hold references to more than one `vmspace` object, along with a pointer to the current address space. Knowing the active address space is required for correct operation of the page fault handler and system calls which modify the address space, such as the segment attach operation.

Inadvertent address collisions may arise between segments, such as those for stack, heap, and code. For example, attaching to a VAS may fail if a (global) segment within it conflicts with the attaching process' (private, fixed-address) code segments. Our current implementation in DragonFly BSD avoids this by ensuring both globally visible and process-private segments are created in disjoint address ranges.

Runtime library: We developed a library to facilitate application development using the SpaceJMP kernel interface, avoiding complexities such as locating boundaries of process code, globals, and stack for use within an address space across switches. The library performs much of the bookkeeping work involved in attaching an address space to a process: private segments holding the process' program text, globals, and thread-specific stacks are attached to the process-local `vmspace` object by using the VAS handle.

Furthermore, the library provides allocation of heap space (`malloc`) within a specific segment while inside an address space. SpaceJMP complicates heap management since programs need to allocate memory from different segments depending on their needs. These segments may not be attached to every address space of the process, and moreover a call to free up memory can only be executed by a process if it is currently in an address space which has the corresponding segment attached.

To manage this complexity, the SpaceJMP allocator is built over Doug Lea's dlmalloc [48], providing the notion of a memory space (mspace). An mspace is an allocator's internal state and may be placed at arbitrary locations. Our library supports distinct mspaces for individual segments, and provides wrapper functions for `malloc` and `free` which supply the correct mspace instance to dlmalloc, depending on the currently active address space and segment.

VAS switching: Attaching to a global VAS creates a new process-private instance of a `vmspace` object, where the process code, stack, global regions are mapped in. A program or runtime initiates a switch via a system call; the kernel identifies the `vmspace` object specified by the call, then simply overwrites `CR3`[1] with the physical address of the page table of the new VAS. Other registers, such as the stack pointer, are not modified. A kernel may also transparently initiate a switch, e.g., after a page fault, or triggered via any other event.

4.2 Barrelfish

Barrelfish [7] is a research operating system structured as a "multikernel", a distributed system of cores communicating solely via asynchronous messages.

In contrast to BSD, Barrelfish prohibits dynamic memory allocation in the kernel. Instead, Barrelfish has user-space memory servers which allocate physical memory to applications. Each memory region is typed using a capability system inspired by seL4 [31]. Retyping of memory is checked by the kernel and performed by system calls. The security model guarantees that a user-space process can allocate memory for its own page tables (and other kernel-level objects) and frames for mapping memory into the virtual address spaces.

The capability system provides fine-grained access control to resources, allowing processes great flexibility in implementing policies, and safely performs security-relevant actions via explicit capability invocations. Therefore, SpaceJMP is implemented almost entirely in user space with no additional logic added to the kernel: all VAS management operation translate into explicit capability invocations.

We use this functionality to explicitly share and modify page tables without kernel participation. In particular, we implemented a SpaceJMP-service to track created VASes in the system, together with attached segments and attaching processes, similar to the kernel extensions employed in BSD. Processes interact with the user-level service via RPCs.

Upon attaching to a VAS, a process obtains a new capability to a root page table to be filled in with mapping information. Switching to the VAS is a capability invocation to replace the thread's root page table with the one of the VAS. This is guaranteed to be a safe operation as the capability system enforces only valid mappings. Initially, all page tables other than the root of a VAS are shared among the attached processes, allowing easy propagation of updated mappings.

[1] *Control register*—indicates currently active page table on an x86 CPU core.

The Barrelfish kernel is therefore unaware of SpaceJMP objects. To enforce their proper reclamation we can rely on the capability revocation mechanism: revoking the process' root page table prohibits the process from switching into the VAS. The result is a pure user space implementation of SpaceJMP, enabling us to enforce custom policies, such as alignment constraints or selecting alternative page sizes.

4.3 Compiler Support

This section describes the implementation of a compiler tool for detecting unsafe behavior according to the semantics presented in Section 3.3. A trivial solution would be to tag all pointers with the ID of the address space they belong to, and insert checks on the tags before any pointer dereference. However, because checking every pointer dereference is too conservative, we present a compiler analysis to prove when dereferences are safe, and a transformation that only inserts checks where safety cannot be proven statically. These are briefly described in this section while detailed algorithms, optimizations, and evaluation are left for future work.

The analysis begins by finding the potentially active VASes at each program point and the VASes each pointer may be valid in. It produces the following sets of information:

- $VAS_{valid}(p)$: For each pointer p, the set of IDs for VASes that p may be valid in. $VAS_{valid}(p)$ may also contain two special values:
  - v_{common}: p may point to the common region
  - v_{unkown}: the ID of the VAS p may point to is unkown statically
- $VAS_{in}(i)$ and $VAS_{out}(i)$: For each relevant program instruction i, the set of IDs for VASes which may be current before and after i executes.

The instructions relevant to the analysis are shown in Figure 5 along with how they impact the analysis information. We provide an additional instruction *vcast* that enables users to cast pointers across VASes to override the safety rules.

After obtaining these two sets of information, the analysis uses them to identify all load or store instruction that may dereference a pointer in the wrong address space. These include any load or store instruction i that dereferences a pointer p such that at least one of the following three conditions hold:

1. $(|VAS_{valid}(p)| > 1) \vee (VAS_{valid}(p) \ni v_{unkown})$ (The VAS that p points to is ambiguous)
2. $|VAS_{in}(i)| > 1$ (The current VAS when i is executed is ambiguous)
3. $VAS_{valid}(p) \neq VAS_{in}(i)$ (The VAS that p points to may not be the same as the current VAS)

For these instructions, the transformation inserts a check before i that checks if the VAS which p points to matches the currently active VAS.

