SPX64: A Scratchpad Memory for General-purpose Microprocessors

ABHISHEK SINGH, Lehigh University
SHAIL DAVE, Arizona State University
PANTEA ZARDOSHTI, Lehigh University
ROBERT BROTZMAN, Pennsylvania State University
CHAO ZHANG and XIAOCHEN GUO, Lehigh University
AVIRAL SHRIVASTAVA, Arizona State University
GANG TAN, Pennsylvania State University
MICHAEL SPEAR, Lehigh University

General-purpose computing systems employ memory hierarchies to provide the appearance of a single large, fast, coherent memory. In special-purpose CPUs, programmers manually manage distinct, non-coherent scratchpad memories. In this article, we combine these mechanisms by adding a virtually addressed, set-associative scratchpad to a general purpose CPU. Our scratchpad exists alongside a traditional cache and is able to avoid many of the programming challenges associated with traditional scratchpads without sacrificing generality (e.g., virtualization). Furthermore, our design delivers increased security and improves performance, especially for workloads with high locality or that interact with nonvolatile memory.

CCS Concepts: • **Hardware → Memory and dense storage**; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Security and privacy → Hardware security implementation**; **Hardware attacks and countermeasures**;

Additional Key Words and Phrases: Scratchpad memory, cache, persistent memory, security, software managed memory

New article, not an extension of a conference paper.
This work was partially supported by the Intel and NSF joint research program for Computer Assisted Programming for Heterogeneous Architectures (CAPA). At Lehigh University, this work was supported under grant CCF-1723624 and CCF-1750826. At the Arizona State University, this work was supported under grant and CCF-1723476 and CNS-1525855. At the Pennsylvania State University, this work was supported under grant CCF-1723571. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Intel or the National Science Foundation.
Authors' addresses: A. Singh, C. Zhang, and X. Guo, Department of Electrical and Computer Engineering, 19 Memorial Drive West, Bethlehem, PA 18015; emails: {abs218, chz616, xig515}@lehigh.edu; S. Dave and A. Shrivastava, 699 S. Mill Ave, Suite 203-15 Centerpoint, Tempe, AZ 85281; emails: {Aviral.Shrivastava, Shail.Dave}@asu.edu; P. Zardoshti and M. Spear, Computer Science and Engineering Department, 113 Research Drive, Bethlehem PA 18015; emails: {mfs409, paz215}@lehigh.edu; R. Brotzman and G. Tan, W358 Westgate Building, Dept. of Computer Science & Engineering, The Penn State University, University Park, PA 16802, USA; emails: rcb44@psu.edu, gtan@cse.psu.edu.

1544-3566/2020/12-ART14
https://doi.org/10.1145/3436730

ACM Reference format:
Abhishek Singh, Shail Dave, Pantea Zardoshti, Robert Brotzman, Chao Zhang, Xiaochen Guo, Aviral Shrivastava, Gang Tan, and Michael Spear. 2020. SPX64: A Scratchpad Memory for General-purpose Microprocessors. *ACM Trans. Archit. Code Optim.* 18, 1, Article 14 (December 2020), 26 pages.
https://doi.org/10.1145/3436730

1 INTRODUCTION

General-purpose microprocessors employ coherent caches to ensure that the processor has the data it needs to continue executing a program. Some special-purpose [18, 24] and embedded [5, 16] microprocessors do not have caches. Instead, they employ scratchpad memories: small, physically addressed, direct-mapped storage devices that offer low latency and high predictability, at the cost of generality. These special-purpose systems must use explicit data transfer instructions (e.g., DMA) to fetch data from DRAM to the scratchpad or to write data from the scratchpad to DRAM. In some cases, the compiler can generate DMA instructions automatically [38, 40]. In others, programmers must explicitly manage data movement [10, 13, 24, 27].

Explicitly managing data movement and remapping to the scratchpad's flat address space is cumbersome, but brings some advantages. First, scratchpads require fewer transistors than caches: They do not implement coherence, and they have less metadata (e.g., tag, valid, and dirty bits). The lower transistor counts and simpler design lead to lower energy consumption and lower access latency than caches. Second, scratchpad performance is more predictable than cache performance: When data is in the scratchpad, the programmer can be certain that it will not be evicted, and thus the average memory access time should have low variance. Third, scratchpad geometry can be independent of system-wide memory hierarchy considerations. This is in stark contrast to caches, which must handle aliasing and must obey constraints imposed by page size and address translation.

While these benefits are appealing, using scratchpads in general-purpose processors bring new challenges. Chief among them is resource management: Whereas a special-purpose chip may never context switch among programs that require the scratchpad, a general-purpose design must deal with context switching as well as the possibility that a scratchpad resource needs to be shared (i.e., partitioned) among many hardware threads. Effective resource management is likely to require both hardware and software support that is not present in traditional scratchpad-based systems.

In this article, we explore the question of whether it would be beneficial to add a scratchpad memory beside a traditional cache. The simplest motivation for doing so is that the scratchpad could provide the CPU with an additional close, fast, low-power memory without changing the L1 design. We also show that augmenting a CPU with a scratchpad provides the ability for programs to shut down certain side-channels by placing their most sensitive data in the scratchpad. Furthermore, if we expand the scratchpad design to support more general-purpose features such as associativity and virtual addressing, it also becomes possible to use the scratchpad as a shadow address space for executing code speculatively, as is the case in transactional programming models for nonvolatile memory.

Our design, SPX64, blurs the distinction between caches and scratchpads. It introduces a **"scratchpad data cache" (SD$)**, which is not a part of the regular memory hierarchy, and is virtually addressed. The SD$ is set associative, but raises an exception instead of evicting data in response to an associativity overflow. Furthermore, it is populated via register transfers, not DMA. Thus, it is both separate from the regular memory hierarchy, but also interacts with the regular memory hierarchy through explicit data movement between from and to it via uncachable memory accesses.