Instruction	Description	Impact on Analysis
switch v	Switch to VAS v	$VAS_{out}(i) = \{v\}$
x = vcast y v	Cast y to VAS v	$VAS_{valid}(x) = \{v\}$
x = alloca	Stack allocation	$VAS_{valid}(x) = v_{common}$
x = *global*	Global variable	$VAS_{valid}(x) = v_{common}$
x = malloc	Heap allocation	$VAS_{valid}(x) = VAS_{in}(i)$
$x = y$	Copy (arith., casts)	$VAS_{valid}(x) = VAS_{valid}(y)$
x = phi y z ...	Phi instructions	$VAS_{valid}(x) = VAS_{valid}(y) \cup VAS_{valid}(z) \cup ...$
$x = *y$	Loads	$VAS_{valid}(x) = VAS_{in}(i)$ or v_{unkown} if stack dereference
$*x = y$	Stores	No impact
x = foo(y, ...)	Function calls	Update VAS_{valid} of parameters and VAS_{in} at function entry
ret x	Returns	Update VAS_{valid} and VAS_{out} at callsites

Figure 5: SSA Instructions Relevant to Analysis

The analysis information is also used to identify stores that may store pointers to illegal locations. These include any store instruction i of a pointer v to a pointer p such that i does not meet any of the following conditions:

1. $VAS_{valid}(p) = \{v_{common}\}$ (The store is to the common region)
2. $(|VAS_{valid}(p)| = 1) \wedge (VAS_{valid}(p) = VAS_{valid}(v))$ (The stored pointer points within the region it is stored to)

For these instruction, the transformation inserts checks that ensure that either p points to the common region or that p and v both point to the current VAS.

The VASes of pointers used in checking code is obtained by inserting appropriate tracking code where those pointers are defined. VASes of pointers across function boundaries are tracked via a global array. VASes of pointers that escape to the common region are tracked via tagged pointers (using the unused bits of the pointer), though shadow memory may also be used for this purpose.

4.4 Discussion

Current implementations of SpaceJMP are relatively unoptimized, and there are several directions for improvement.

On the compiler analysis side, there are situations where our conservative algorithm will insert unnecessary safety checks which a more involved analysis would elide.

More effective caching of page table structures would be beneficial for performance, but imposes additional constraints on the virtual addresses to be efficient. For example, mapping an 8 KiB segment on the boundaries of a PML4 slot requires 7 page tables[2]. By restricting the virtual addresses of segments, to avoid crossing such high-level page table boundaries, page table structures can be cached more efficiently.

The hardware side is also an interesting source of future work. Modern x86-64 CPUs support tagging of TLB entries

[2] One PML4 table, two tables each of PDPT, PDT, PT.

Name	Memory	Processors	Freq.
M1	92 GiB	2x12c Xeon X5650	2.66 GHz
M2	256 GiB	2x10c Xeon E5-2670v2	2.50 GHz
M3	512 GiB	2x18c Xeon E5-2699v3	2.30 GHz

Table 1: Large-memory platforms used in our study.

Operation	DragonFly BSD		Barrelfish	
CR3 load	130	**224**	130	**224**
system call	357	–	130	–
`vas_switch`	1127	**807**	664	**462**

Table 2: Breakdown of *context switching*. Measurements on *M2* in cycles. Numbers in **bold** are with tags enabled.

with a compact (e.g., 12-bit) *address space identifier* to reduce overheads incurred by a full TLB flush on every address space switch. Our current implementations reserve the tag value zero to always trigger a TLB flush on a context switch. By default, all address spaces use tag value zero. The user has the ability to pass hints to the kernel (`vas_ctl`) to request a tag be assigned to an address space.

The trade-off can be complex: use of many tags can decrease overall TLB coverage (particularly since in SpaceJMP many address spaces share the same translations) and result in lower performance instead of faster context switch time.

Furthermore, this trade-off is hardware-specific. Only a single TLB tag can be current in an x86 MMU, whereas other hardware architectures (such as the "domain bits" in the ARMv7-A architecture [3] or the protection attributes in PA-RISC [81]) offer more flexible specification of translations, which would allow TLB entries to be shared across VAS boundaries. In any case, efficient dynamic use of TLB identifiers is a topic for future work.

5. Evaluation

In this section, we evaluate potential benefits afforded through SpaceJMP using three applications across domains: (i) a single-threaded benchmark derived from the HPCC GUPS [64] code to demonstrate the advantages of fast switching and scalable access to many address spaces; (ii) Redis$_{JMP}$, an enhanced implementation of the data-center object-caching middleware Redis, designed to leverage the multi-address-space programming model and lockable segments to ensure consistency among clients; and (iii) a genomics tool to demonstrate ease of switching with complex, pointer-rich data structures. The applications have not been instrumented with the safety checks described in Section 4.3.

Three hardware platforms support our measurements, code-named *M1*, *M2*, and *M3*, as shown in Table 1. All are dual-socket server systems, varying in total memory capacity, core count, and CPU micro-architecture; symmetric multi-threading and dynamic frequency scaling are disabled. Unless otherwise mentioned, the DragonFly BSD SpaceJMP implementation is used.

5.1 Microbenchmarks

This section explores trade-offs available with our solution for address space manipulation, and further evaluates its performance characteristics as an RPC mechanism (Figure 7). We begin first with an evaluation of VAS modification overheads, followed by a breakdown of context switching costs with and without TLB tagging optimizations.

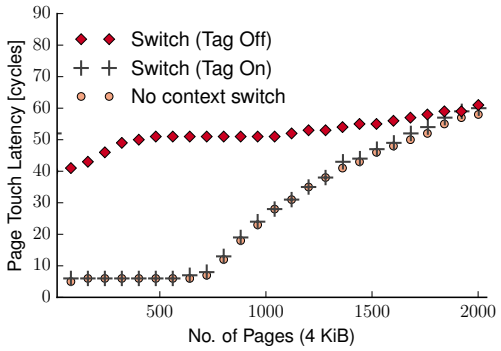


Figure 6: Impact of TLB tagging (*M3*) on a random-access workload. Tagging retains translations, and can lower costs of VAS switching.

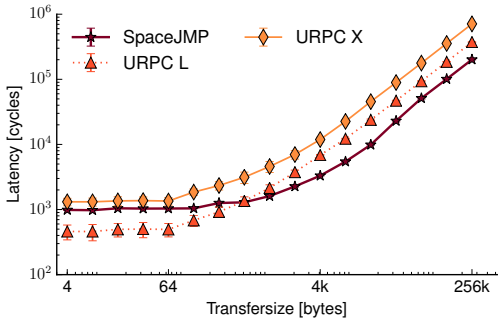


Figure 7: Comparison of URPC and SpaceJMP on Barrelfish (*M2*) as an alternative solution for fast local RPC communication.

Recall from Figure 1 that **page table modification does not scale**, even in optimized, mature OS implementations. The reason is because entire page table subtrees must be created – costs which are directly proportional to the region size and inversely proportional to page size. When restricted to a single per-process address space, changing translations for a range of virtual addresses using `mmap` and `munmap` incurs these costs. Copy-on-write optimizations can ultimately only reduce these costs for large, sparsely-accessed regions, and random-access workloads that stress large areas incur higher page-fault overheads using this technique. With SpaceJMP, these costs are removed from the critical path by switching

translations instead of modifying them. We demonstrate this performance impact on the GUPS workload in Section 5.2.

Given the ability to switch into other address spaces at arbitrary instances, a process in SpaceJMP will cause **context switching** to occur at more frequent intervals than is typical, e.g., due to task rescheduling, potentially thousands of times per second. Table 2 breaks down the immediate costs imposed due to an address space switch in our DragonFly BSD implementation; a system call imposes the largest cost. Subsequent costs are incurred as TLB misses trigger the page-walking hardware to fetch new translations.

While immediate costs may only be improved within the micro-architecture, subsequent costs can be improved with **TLB tagging**.

Notice in Table 2 that changing `CR3` register becomes more expensive with tagging enabled, as it invokes additional hardware circuitry that must consider extra TLB-resident state upon a write. Naturally, these are hardware-dependent costs. However, the overall performance of switching is improved, due to reduced TLB misses from shared OS entries.

We directly measured the impact of tagging in the TLB using a random page-walking benchmark we wrote. For a given set of pages, it will load one cache line from a randomly chosen page. A write to `CR3` is then introduced between each iteration, and the cost in cycles to access the cache line are measured. Lastly, we enable tags, shown in Figure 6. The benefits, however, tail off: with tags, the cost of accessing a cache line reduces to the cost incurred without writes to `CR3` as the working set grows. As expected, benefits gained using tags are limited by a combination of TLB capacity (for a given page size), and sophistication of TLB prefetchers. In our experiment, access latencies with larger working sets match that of latencies where the TLB is flushed.

Finally, we compare the **latency of switching address spaces** in SpaceJMP, with issuing a remote procedure call to another core. In this benchmark, an RPC client issues a request to a server process on a different core and waits for the acknowledgment. The exchange consists of two messages, each containing either a 64-bit key or a variable-sized payload. We compare with the same semantics in SpaceJMP by switching into the server's VAS and accessing the data directly by copying it into the process-local address space. Figure 7 shows the comparison between SpaceJMP and different RPC backends.

Our point of comparison is the (highly optimized) Barrelfish low latency RPC, stripped of stub code to expose only the raw low-level mechanism. In the low-latency case, both client and server busy-wait polling different circular buffers of cache-line-sized messages in a manner similar to FastForward [36]. This is the best-case scenario for Barrelfish RPCs – a real-world case would add overhead for marshalling, polling multiple channels, etc. We see a slight difference between intra (URPC L) and inter-socket (URPC X) performance.

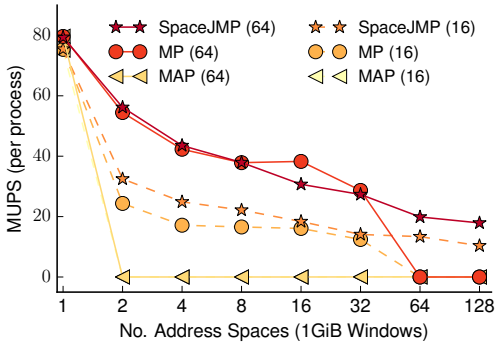


Figure 8: Comparison of three designs to program large memories with GUPS (*M3*). Update set sizes 16 and 64.

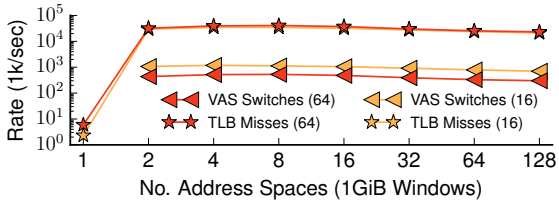


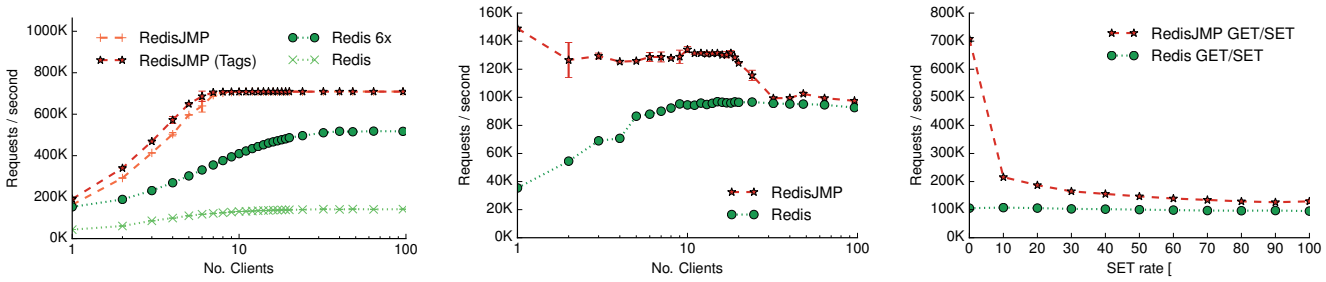
Figure 9: Rate of VAS switching and TLB misses for GUPS executed with SpaceJMP, averaged across 16 iterations. TLB tagging is disabled. A larger window size would produce a greater TLB miss rate using one window.