Table 1. SPX64 Instructions

Instruction	Behavior	Instruction	Behavior
`SpxRead %r1 [%r2]`	Read from a scratchpad block	`SpxClear`	Drop all blocks from the scratchpad
`SpxWrite [%r1] %r2`	Write to a scratchpad block	`SpxManage %r1 %r2`	Allocate scratchpad-way (supervisor mode)
`SpxZero [%r1]`	Zero a scratchpad block	`SpxAlloc %r1`	Request scratchpad-ways (user mode)
`SpxInv [%r1]`	Drop a block from scratchpad	`SpxRelease`	Release scratchpad-ways (user mode)

The **operating system (OS)** manages a core's SD$ at the granularity of its *ways* by securely partitioning them among a core's hardware threads. The OS reserves the right to revoke a software thread's scratchpad allocation at any time, which leads to a best-effort programming model. Through a trivial OS extension, programs can give "hints" about their scratchpad use. These hints allow the OS to virtualize the scratchpad for two of the three use cases we consider, enabling context switching and thread migration.

In summary, this article proposes a practical method to achieve the performance, predictability, security, and transactional benefits of a scratchpad memory in general-purpose computing setting without adding much programming complexity or hardware overhead. In specific, the salient points of the proposed SPX64 design are:

- SPX64 integrates SPM in the regular cache hierarchy of a general-purpose processor to provide power-efficient and predictable (latency) computing without excessive increase in programming complexity.
- SPX64 provides a way to isolate security-critical data of an application to protect it against many cache-based side-channel attacks.
- SPX64 can serve as a private shadow address space, which can be used to reduce the cost of redo-logging, thereby improving the performance of transactional workloads.

We demonstrate through simulation that SPX64 can deliver significant performance improvements: up to 17% for security workloads, up to 10% for general compute workloads, and up to 20% for persistent workloads.

The remainder of this article is organized as follows: In Section 2, we describe SPX64 from a programmer's perspective and present its software API. We give examples of how an SD$ can benefit programs in Section 3. Section 4 discusses the implementation of SPX64 and its interaction with the rest of the CPU. Sections 5 and 6 describe the operating system support and programming model for SPX64, respectively. Section 7 describes experimental methodology. In Section 8, we evaluate the programmability and performance of SPX64 for three use cases: shutting down cache-based side channels, reducing the overhead of redo logging for persistent transactions over nonvolatile memory, and accelerating workloads with high locality. We contrast SPX64 with related work in Section 9, and then discuss conclusions and future work in Section 10.

2 SPX64: A PROGRAMMER'S VIEW

The easiest way to think of SPX64 is as a fixed-size hash table parameterized by a power-of-two block size (B). The keys in this hash table are virtual addresses, where the lowest $log_2(B)$ bits are zero, and the values are B-byte arrays. A thread reserves an SD$ through a system call (Section 5). Below, we discuss the instructions that interact with the SD$. They are summarized in Table 1.

A thread creates a key/value mapping in the SD$ with the `SpxWrite` command. The data to be written is provided via the `%r2` operand, and the destination within SD$ is determined by `%r1`. The size of `%r2` is typically 1, 2, 4, 8, or 16 bytes, but can be larger when using SIMD registers. It is most natural for the block size to be larger than the register size (e.g., 64 bytes, to match the L1 cache). When the register size is larger than the block size, multiple memory access instructions are generated. If the store is a hit (i.e., there is an existing mapping for `%r1`), then the appropriate bytes of the corresponding mapping in the SD$ are updated and the operation completes. When there is no mapping for `%r1`, SPX64 uses a *copy-on-allocate* strategy: It issues non-cached load requests to the lower level of the memory to create a mapping in the SD$ from the address to the data and then updates the data. If there is insufficient space in the SD$ to create the mapping, then an exception is raised. `SpxZero` is an optimized version of `SpxWrite`: On a hit, it zeroes the block, and on a miss, it skips fetching data from the memory hierarchy, instead zeroing the entire block. As with `SpxWrite`, `SpxZero` can raise an exception if the SD$ cannot allocate a block.

`SpxRead` reads from the SD$ and places the result in `%r1`. On a *hit*, the address is already present as a key, and the appropriate bytes are read from the corresponding block, with the size of `%r1` determining the number of bytes read. If there is no mapping for the requested address, then the same copy-on-allocate technique is used as in `SpxWrite`: Either the block will be requested from the memory hierarchy via a non-cached load or else an exception will raise.

The programmer is able to selectively drop blocks from the SD$ via `SpxInv`. If the block is dirty, its modifications will be discarded. If an invalidated block is subsequently re-accessed via `SpxRead` or `SpxWrite`, the contents will be refreshed with the latest values from the memory hierarchy. Finally, the programmer can drop all blocks from the SD$ with the `SpxClear` instruction. This returns the SD$ to the state it was in immediately after the system call that allocated the SD$ to the thread.

We briefly highlight differences between an SPX64 scratchpad and a cache. First, if capacity or associativity constraints prohibit the allocation within the SD$, an exception is raised, which must be handled by the programmer or a runtime library; there are no silent evictions of scratchpad blocks. Second, when any core issues a regular store to its cache, no SD$ is notified: Remote memory accesses do not cause evictions from the scratchpad, nor do local stores cause updates. Third, when any core issues a regular load to its cache, no SD$ is notified: Regular memory accesses do not receive the latest scratchpad updates, even from a local SD$. The scratchpad is not coherent, even with the local cache of the same core. Fourth, the programmer must explicitly write-back data to the memory hierarchy by `SpxReading` it into a register and then performing a regular store to the memory hierarchy. Finally, the programmer can manage the contents of the SD$ by way of the `SpxClear` and `SpxInv` commands.

3 NOVEL USES OF SPX64

The availability of an SD$ immediately provides the programmer with the benefits of a traditional scratchpad: predictable performance at lower energy than a cache. The argument for this claim is simple: If a compute kernel can use DMA to fetch a region of memory, then compute over that region, and finally use DMA to write back some subset of that region to main memory, then it can do the same with the SD$. The initial DMA is replaced with a loop that repeatedly calls `SpxRead`, the final DMA is replaced with a loop that repeatedly calls `SpxWrite`, and in between, loads and stores become `SpxRead` and `SpxWrite` instructions. While it may be possible to avoid some latency by skipping the initial `SpxRead` loop, doing so might decrease predictability. Additional benefits can be gained by using the SD$ to avoid unnecessary writebacks of temporary data [17]. We now turn our attention to novel use cases that SPX64 provides for general-purpose CPUs.

3.1 Preventing Cache-based Side Channels

Side channels are unintended information channels that allow an attacker to steal sensitive data by exploiting observable execution behaviors (e.g., cache states, timing, power consumption). Side-channel attacks have been demonstrated to be feasible against a variety of cryptographic ciphers, such as RSA [26, 36, 44], AES [19, 35, 39], and ElGamal [30, 48]. These attacks only require the attacker to share hardware resources with the confidential computation (e.g., in a cloud-computing environment) and can reveal private information such as a cryptographic key.