In all cases, the latency grows once the payload exceeds the buffer size. SpaceJMP is only out-performed by intra-socket URPC for small messages (due to system call and context switch overheads). Between sockets, the interconnect overhead for RPC dominates the cost of switching the VAS. In this case, using TLB tags further reduced latency.

5.2 GUPS: Addressing Large Memory

To deal with the limits of addressability for virtual memory, applications adopt various solutions for accessing larger physical memories. In this section, we use the GUPS benchmark [64] to compare two approaches in use today with a design using SpaceJMP. We ask two key questions: (i) how can applications address large physical memory regions, and (ii) what are the limitations of these approaches?

GUPS is appropriate for this: it measures the ability of a system to scale when applying random updates to a large in-memory array. This array is one large logical table of integers, partitioned into some number of *windows*. GUPS executes a tight loop that, for some number of updates per iteration, computes a random index within a given window for each update and then mutates the value at that index. After each set of updates is applied, a new window is randomly chosen. Figure 8 illustrates the performance comparison between three different approaches, where performance is reported as the rate of million updates applied per second.



(a) Throughput of GET requests

(b) Throughput of SET requests

(c) Varying GET and SET requests

Figure 10: Performance comparison of Redis vs. a version of Redis using SpaceJMP

The first approach (MAP) leverages **address space remapping**: a traditional process may logically extend the reachability of its VAS by dynamically remapping portions of it to different regions of physical memory. Our implementation of this design uses the BSD `mmap` and `munmap` system calls (configured to attach to existing pages in the kernel's page cache) opening new windows for writing.

The second traditional approach (MP) uses **multiple processes**: each process is assigned a distinct portion of the physical memory (a window). In our experiment, one process acts as master and the rest as slaves, whereby the master process sends RPC messages using OpenMPI to the slave process holding the appropriate portion of physical memory. It then blocks, waiting for the slave to apply the batch of updates before continuing, simulating a single thread applying updates to a large global table. Each process is pinned to a core.

We compare these techniques with **VAS switching**, modifying GUPS to take advantage of SpaceJMP. Unlike the first technique, we do not modify mappings when changing windows, but instead represent a window as a segment in its own VAS. Pending updates are maintained in a shared heap segment mapped into all attached address spaces. Figure 9 illustrates the rate of VAS switching and TLB misses.

Discussion. For a single window – no remapping, RPC, or switching – all design configurations perform equally well. Changing windows immediately becomes prohibitively expensive for the MAP design, as it requires modification of the address space on the critical path. For the MP and SpaceJMP designs, the amount of CPU cache used grows with each added window, due to cache line fills from updates, as well as the growth in required page translation structures pulled in by the page-walker. The SpaceJMP implementation performs at least as well as the multi-process implementation, despite frequent context switches, with all data and multiple translation tables competing for the same set of CPU caches (for MP, only one translation table resides on each core). At greater than 36 cores on *M3*, the performance of MP drops, due to the busy-wait characteristics the OpenMPI implementation. The same trends are visible across a range of update set sizes (16 and 64 in the figure). Finally, a design leveraging SpaceJMP is more flexible, as a single process can independently address multiple address spaces without message passing.

This experiment shows that SpaceJMP occupies a useful point in the design space between multiple-process and page-remapping techniques – there is tangible benefit from switching between multiple address spaces rapidly on a single thread.

5.3 Redis with Multiple Address Spaces

In this section, we investigate the trade-off in adapting an existing application to use SpaceJMP and the potential benefits over traditional programming models.

We use Redis (v3.0.2) [67], a popular in-memory key-value store. Clients interact with Redis using UNIX domain or TCP/IP sockets by sending commands, such as SET and GET, to store and retrieve data. Our experiments use local clients and UNIX domain sockets for performance.

We compare a basic single-threaded Redis instance with Redis$_{JMP}$, which exploits SpaceJMP by eliding the use of socket-based communications. Redis$_{JMP}$ avoids a server process entirely, retaining only the server data, and clients access the server data by switching into its address space. Redis$_{JMP}$ is therefore implemented as a client-side library, and the server data is initialized lazily by its first client.

This client creates a new lockable segment for the state, maps it into a newly created address space, switches into this address space, and runs the initialization code to set-up Redis data-structures. We replaced the Redis memory allocator with one based on SpaceJMP, and moved all Redis global variables to a statically-allocated region inside the lockable segment.

Each client creates either one or two address spaces with the lockable segment mapped read-only or read-write, and invokes commands by switching into the newly created address space, executing server code directly. Our Redis$_{JMP}$ implementation currently supports only basic operations for simple data-types. Some Redis features such as publish–subscribe would be more challenging to support, but could be implemented in a dedicated notification service.

Redis$_{JMP}$ uses locked segments to provide parallel read access to Redis state, but two further modifications were needed to the Redis code. First, Redis creates heap objects even on GET (read-only) requests for parsing commands, which would require read-write access to the lockable segment for all commands. We therefore attached a small, per-client scratch heap to each client's server address space. Second, Redis dynamically grows and shrinks its hash-tables asynchronously with respect to queries; we modified it to resize and rehash entries only when a client has an exclusive lock on the address space.

We used the *redis-benchmark* from the Redis distribution to compose the throughput of Redis$_{JMP}$ with the original on a single machine (*M1*). The benchmark simulates multiple clients by opening multiple file descriptors and sending commands while asynchronously polling and waiting for a response before sending the next command. For SpaceJMP we modified *redis-benchmark* to use our API and individual processes as clients that all attach the same server segment.

Fig. 10a and Fig. 10b show the performance of the GET and SET commands (4-byte payload) for a regular Redis server and Redis$_{JMP}$. In case of a single client (one thread), SpaceJMP outperforms a single server instance of Redis by a factor of *4x* for GET and SET requests, by reducing communication overhead.