CPU cache side-channel attacks take two general forms [7, 19, 22, 30, 35–37, 39, 43, 44, 48]. In access-based side-channel attacks, an attacker probes the cache to determine what memory locations the victim has recently used. These attacks can occur during execution or after program termination [14] by observing the memory access patterns of the victim program. In timing-based attacks, an attacker observes the overall execution time of the victim program and correlates faster times with cached data and slower times with uncached data. More recent attacks, such as Spectre [25], leverage speculative execution in conjunction with the aforementioned cache side-channel techniques to obtain an even broader attack surface.

SPX64 provides programmers with a new mechanism for hiding their program's memory access pattern. Despite its proximity to a CPU core and its reliance on the memory hierarchy to fetch data, memory accesses that hit in the SD$ are not visible to other hardware threads. This allows the programmer to mitigate cache-based side-channel attacks by isolating the user's cache from potential adversaries, preventing the adversary from observing the user's cache access patterns. Note that while SPX64 removes a shared medium that an adversary could otherwise access, isolation alone does not prevent timing attacks. SPX64 can also be used to remove timing variations due the CPU cache, but may not prevent all possible timing attacks, since timing variations can result from many sources (e.g., variable time instructions such as division, branches). These sources of variation are outside of the scope of this work.

```
void AES() {   // Round 1
  cipher[0] = rol(SBox[(plain[0]^rk[0])>>0],0);
  cipher[3] = rol(SBox[(plain[1]^rk[1])>>8],8);
  cipher[2] = rol(SBox[(plain[2]^rk[2])>>16],16);
  cipher[1] = rol(SBox[(plain[3]^rk[3])>>24],24);...}
```

Listing 1. The use of the round key (secret key) by AES.

Listing 1 shows part of the first round of an AES encryption, in which each of the lines of the plaintext and round key is XORed together and then used as an index to the **substitution box (SBox)**. This code is vulnerable to cache side-channel attacks: Different indices in the SBox will map to different and predictable cache lines; to steal a victim's secret AES key, an adversary observes which cache lines are accessed by the victim [7, 35]. The mapping from cache lines to indices is typically known in these scenarios. The last step for the adversary is to compute the key, which is trivial if the plaintext is known. Even if the plaintext is unknown, the adversary can correlate observed cache lines with the key by collecting many samples. With SD$, we can shut down this side channel. We place the victim's SBox in the SD$. While the initial bulk copy of the SBox to SD$ is observable, it reveals no information about the key. All subsequent accesses to the SBox by the victim are invisible to the attacker, who cannot see SD$ accesses, since they are private to the core.

Cache-based side-channel attacks have also been shown to be effective on modern asymmetric cryptographic algorithms that use modular exponentiation, such as RSA and ElGamal. Attacks such as CacheBleed [45] target precomputed exponentiation tables in these routines that are indexed using key dependent data. Again, by placing these precomputed tables into the SD$, a hardware thread can ensure that its access pattern cannot be inferred by other hardware threads.

An SD$ can also aid in preventing the variant of Spectre-style attacks [25] that leverage the CPU cache to exfiltrate data (this discussion does not apply to non-cache leakage mediums). Once a user identifies the sensitive data, that data can be loaded into the SD$ without involving shared caches, after which accesses to it cannot be observed by an attacker as long as the sensitive data is only accessed using SD$ instructions, thus preventing inter-procedural attacks. This suffices to prevent Spectre attacks on direct branches, since the potential targets are known. For speculative attacks leveraging indirect branches, SPX64 can be used with existing software mitigations, such as retpoline [1]. Additionally, Section 8.1.1 shows a variant of SPX64 (spx-2-nonSpec) that does not access SD$ speculatively, and thus denies a cache-based covert channel for retrieving sensitive data even in an intra-procedural setting. This also applies to hardware multithreading, since SPX64 allocates resources on a per-thread basis.

While the example given in Listing 1 only requires a few kilobytes of data to be stored in the scratchpad, some applications may require more memory than the SD$ has available at a given time. This may seem like a cause for concern; however, our design prohibits other threads from accessing the data in the SD$, ensuring cache-based side channels are mitigated even if data needs to be replaced in the SD$. The isolation enforced by the SD$ ensures an adversary cannot infer the state of the user's cache, thus mitigating the threat. We note that if the user application replaces data from the SD$ in a manner that depends on sensitive data, they may introduce a timing-channel, which is outside of the scope of this design.

3.2 Accelerating Log Lookups in Persistent Transactions

Non-volatile memory (NVM) is byte-addressable (like DRAM) and persistent (like storage). The easiest way to interact with it is to treat it as a RAM disk and interact with it via a file-system interface [3, 9, 15]. Better performance is possible by directly mapping the NVM into a process's address space. However, for as long as caches are not persistent, such a strategy requires the explicit addition of special "flush" instructions to ensure that data moves from cache to NVM at the right time and "fence" instructions to ensure persistent ordering for crash recovery.

Correct placement of fences and flushes is easiest when using language-level storage transactions [41]. There are two main strategies for implementing storage transactions. In undo logging [8, 31], every modification of memory is preceded by a store of the original value to a persistent log. The log is used to restore memory if the system crashes during a transaction and discarded when a transaction commits. In redo logging [41], all updates are computed and stored in a separate log, which is persisted and written back at transaction commit time. Undo logging requires up to $O(W)$ memory fences, where W is the number of stores performed by a transaction. Redo logging avoids this cost, but every read by a transaction must check the redo log to see if the enclosing transaction has a pending write. This lookup overhead is expensive [12, 46].