The maximum read throughput of a single-threaded Redis server is naturally limited by the clock-speed of a single core, whereas Redis$_{JMP}$ allows multiple readers to access the address space concurrently. Therefore, we also compare throughput of a single Redis$_{JMP}$ instance with six independent Redis instances (Redis 6x) paired with six instances of *redis-benchmark* running on the twelve core machine.

Even in this case, at full utilization Redis$_{JMP}$ is still able to serve *36%* more requests than 6 regular Redis instances. We also compare SpaceJMP with and without TLB tagging enabled and notice a slight performance improvement using tags until the synchronization overhead limits scalability. For TLB misses, we measured a rate of 8.9M misses per second with a single client and 3.6M per core per second at full utilization (using 12 clients). With TLB tagging, the miss rate was lower, 2.8M misses per second for a single client and 0.9M per core per second (using 12 clients). The total number of address space switches per second is equals to two times the request rate for any number of clients.

For SET requests we sustain a high request rate until too many clients contend on the segment lock. This is a fundamental SpaceJMP limit, but we anticipate that a more scalable lock design than our current implementation would yield further improvements. Fig. 10c shows maximum system throughput while increasing the percentage of SET requests. The write lock has a large impact on throughput even when 10% of the requests are SETs, but Redis$_{JMP}$ still outperforms traditional file-based communication.

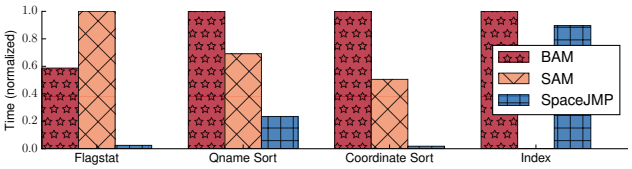


Figure 11: SAMTools vs. an implementation with SpaceJMP. BAM and SAM are alternative in-memory serialization methods; SpaceJMP has no serialization.

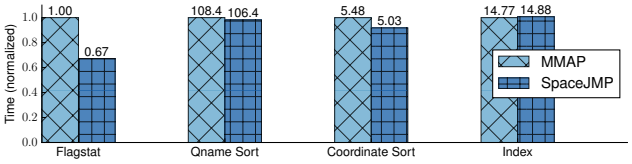


Figure 12: Use of `mmap` vs. SpaceJMP in SAMTools. Absolute runtime in seconds shown above each bar.

In summary, small changes are needed to modify an existing application to use SpaceJMP, but can make a single-threaded implementation both scale better and sustain higher single-thread performance by reducing IPC overhead.

5.4 SAMTools: In-Memory Data Structures

Finally, we show the benefit of using SpaceJMP as a mechanism to keep data structures in memory, avoiding both regular file I/O and memory-mapped files.

SAMTools [50] is a toolset for processing DNA sequence alignment information. It operates on multiple file formats that encode aligned sequences and performs various operations such as sorting, indexing, filtering, and collecting statics and pileup data. Each of these parses file data, performs a computation, and may write output to another file. Much of the CPU time in these operations is spent converting between serialized and in-memory representations of data.

We implement a version of SAMTools that uses SpaceJMP to keep data in its in-memory representation, avoiding frequent data conversions such as serialization of large data structures; such procedures comprise the majority of its execution time. Instead of storing the alignment information in a file according to a schema, we retain the data in a virtual address space and persist it between process executions. Each process operating on the data switches into the address space, performs its operation on the data structure, and keep its results in the address space for the next process to use.

Figure 11 compares the performance of using SpaceJMP to the original SAMTools operations on Sequence Alignment/Map (SAM) and BGZF compressed (BAM) files of sizes 3.1 GiB and 0.9 GiB respectively. It is evident from the graph that keeping data in memory with SpaceJMP results in significant speedup. The SAM and BAM files are stored using an in-memory file-system so the impact of disk access in the

original tool is completely factored out. The performance improvement comes mainly from avoiding data conversion between pointer-rich and serialized formats.

We also implement a version of SAMTools that uses memory-mapped files to keep data structures in memory. Processes access data by calling `mmap` on a file. Region-based programming is used to build the data structures within the file. To maximize fairness, we make `mmap` as lightweight as possible by using an in-memory file system for the files, and pass flags to `mmap` to exclude regions from the core file and inform the pager to not gratuitously sync dirty pages back to disk. Moreover, we stop timers before process exit to exclude the implicit cost of unmapping data.

Figure 12 compares the performance of SpaceJMP to its counter-part using memory-mapped files. It is evident that the SpaceJMP has comparable performance. Therefore, the flexibility provided by SpaceJMP over memory-mapped files (i.e., not using special pointers or managing address conflicts) comes at a free cost. Moreover, with caching of translations enabled, it is expected that SpaceJMP's performance will improve further.

It is noteworthy that flagstat shows more significant improvement from SpaceJMP than the other operations in Figure 12. That is because flagstat runs much quicker than the others so the time spent performing a VAS switch or `mmap` takes up a larger fraction of the total time.

6. Related work

SpaceJMP is related to a wide range of work both in OS design and hardware support for memory management.

System software Memory management is one of the oldest areas of computer systems with many different design considerations that have been proposed so far. The most familiar OSes support address spaces for protection and isolation.

Strong isolation limits flexibility to share complex, pointer rich data structures and shared memory objects duplicate page table structures per process. OpenVMS [61] shares page tables for the subtree of a global section. Munroe patented sharing data among multiple virtual address spaces [59].

Threads of a process usually share a single virtual address space. The OS uses locks to ensure thread safety which leads to contention forcing memory management intensive programs to be split up into multiple processes to avoid lock contention [79]. RadixVM [20] maintains a radix tree from virtual addresses to meta data and allows concurrent operations on non-overlapping memory regions. VAS abstractions as first-class citizens provides a more natural expression of sharing and isolation semantics, while extending the use of scalable virtual memory optimizations.