SPX64 can serve as a sparse, private, shadow address space, which reduces the cost of redo logging. Listing 2 shows how traditional transactions interact with a redo log, and Listing 3 shows a version that leverages SPX64. To illustrate the difference, consider a transaction that increments the values at locations $\{l_0, l_1, l_2, \ldots, l_n\}$, where for $i \neq j$, it is possible that $l_i = l_j$. For each location, the transaction executes `increment_byte(&l_i)`. Without SPX64, the transaction must maintain a hash table in which the keys are addresses (with the k low bits zeroed, for some k) and values are blocks of 2^k bytes. To increment the value at some location l, the transaction would employ the

```
void increment_byte(char* addr) {
  align_addr = mask_lowbits(addr);
  if (!redo_hash.contains(align_addr)) {
    b = read_cache_line(align_addr);
    redo_hash.put(align_addr, b);}
  b = redo_hash.get(align_addr);
  b[addr - align_addr]++;}
```

Listing 2. Interaction with a redo log (no SPX64).

technique in Listing 2. There are two reasons to use "read_cache_line" in Listing 2: (1) if there is good spatial locality, then caching the entire cache line once will be more efficient for subsequent reads and for commit-time writeback; (2) if the program accesses an address with casts to different sized primitive types (e.g., write a char now, read the enclosing int that covers that byte later), then this approach is significantly faster than storing masks in the hash table and reconstructing values with partial results from the hash table and partial results from memory. Note that overheads of the operation include the hash function itself, traversing links in the hash table implementation, and computing offsets.

With a virtually tagged, set-associative SD$, as long as the working set of the transaction does not overflow the SD$, a much simpler approach is possible, as depicted in Listing 3. The hash table is replaced with a simple vector (the writeback log). To update a value, rather than search the hash, the thread uses SpxRead to search the scratchpad. On a miss in the scratchpad, the scratchpad *automatically* fetches the needed data from the L1 cache. The thread then computes the new value and writes it to the scratchpad and log.

```
void increment_byte(char* addr) {
  tmp = SpxRead(addr) + 1;
  SpxWrite(addr, tmp);
  writeback_log.push_back({addr, tmp});}
```

Listing 3. Interaction with a redo log (with SPX64).

Whereas the original code included an explicit hash table, which could be iterated through during the writeback phase, our new code requires an explicit writeback log. Reads never perform lookups in this log, so it does not require hashing or a complex linked structure; a simple vector suffices. Additionally, with SPX64, each writeback is of exactly the granularity of the corresponding update, whereas in the hash table code, each writeback is a full block of data. Since there are no lookups, the SPX64 approach may lead to multiple entries for the same l in the vector. The scratchpad approach is expected to be favorable when high frequencies of writes to a single location from a single transaction are rare, or when writes to only a small number of bytes per cache line are common.

This simplified write logging is possible, because SPX64 is virtually indexed but not coherent. The write will not be evicted back into the memory hierarchy due to capacity or associativity: software management of the scratchpad will be invoked instead. In addition, the scratchpad does not observe or reply to remote requests for data that it stores. Consequently, the scratchpad can be used as a small hash table that shadows main memory without leaking a transaction's intermediate state.

4 MICROARCHITECTURE DESIGN

To a programmer or a compiler, SPX64 appears as a non-coherent, associative cache, whose ways can be allocated to threads via a system call and returned to the OS through another system call. Through a handful of new assembly instructions, the programmer can manage the SD$ by writing data to it, reading data from it, and invalidating lines within it. We now describe the SPX64 architecture, including its integration with the memory hierarchy and modifications to other microarchitectural structures such as the **load-store queue (LSQ)** unit, the **miss status holding registers (MSHRs)**, and the decode unit.

4.1 An Overview

Figure 1 provides an overview of the core's microarchitecture with SPX64. Light-shaded hardware components indicate the presence of SPX64 instructions or SD$ blocks, while dark-shaded regions

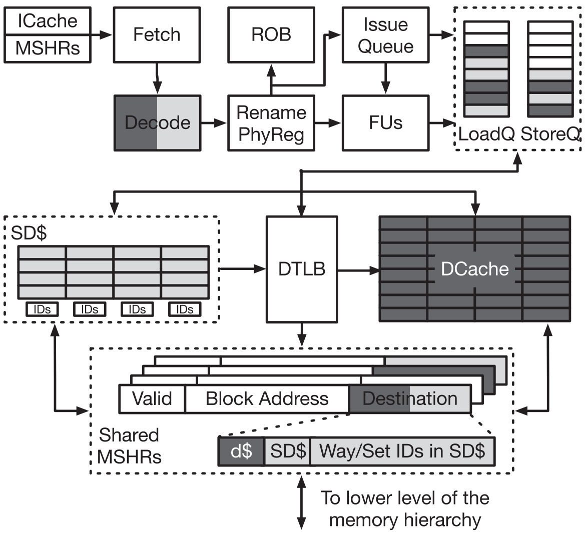

Fig. 1. Microarchitecture overview for SPX64. SD$ is a virtually addressed, software-managed cache. Enhancements in the LSQ, MSHRs, and the decode unit facilitate its functionality.

represent regular memory instructions or L1-DCache blocks. The logical location of the SD$ within a core is parallel to the L1-DCache, so the execution engine can issue loads and stores to the L1-DCache and SD$. The SD$ is a virtually addressed, set-associative, software-managed data cache. Set associativity enables secure partitioning of the SD$ (in ways) among the hardware threads of a core and provides associative search capabilities to software. Finally, software management of the SD$ allows threads to securely manage private data such that accesses to the allocated space of one hardware thread do not influence cache states of other hardware threads or the memory hierarchy. When SPX64 accesses miss, the SD$, data blocks are fetched from the memory hierarchy *without allocating a new line* or changing the LRU stacks of the existing cache lines in the L1-DCache/LLC. It also does not change the existing coherence state of the cache line on hit/miss in the L1-DCache/LLC. Hence, the proposed SD$ can be used as a separate shadow address space and provides protection of private data, which is not brought into the L1-DCache or L2-Cache if it is already not there.

4.2 SPX64 Architecture and Peripheral Logic

Figure 2 depicts the design of the SD$. The SD$ is organized into set-associative ways; each way stores the same number of blocks, similar to ways in an L1 data cache. While it is possible for an implementation to provide any number of ways, and a variety of block sizes, the SD$ is easiest to reason about when its geometry matches the L1 cache. To that end, our discussion focuses on an 8-way set-associative design with 2^6 blocks, where each block is 2^6 bytes. When paired with a standard x86_64 ISA, SD$ has the following characteristics:

(1) *Index for SD$ data blocks:* For a 48-bit virtual address, the page offset is 12 bits (6 each for the index and block offset). A 6-bit index from the page offset locates the appropriate set.
(2) *Metadata per block:* Each block requires a valid bit. For a 48-bit virtual address space, 64-byte blocks, and 64 blocks per way, 36 tag bits from the virtual address are required per block.

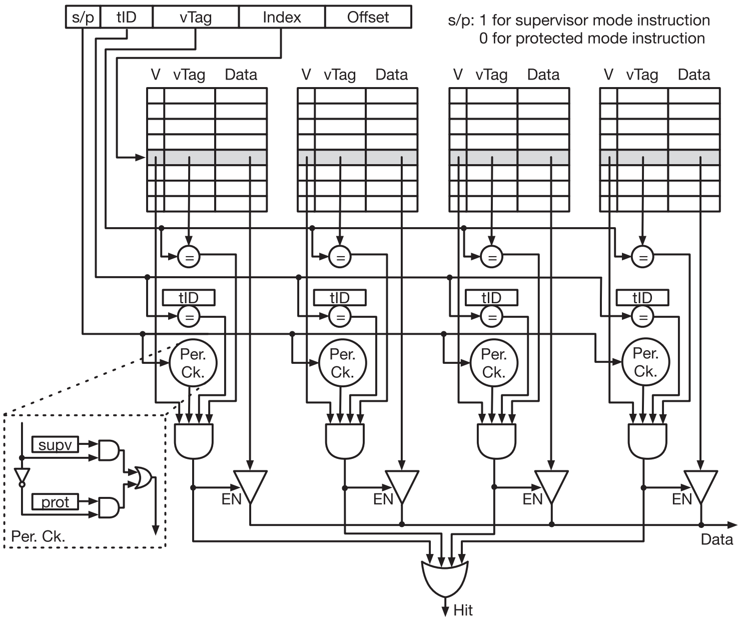
s/p
tID
vTag
Index
Offset
s/p: 1 for supervisor mode instruction
0 for protected mode instruction
V vTag Data
tID
Per. Ck.
supv
prot
EN
Data
Hit

Fig. 2. An illustration of a 4-way SD$ architecture; a way can be allocated to a hardware-thread by storing its (OS-assigned) hardware thread ID (tID) and protection bits (supv and prot).

(3) *Metadata per way:* For secure allocation of SD$ ways, we require: (a) *Two protection bits:* To restrict some ways to kernel-mode access, the protection bits indicate if an SD$ is accessible by a hardware thread running in kernel mode (supv=1) and user mode (prot=1). (b) *Hardware thread ID bits:* Since SD$s are attached to cores, each way must store the ID of the hardware thread that access it. Note that these bits are managed via the supervisor-mode `SpxManage` instruction and their purposes are summarized in Table 3 (Section 5).
(4) *Data storage:* With 64-byte blocks and an 8-way set associative SD$, every set consists of 8 cache blocks or 512 bytes. With 64 blocks in each way, up to 32 KB of data can be stored.
(5) *Hit/miss comparators:* Hits and misses are determined by comparing the virtual tag of the request with the stored tags in the appropriate set. The data array is accessed in parallel with the tag array for fast accesses.
(6) *Accessing data:* Data is returned to the core only if (i) the block is valid, (ii) the tag-bits of the block match those of the incoming request, (iii) the block corresponds to a way for which the requesting hardware thread's tID matches, and (iv) the way is accessible, as determined by the protection bits. Thus, a hardware-thread can access only its private data, enabling secure processing of thread-specific sensitive information.

4.3 Handling Memory Requests

When an SPX64 instruction accesses the SD$ for a block that is present (analogous to a cache hit), the SD$ provides or updates data accordingly. The access latency is lower than an L1-DCache, because there is no need for replacement bit management. For write hits, the SD$ does not generate coherence traffic, further reducing energy and latency. When a block corresponding to the request is not present in the SD$ (analogous to a cold miss), the SD$ hardware issues a *non-cacheable request* to the L1-DCache for the block. In this case, the virtual address needs to be translated to the physical address through the DTLB, which can be done concurrently with searching the data in the

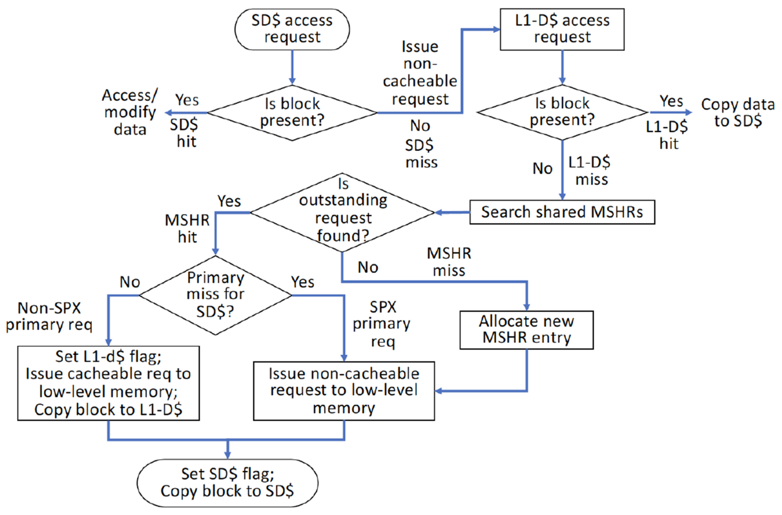

Fig. 3. Flow of the execution events for processing an SD$ access request.

SD$. When the data block is already present in L1-DCache (due to previously executed non-SPX64 instruction corresponding to the program functionality), it is returned to the SD$. Otherwise, a request to lower-level memory is initiated based on the status of MSHRs. SPX64 instructions are implemented as non-cacheable memory instructions that should not have any effect on the cache coherence state of cache lines residing in the lower-level of the cache hierarchies (e.g., L1-DCache and LLC). It ensures that an SPX64 instruction does not alter the state of the L1-DCache. Figure 3 illustrates an example about flow of various events regarding the memory accesses to SD$.