Cyclone [37] has memory regions with varying life times and checks the validity of pointers in the scopes where they are dereferenced. Systems like QuickStore [80] provide shared-memory implementations of object stores, leveraging the virtual memory system to make objects available via their virtual addresses. Quickstore, however, incurs the cost of in-place pointer swizzling for objects whose virtual address ranges overlap, and the cost itself of mapping memory. SpaceJMP elides swizzling by reattaching alternate VASes, and combines VAS-aware pointer tracking for safe execution.

BSD operating system kernels are designed around the concept of virtual memory objects, enabling more efficient memory mappings, similar to UVM [23].

Prior 32-bit editions of Windows had only 2 GiB of virtual address space available for applications but allowed changing this limit to 3 GiB for memory intensive applications [56]. Furthermore, Address Windowing Extensions [57] gave applications means to allocate physical memory above 4 GiB and quickly place it inside reserved windows in the 32-bit address space, allowing applications to continue using regular 32-bit pointers.

Some single-address-space operating systems (SASOSes) (e.g., Opal [17, 18] and IVY [51]) eliminated duplicate page tables. Nevertheless, protection domains need to be enforced by additional hardware (protection lookaside buffers) or page groups [47]. Systems like Mondrix [83] employ isolation by enforcing protection on fixed entry and exit points of cross-domain calls, much like traditional "call gates" or the CAP computers' "enter capabilities" [82]. Such mechanisms complement SpaceJMP by providing protection capabilities at granularities other than a page within a VAS itself.

Also, SASOSes assume large virtual address spaces, but we predict a limit on virtual address bits in the future. More recent work such as SMARTMAP [12] evaluates data-sharing on high-performance machines. Distributed shared memory systems in the past [46] gained little adoption, but have recently acquired renewed interest [60].

Our work is influenced by the design of Mach [65], particularly the concept of a *memory object* mappable by various processes. Such techniques were adopted in the Opal [17] and Nemesis [38] SASOSes to describe memory *segments* characterized by fixed virtual offsets.

Lindstrom [52] expands these notions to shareable *containers* that contain code segments and private memory, leveraging a capability model to enforce protection. Application threads called *loci* enter containers to execute and manipulate data. *loci* motivated our approach to the client–server model.

In the 90s, research focused on exporting functionality to user space, reducing OS complexity, and enabling applications to set memory management policies [32, 38, 66]. Our work is supported by such ideas, as applications are exposed to an interface to compose virtual address spaces, with protection enforced by the OS, as with seL4 [31] and Barrelfish [7].

The idea of multiple address spaces has mainly been applied to achieve protection in a shared environment [17, 73], and to address the mismatch of abstractions provided by the OS interfaces [2, 66]. Some operating systems also provide similar APIs in order to simplify kernel development by

executing the kernel directly in user-space [26, 30]. Dune [8] uses virtualization hardware to sandbox threads of a process in their own VAS. Further work has used them to transparently manage costs in NUMA systems [11, 25, 43, 54, 62, 68, 77, 85] or HPC applications [13]. SpaceJMP goes further by promoting a virtual address space to a first-class citizen in the OS, allowing applications to compose logically related memory regions for more efficient sharing.

Anderson *et al.* [1] recognized that increased use of RPC-based communication requires both sender and receiver to be scheduled to transfer data involving a context switch and kernel support (LRPC [9]) or shared memory between cores (URPC [10]). In both cases, data marshalling, cache misses, and buffering lead to overhead [16, 45, 75]. Compared to existing IPC mechanisms like Mach Local RPC [28] or overlays [49] SpaceJMP distinguishes itself by crossing VAS boundaries rather than task or process boundaries.

Ideally, bulk-data transfer between processes avoids copies [19, 41, 70, 72], although there are cases where security precludes this [29, 41].

Collaborative data processing with multiple processes often requires pointer-rich data structures to be serialized into a self-contained format and sent to other processes which induces an overhead in web services but also in single multi-core machines [14, 39, 42, 71].

Hardware support Multiple address spaces are also supported by hardware. We have already discussed the trade-offs involved in tagged TLBs, and observed that SpaceJMP would benefit from a more advanced tagging architecture than that found in x86 processors. The HP PA-RISC [53] and Intel Itanium [40, 84] architectures both divide the virtual address space as windows into distinct address regions. The mapping is controlled by region registers holding the VAS identifier.

Our research is further complemented by Direct Segments [6]. SpaceJMP segments are currently backed by the underlying page table structure, but integrating our segments API with Direct Segments would yield further benefits. Alternatively, RMM [44] proposes hardware support that adds a redundant mapping for large ranges of contiguous pages. RMM seems like a good match for SpaceJMP segments, requiring fewer TLB entries than a traditional page-based system. SpaceJMP could be extended straightforwardly to implement RMM's eager allocation strategy. Other work proposes hardware-level optimizations that enable finer granularities than a page [44, 69].

Large pages have been touted as a way to mitigate TLB flushing cost, but such changes require substantial kernel modifications [74] and provide uncertain benefit to large-memory analytics workloads [6], as superpage TLBs can be small. Superpages on NUMA systems may be unfavorable [34].

Bailey *et al.* [5] argue the availability of huge and fast non-volatile, byte addressable memory (NVM) will fundamentally influence OS structure. Persistent storage will be managed by existing virtual memory mechanisms [22, 58] and accessed directly by the CPU. Atlas [15] examines durability when accessing persistent storage by using locks and explicit publishing to persistent memory. The approach in this paper provides a foundation for designing more memory centric operating system architectures.

Compiler-based program analysis for pointer safety introduces runtime checks with associated overhead. These could be made more efficient by labeling pointers with address space information. Some architectures such as ARM's *aarch64* [4] support the notion of "tagged pointers" where some bits (8, in aarch64) in every pointer can be made available to software and not passed to the MMU.

Lockable segments may benefit from hardware transactional memory (HTM) support: encapsulating the address space switch in an HTM-supported transactions can avoid the need of locking (or other synchronization) and improve performance of shared regions.

7. Conclusion

SpaceJMP extends OS memory management with lockable segments and multiple address spaces per process as a way to more efficiently execute data-centric applications which use massive memory and/or pointer-rich data structures.