To consolidate outstanding misses, SPX64 shares MSHRs between the SD$ and the L1-DCache (Figure 1). Each MSHR entry has two 1-bit flags to indicate the origin of each miss (SD$ and L1-DCache). For example, if an SpxRead misses, and also misses in the L1-DCache, the shared MSHRs are searched. A hit in the MSHR indicates that a primary/outstanding miss had already happened and was either initiated by a non-SPX64 memory access or an SPX64 memory access. On a miss in the MSHR, a new MSHR entry is allocated. In both cases, a target in the address stack is created corresponding to the SPX64 instruction and the SD$ flag for the corresponding MSHR entry is set, indicating that the returned block should be copied to the SD$. When the primary miss is initiated by an SPX64 memory access, the returned block is copied to *only* SD$, whereas it is copied to both SD$ and L1-DCache, when the primary miss is triggered by a non-SPX64 memory access. In the later scenario, both MSHR flags are set, indicating the copying to both the L1-DCache and SD$. Similarly, when an L1-DCache miss is triggered by a non-SPX64 request, an MSHR entry is created (if not already present). A target is created corresponding to the non-SPX64 instruction and the data cache flag is set high for that MSHR entry to indicate that the returned block should be copied to only L1-DCache. Note that SD$ uses virtual address, whereas the shared MSHRs use physical address. When the SD$ flag is set, the MSHR entry also stores the allocated location in the SD$ (way and set IDs) to avoid reverse address translation.

The SD$ is not coherent, updates caused by `SpxWrite` are invisible to other hardware threads, and cannot be read/updated using non-SPX64 instructions. Subsequent coherence events on the concerned block are not conveyed to the SD$, and any write-back to main memory via the cache-based hierarchy happens only if it is desired by the programmer. The latency model for coherence updates used in our experiments is the cache access latency (2–4 cycles). However, additional

Table 2. Example Encoding of SPX64 Instructions

Instruction	`SpxReadB`	`SpxReadW`	`SpxWriteB`	`SpxWriteW`	`SpxClear`	`SpxZero`	`SpxInv`	`SpxManage`
Opcode	0F 6C	0F 24	0F 7C	0F 7D	0F 18 18	0F 0C	0F 0D	0F 04

queuing delay can increase the response latency. The SD$ is a non-coherent cache structure, so it will not be snooped during coherence events. This leads to savings in both energy and latency.

Before write-back, programmers must use `SpxRead` to bring data from the SD$ into registers and then use regular stores to update the L1-DCache. This regular store instruction following the SpxRead instruction is responsible for updating the cache line coherence state in L1-DCache. If the cache line hits L1-DCache, the coherence state of the cache line is changed to a Modified state and contents of the cache line are updated. If the cache line misses in L1-DCache, it is fetched from the memory hierarchy as store miss request. Note that it is not possible for another hardware thread to read/alter the content of the SD$ allocated to the target thread. Thus, the proposed write-back scheme ensures safe propagation of sensitive data from the SD$ to the main memory.

Since SPX64 resources are managed by software, the programmer must ensure that there are enough SD$ blocks for program execution. On an SPX64 miss, block insertion and replacement are simple: A newly brought in block is inserted at the location of the first empty (invalid) block encountered in the allocated SD$ ways, and then the valid-bit is set. If no such block is found, an exception is raised.

4.4 SPX64 Front-end Pipeline

The decode unit of the processor core must also be extended to support SPX64 functionality. After decoding a memory instruction, it is assigned the following metadata, which is propagated through the following microarchitecture structures: one bit indicating if the memory request is an SPX64 instruction; one bit indicating loads versus stores, and two additional one-bit flags to indicate `SpxClear`, `SpxZero`, or `SpxInv`.

Table 2 presents an example encoding of the instructions from Table 1 into the x86_64 ISA. For example, `SpxReadB` results in the opcode `0x0F 0x6C`, instead of `0x8A` for a regular load-byte request. Note that processing of the macro/micro-ops is not altered for SPX64 instructions: They are decoded, dispatched, and executed by the pipeline like regular instructions. Thus, `SpxRead` and `SpxWrite` instructions can be processed like regular `mov` instructions, i.e., with the same prefix and suffix byte-codes, but are executed with different op-codes.

4.5 Load-store Queue Architecture for SPX64

The design of the load-store queue for SPX64 (SPX64-LSQ) is similar to a conventional LSQ design. However, the proposed SPX64-LSQ also processes `SpxRead`, `SpxWrite`, `SpxClear`, `SpxZero`, `SpxInv`, and `SpxManage`. Regular loads and stores (i.e., `mov` instructions) are processed in the same manner as they are in a conventional LSQ. For example, a regular load searches for older regular stores in the store queue for forwarding opportunities before accessing the data cache. However, since both SPX64 and regular memory instructions can enter the SPX64-LSQ, it must employ additional measures to ensure that the data can be safely forwarded among the correct instruction types and that it is safe to issue read/write requests for their propagation to the SD$ or lower-level memories in the hierarchy.

For processing SPX64 instructions, SPX64-LSQ can typically forward the data for an `SpxRead` from a `SpxWrite` with matching address, returning the data to the execution pipeline. `SpxRead` can also forward the data with value zero from an older `SpxZero` (that set a block to all-zeros) with matching block address. The SPX64-LSQ ensures that the sensitive data in the SD$ is managed

Table 3. Additions to the Architectural State of the CPU

# Bits	Name	Location	Purpose
$1 + log_2(HT)$	`sp_id`	HW thread	ID used when accessing scratchpad.
1	`has_prot_sp`	HW thread	Flag allowing SD$ access from protected mode.
1	`has_supv_sp`	HW thread	Flag allowing SD$ access from supervisor mode.
$1 + log_2(HT)$	`id`	SP Way	ID of hardware thread allowed to access this way.
1	`prot`	SP Way	Flag allowing SD$ access from protected mode.
1	`supv`	SP Way	Flag allowing SD$ access from supervisor mode.

securely while copying the latest version of data from the normal memory hierarchy when accesses miss in the SD$. To enforce this, an SPX64 instruction cannot forward from a regular instruction or vice versa. Moreover, `SpxRead` is stalled if it encounters an older regular store instruction with matching or unresolved address, which is not committed yet to the memory. Since the SPX64-LSQ is unaware of the state of the SD$, it must consider the possibility that the `SpxRead` requires loading the data from lower memory to the SD$ (compulsory SPX64-miss). Therefore, the SPX64-LSQ skips forwarding and stalls the `SpxRead` until the regular store (with matching address) commits.

Before forwarding data to an `SpxRead` instruction from an `SpxWrite` (with matching address), SPX64-LSQ checks for intervening `SpxInv`, `SpxClear`, or `SpxManage` instructions. If found, data is not forwarded, and the `SpxRead` stalls until the conflicting instruction commits. Stalling ensures that if some data is evicted from the SD$ by a hardware thread, the `SpxRead` (with the matching address) cannot access the same data from the SD$. Finally, `SpxClear`, `SpxZero`, `SpxInv`, and `SpxManage` can be issued to the SD$ only when they reach the head of the ROB. To eliminate security vulnerabilities, the SPX64-LSQ does not allow speculative issuing (i.e., when following SPX64 stores with unresolved addresses) of `SpxRead` to the SD$. However, regular loads can be issued speculatively to the normal memory hierarchy to boost performance by prefetching to the L1-DCache.

5 OPERATING SYSTEM SUPPORT FOR SPX64

The OS is responsible for resource management, virtualization, and security of the SD$. The supervisor-mode `SpxManage` instruction is the primary mechanism for doing so. Recall from Section 4 that an SD$ is attached to a core, and its ways are allocated among the threads of that core. A way is protected in two dimensions: by the thread allowed to access it and by the protection level at which that thread must be operating. For simplicity in our presentation, we only consider two protection levels, corresponding to user mode (`prot`) and kernel mode (`supv`). Extending to additional levels is straightforward.

Let HT be the number of hardware threads per core. Each is assigned $(1 + log_2(HT)) + 2$ bits of new architectural state: 1-bit fields `has_prot_sp` and `has_supv_sp`, and the $1 + log_2(HT)$-bit field `sp_id`. For simplicity, we assume that these bits reside in a single **model-specific register (MSR)** that can be read and written via existing supervisor-mode instructions.

In addition to the bits of metadata (tag and valid bits) and block storage required to implement the SD$, it must also contain $(1 + log_2(HT)) + 2$ bits of metadata per way. The first $1 + log_2(HT)$ bits represent the hardware thread ID (`id`) that is allowed to access the scratchpad. The remaining bits serve as Boolean flags, indicating whether that thread is allowed to access the SD$ way from `prot` and/or `supv` mode. A summary is provided in Table 3.

Managing the above information is achieved through a single new supervisor-mode instruction, `SpxManage %r1 %r2`. `%r1` holds an unsigned integer representing the SD$ way being managed.

%r2 holds $(1 + log_2(HT)) + 2$ bits, representing the hardware thread ID required for accessing the way, and flags for whether that thread may access the SD$ from prot and supv modes.

We assume that the OS will not assign the same sp_id value to multiple hardware threads of the same core at the same time. Under such a scenario, two threads could simultaneously access the same SD$, leading to undefined behavior. Furthermore, note that prot mode does not imply supv mode: The OS can assign prot=1 and supv=0, so a program may access its SD$ but the OS cannot. We allow this behavior in case a system has a trusted hypervisor and untrusted guest OS. Last, note that with HT hardware threads per core and two modes, the ways of the SD$ can be divided into as many as $2 \times HT$ distinct sets. The extra bit in the sp_id and id fields accommodates this possibility.

In addition to the above resource management functions, an OS must have a means of delivering signals to a program when it uses the SD$ incorrectly. There are two conditions that can generate a signal. The first is the case in which a thread executes one of the five SPX64 instructions but has_prot_sp is not set. The second case is when a thread has W' ways allocated to it, an access to the SD$ requires a block to be allocated, but there are no remaining ways within the set.

6 PROGRAMMING MODELS

Clearly, the OS must provide a way for a program to request that SpxManage is executed on its behalf and to release its SD$ when it is no longer in use via two system calls. The two system calls are SpxAlloc and SpxRelease. The guarantees that the OS provides to the program between these two system calls are the basis for the programming models that SPX64 can provide to software.

The SpxAlloc system call reserves ways of an SD$ for use by a process, and SpxRelease returns those ways to the OS when the thread is finished using them. SpxRelease takes no parameters and has no return value. SpxAlloc has a single parameter: the amount of memory requested. It may return -1 in the case where the request cannot be granted. Otherwise, it returns s, the size in bytes of the allocated region. The OS is free to return more memory than requested, i.e., by rounding up to a multiple of the size of a way. Note that as part of SpxRelease, the OS must temporarily assign the SD$ to itself, so it can SpxClear it, and must also clear the calling thread's has_prot_sp bit. SpxAlloc must assign the thread a unique (across threads of the core) sp_id and set has_prot_sp.

Internally, the OS must track how each way of each SD$ is allocated. At a bare minimum, this is required to ensure that an SD$ is not taken away from a thread accidentally. Given the expanded range of thread IDs, the OS may also reserve some ways of each core's SD$ for itself, e.g., as a way to protect its secrets from Meltdown-style attacks [29]. Note, too, that as defined, an SD$ is not cleared upon context switch. These characteristics provide the foundation upon which the OS can introduce policies for how SD$ resources are managed upon a context switch, and hence the SPX64 programming models. Below, we summarize three programming models for SD$. Table 4 contrasts these programming models in terms of the workloads for which each is well-suited.

6.1 Model #1: Static Assignment

A general-purpose SD$ must remain secure in the event of context switches. An SD$ (or part of it) can be dedicated to a performance-critical application, in which case the OS could leave a switched-out thread's data in the scratchpad and simply deny SpxAlloc requests by the switched-in thread. This technique is simple to implement: The new thread can be denied access to the SD$ by simply clearing the *has_prot_sp* bit. Virtual addressing of the SD$ is a fundamental enabler of this approach: Even if the switched-out thread's pages are all flushed to disk, and when the thread is switched back in, it is allocated an entirely new set of physical pages, its contents in the SD$ remain correct, because the SD$ is virtually tagged, and the virtual addresses did not change, regardless of TLB remapping.

Table 4. Appropriate Workloads for Each Programming Model

Model	**Workload Characteristics**
Static Assignment	Appropriate for embedded and compute-intensive workloads; poor fit for workloads with high rates of context switching (e.g., web services)
Direct Mapped	Appropriate for use in a single kernel with regular access pattern (e.g., embedded workloads, security applications)
Best Effort	Appropriate for workloads that leverage speculation (e.g., transactional workloads); requires programmer-provided fallback code.

The downside of this approach is that threads cannot migrate once they have been assigned SPX64 resources statically. An OS designer may choose to make extensive modifications on account of this constraint. Examples include incorporating knowledge of SD$ requirements when mapping processes to cores or incorporating past SD$ use into affinity scheduling and using the SD$ to protect the OS from malicious device drivers. Note that the first call to `SpxAlloc` by a thread serves as a just-in-time declaration of the resources needed by a thread and that the OS may need to context switch threads immediately after a `SpxRelease` to reallocate SPX64 resources to waiting threads.