However, we also claim that SpaceJMP has broader applicability. For example, SpaceJMP can also help in handling the growing complexity of memory. We expect future memory systems will include a combination of several heterogeneous hardware modules with quite different characteristics: a co-packaged volatile performance tier, a persistent capacity tier, different levels of memory-side caching, private memory (for example, only accessible by accelerators), and so on. Applications today use cumbersome techniques such as explicit copying and DMA to operate in these environments. SpaceJMP can be the basis for tying together a complex heterogeneous memory system and provide the application support for efficient programming. Another potential application is sandboxing, using different address spaces to limit access only to trusted code.

Our ongoing work is exploring these possibilities, but we also plan to address other issues such as the persistency of multiple virtual address spaces (for example, across reboots), and apply optimizations on address space creation, such as copy-on-write, snapshotting, and versioning.

So far, our experience has been encouraging, and we hope that SpaceJMP becomes a solid foundation for using memory in next-generation hardware platforms with high demands for data and with very large memories.

8. Acknowledgements

We thank the anonymous reviewers and our shepherd, Galen Hunt, for their many helpful suggestions. We would also like to thank our University colleagues and many Hewlett Packard Labs staff for their valuable insights and feedback.

References

[1] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 108–120, Santa Clara, California, USA, 1991.

[2] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 96–107, Santa Clara, California, USA, 1991.

[3] ARM Ltd. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*, 2014. ARM DDI 0406C.c.

[4] ARM Ltd. ARMv8-A Architecture. Online, 2015. `http://www.arm.com/products/processors/armv8-architecture.php`.

[5] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating System Implications of Fast, Cheap, Non-volatile Memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 2–2, Napa, California, 2011.

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, Tel-Aviv, Israel, 2013.

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS Twenty-Second Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, Big Sky, Montana, USA, 2009.

[8] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Hollywood, CA, USA, 2012.

[9] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM Trans. Comput. Syst.*, 8(1):37–55, February 1990.

[10] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level Interprocess Communication for Shared Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(2):175–198, May 1991.

[11] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 212–221, Santa Clara, California, USA, 1991.

[12] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-core Processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 25:1–25:12, Austin, Texas, 2008.

[13] Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 359–368, Eugene, Oregon, USA, 2013.

[14] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. In *Proceedings of the ACM 1999 Conference on Java Grande*, JAVA '99, pages 66–71, San Francisco, California, USA, 1999.

[15] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, Portland, Oregon, USA, 2014.

[16] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-passing and Shared-memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 61–73, San Jose, California, USA, 1994.

[17] Jeff Chase, Miche Baker-Harvey, Hank Levy, and Ed Lazowska. Opal: A Single Address Space System for 64-bit Architectures. *SIGOPS Oper. Syst. Rev.*, 26(2):9–, April 1992.

[18] Jeffrey S. Chase, Henry M. Levy, Miche Baker-harvey, and Edward D. Lazowska. How to Use a 64-Bit Virtual Address Space. Technical report, Department of Computer Science and Engineering, University of Washington, 1992.

[19] Ting-Hsuan Chien, Chia-Jung Chen, and Rong-Guey Chang. An Adaptive Zero-Copy Strategy for Ubiquitous High Performance Computing. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 139:139–139:144, Kyoto, Japan, 2014.

[20] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, Prague, Czech Republic, 2013.

[21] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, Newport Beach, California, USA, 2011.

[22] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS Twenty-Second Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, Big Sky, Montana, USA, 2009.

[23] Charles D. Cranor and Gurudatta M. Parulkar. The UVM Virtual Memory System. In *Proceedings of the 1999 USENIX Annual Conference*, ATEC '99, pages 9–9, Monterey, California, 1999.

[24] Danga Interactive. memcached - a distributed memory object caching system. Online, April 2015. `http://memcached.org/`.

[25] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, Houston, Texas, USA, 2013.

[26] Jeff Dike. *User Mode Linux*. Upper Saddle River, NJ, USA, 2006.

[27] Matt Dillon. DragonFly BSD Sources. `http://www.dragonflybsd.org/`, June 2015.

[28] Richard Draves. A Revised IPC Interface. In *USENIX MACH Symposium*, pages 101–122. USENIX, 1990.

[29] Peter Druschel and Larry L. Peterson. Fbufs: A High-bandwidth Cross-domain Transfer Facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 189–202, Asheville, North Carolina, USA, 1993.

[30] Aggelos Economopoulos. A peek at the DragonFly Virtual Kernel (part 1). `https://lwn.net/Articles/228404/`, March 2007.

[31] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel Design for Isolation and Assurance of Physical Memory. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, pages 35–40, Glasgow, Scotland, 2008.

[32] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, Copper Mountain, Colorado, USA, 1995.

[33] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems*, HotOS XV, Kartause Ittingen, Switzerland, May 2015.

[34] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference*, USENIX ATC'14, pages 231–242, Philadelphia, PA, 2014.

[35] Simon Gerber, Gerd Zellweger, Reto Achermann, Kornilios Kourtis, Timothy Roscoe, and Dejan Milojicic. Not Your Parents' Physical Address Space. In *15th Workshop on Hot Topics in Operating Systems*, HotOS XV, Kartause Ittingen, Switzerland, May 2015.

[36] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52, Salt Lake City, UT, USA, 2008.

[37] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, Berlin, Germany, 2002.

[38] Steven M. Hand. Self-paging in the Nemesis Operating System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 73–86, New Orleans, Louisiana, USA, 1999.

[39] Marjan Hericko, Matjaz B. Juric, Ivan Rozman, Simon Beloglavec, and Ales Zivkovic. Object Serialization Analysis and Comparison in Java and .NET. *SIGPLAN Not.*, 38(8):44–54, August 2003.

[40] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual*. Document Number: 245315.

[41] Bär Jeremia and Föllmi Claudio. Bulk Transfer over Shared Memory. Technical report, ETH Zurich, Feburary 2014. `http://www.barrelfish.org/dsl-bulk-shm-report.pdf`.

[42] Matjaz B. Juric, Bostjan Kezmah, Marjan Hericko, Ivan Rozman, and Ivan Vezocnik. Java RMI, RMI Tunneling and Web Services Comparison and Performance Analysis. *SIGPLAN Not.*, 39(5):58–65, May 2004.

[43] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: Smart Aladdress and Replication of Memory For Parallel Programs. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC 15, pages 263–276, Santa Clara, CA, USA, July 2015.

[44] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 66–78, Portland, Oregon, 2015.

[45] Vijay Karamcheti and Andrew A. Chien. Software Overhead in Messaging Layers: Where Does the Time Go? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 51–60, San Jose, California, USA, 1994.

[46] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, San Francisco, California, USA, 1994.

[47] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architecture Support for Single Address Space Operating Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 175–186, Boston, Massachusetts, USA, 1992.

[48] Doug Lea. dlmalloc: A Memory Allocator. `http://g.oswego.edu/dl/html/malloc.html`, April 2000.

[49] J.R. Levine. *Linkers and Loaders*. Morgan Kauffman, 2000.

[50] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[51] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. *ICPP (2)*, 88:94, 1988.

[52] A. Lindstrom, J. Rosenberg, and A. Dearle. The Grand Unified Theory of Address Spaces. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, HotOS-V, pages 66–71, May 1995.

[53] Michael J. Mahon, Ruby Bei-Loh Lee, Terrence C. Miller, Jerome C. Huck, and William R. Bryg. The Hewlett-Packard Precision Architecture: The Processor. *Hewlett-Packard Journal*, 37(8):16–22, August 1986.

[54] Zoltan Majo and Thomas R. Gross. Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 11–20, San Jose, California, USA, 2011.

[55] Collin McCurdy, Alan L. Coxa, and Jeffrey Vetter. Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '08, pages 95–104, 2008.

[56] Microsoft Corp. 4-Gigabyte Tuning: BCDEdit and Boot.ini. `https://msdn.microsoft.com/en-us/library/windows/desktop/bb613473(v=vs.85).aspx`.

[57] Microsoft Corp. Address Windowing Extensions. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366527(v=vs.85).aspx`.

[58] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 1:1–1:17, Farmington, Pennsylvania, 2013.

[59] Steven Jay Munroe, Scott Alan Plaetzer, and James William Stopyro. Computer System Having Shared Address Space Among Multiple Virtual Address Spaces. Online, January 20 2004. US Patent 6,681,239, `https://www.google.com/patents/US6681239`.

[60] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC 15, pages 291–305, Santa Clara, CA, July 2015.

[61] Karen L. Noel and Nitin Y. Karkhanis. OpenVMS Alpha 64-bit Very Large Memory Design. *Digital Tech. J.*, 9(4):33–48, April 1998.

[62] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 3–18, Salt Lake City, Utah, USA, 2014.

[63] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

[64] S.J. Plimpton, R. Brightwell, C. Vaughan, K. Underwood, and M. Davis. A Simple Synchronous Distributed-Memory Algorithm for the HPCC RandomAccess Benchmark. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, pages 1–7, Sept 2006.

[65] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 31–39, Palo Alto, California, USA, 1987.

[66] Barret Rhoden, Kevin Klues, David Zhu, and Eric Brewer. Improving Per-node Efficiency in the Datacenter with New OS Abstractions. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 25:1–25:8, Cascais, Portugal, 2011.

[67] Salvatore Sanfilippo. Redis. `http://redis.io/`, August 2015.

[68] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable Locality-conscious Multithreaded Memory Allocation. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 84–94, Ottawa, Ontario, Canada, 2006.

[69] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 79–91, Portland, Oregon, 2015.

[70] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 57–57, Denver, Colorado, 2001.

[71] Audie Sumaray and S. Kami Makki. A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, pages 48:1–48:6, Kuala Lumpur, Malaysia, 2012.

[72] Brian Paul Swenson and George F. Riley. A New Approach to Zero-Copy Message Passing with Reversible Memory Aladdress in Multi-core Architectures. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, PADS '12, pages 44–52, Washington, DC, USA, 2012.

[73] M. Takahashi, K. Kono, and T. Masuda. Efficient Kernel Support of Fine-grained Protection Domains for Mobile Code. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, ICDCS '99, pages 64–73, Washington, DC, USA, 1999.

[74] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 171–182, San Jose, California, USA, 1994.

[75] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 2–11, San Jose, California, USA, 1994.

[76] Dana Vantrease, Robert Schreiber, Matteo Monchiero, Moray McLaren, Norman P Jouppi, Marco Fiorentino, Al Davis, Nathan Binkert, Raymond G Beausoleil, and Jung Ho Ahn. Corona: System implications of emerging nanophotonic technology. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 153–164. IEEE Computer Society, 2008.

[77] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 279–289, Cambridge, Massachusetts, USA, 1996.

[78] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, Newport Beach, California, USA, 2011.

[79] David Wentzlaff and Anant Agarwal. Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, April 2009.

[80] Seth J. White and David J. DeWitt. Quickstore: A high performance mapped object store. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 395–406, New York, NY, USA, 1994. ACM.

[81] John Wilkes and Bart Sears. A comparison of Protection Lookaside Buffers and the PA-RISC Protection Architecture. Technical Report HPL–92–55, Computer Systems Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA, USA, March 1992.

[82] M. V Wilkes. *The Cambridge CAP Computer and Its Operating System (Operating and Programming Systems Series)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1979.

[83] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 31–44, Brighton, United Kingdom, 2005.

[84] Rumi Zahir, Jonathan Ross, Dale Morris, and Drew Hess. OS and Compiler Considerations in the Design of the IA-64 Architecture. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 212–221, Cambridge, Massachusetts, USA, 2000.

[85] Jin Zhou and Brian Demsky. Memory Management for Many-core Processors with Software Configurable Locality Policies. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 3–14, Beijing, China, 2012.