6.2 Model #2: Effectively Direct Mapped

While the proposed SPX64 design is associative, a program is free to use its SD$ resources as a contiguous array of data (e.g., the example in Section 3.1 and the discussion at the beginning of Section 3). For this use case, a small amount of additional information enables simple virtualization of the SD$ upon context switches. This new information, provided by the `SpxRange` syscall, informs the OS that a specific range of virtual addresses (provided as a base and an offset) are mapped to the SD$. Again, since the SD$ is virtually tagged, both the OS and the process can use the same addresses to access the same data within the scratchpad. There is no address remapping.

If a thread has called `SpxRange`, then upon a context switch, the OS can copy the SD$ contents to a kernel buffer: As long as the `supv` bit is set for each way and `has_supv_sp` is set, the OS can assign to itself the same `sp_id` used by the switched-out thread. It can then use `SpxRead` instructions, starting at the given base virtual address, to read the SD$ contents to registers and then write them to another space in memory. To restore the SD$ before switching the thread back in, the OS performs these steps in reverse, using `SpxWrite`. In this scenario, the OS can store the contents of the SD$ in the thread control block, and thus the thread can migrate to another CPU without losing its SD$ contents.

6.3 Model #3: Best Effort

In Section 3.2, the SD$ served as a sparse shadow copy of the program's virtual address range. Here, `SpxRange` is not useful, because the associativity of the SD$ is allowing the efficient storage and retrieval of non-contiguous data. This use case coincided with two other properties: All of the writes to the SD$ were also shadowed in a vector, and the uses of the SD$ were within a transaction that could roll back. For this use case, total required capacity may not be known in advance, and hence the thread must already be able to handle an inability to use the SD$. Thus, on a context switch, the OS can simply revoke the thread's access to SPX64 resources, clear the thread's allocation of the SD$, and leave it up to the thread to recover when it is swapped back in.

Upon an exception under this "best effort" model, the thread could abort its transaction, request new SPX64 resources, and restart. However, more fine-grained recovery is possible. If the exception is because `has_prot_sp` is unset, then, since the thread has a vector storing its prior `SpxWrites`, it

Table 5. Baseline Configuration

Core Parameters	7-stage OoO, no SMT, 4-issue, 2.6 GHz, 22 nm, 224 ROB, 128 LSQ, 180 INT/180 FP PRF, 8 INT/ 4 MEM/ 6 FP/ 4SIMD FUs, Tournament branch predictor, 4,096 BTB entries, 16 RAS entries
L1 ICache	32 KB, 8-way, LRU, 64 B block, 1 port, 3 cycles, 4-entry MSHR
L1 DCache	32 KB, 8-way, LRU, 64 B block, 3 ports, 4 cycles, 4-entry MSHR
L2 Cache	2 MB, 16-way, LRU, 64 B block, 1 port, 20 cycles, 20-entry MSHR
Main Memory	8 GB (x8 I/Os), DDR4-2400 17-17-17, 1 channel, 2 ranks, 8 banks, tRAS=32 ns, tXAW = 21 ns, tWR = 15 ns, tRTP = 7.5 ns, tRTW = 1.666 ns

Table 6. Modifications for SPX64 Architecture

SD$	32 KB, 8-way, 64 B block, 2–4 cycles, 1 port
Shared MSHRs	8 entries
SPX64-LSQ	128 entries

can request a new SD$, re-populate it by re-issuing `SpxWrites` with values from that vector, and then resume.

Re-creating the SD$ can be beneficial. Recall that the thread must use `SpxRead` for all accesses to program data within a transaction so it can see its pending `SpxWrites`. Many of these reads will be to locations for which there was no prior `SpxWrite`, which means they waste the capacity of the SD$. For programs with reasonable spatial locality, it may be beneficial to explicitly `SpxClear` and then re-create the contents of the SD$ mid-transaction, or upon a SPX64 capacity exception, to "trim" from the SD$ lines that were read but never written.

7 SIMULATION CONFIGURATION

We simulate SPX64 using the gem5 cycle-accurate, execution-driven simulator [6]. We modified the O3CPU, X86 ISA, and the classic cache model in gem5 to provide support for SPX64 instructions and to model the functionality and timing of the proposed architecture.

Table 5 lists architectural configuration parameters. The processor parameters are adapted from the Intel Skylake microarchitecture [21], except that we only simulate a single core with a shallower cache hierarchy with timing parameters verified from CACTI [4]. These decisions were made to simplify the simulation and reduce overhead while retaining as much realism as possible. Main memory parameters are from the Micron MT40A1G8 device [32]. We used McPAT [28] to estimate system power consumption and on-chip area and CACTI [4] to model SPX64 implementation at 22 nm. Table 6 summarizes the SPX64 configuration. According to CACTI, the SD$ has a 2-cycle access latency. However, since it adds to the overall floorplan, we consider this a best-case estimate, and our experiments vary SPX64 access latency from 2 to 4 cycles. The exception raised due to capacity or associativity constraints that prohibits the allocation within the SD$ is currently handled by the programmer. A runtime library can be implemented to handle the same, which is left for future work.

8 EVALUATION

To evaluate the impact of SPX64 on applications where traditional scratchpads are known to provide benefits, we consider workloads from the MiBench suite [20]. To evaluate the effect of SPX64 in security-sensitive applications, we measure microbenchmarks that use open-source implementations of important cryptographic algorithms. Last, we study the benefits that associativity offers

Table 7. Total Committed Instructions for the SPX64-version of Applications Normalized to the Baseline

Application Type	Applications	Committed Instructions
Security	AES-GCRYPT	1.05
	AES-OPENSSL	1.04
	DES-GCRYPT	1.21
	DES-OPENSSL	1.13
	RSA-GCRYPT	1.02
Persistent	TATP_OPT	0.95
	TPCC	0.91
	VACATION	0.99
Embedded System	STRINGSEARCH	1.24
	DIJKSTRA	1.14
	ADPCM_ENC	1.05
	BASICMATH	1.01

to workloads that interact with non-volatile memory via a transactional interface. This section also presents evaluation results on the total power and area overhead. To use SPX64 in the MiBench and cryptographic workloads, we identified the variables that we wanted to place on the scratchpad and then changed the type of each variable to a simple C++ template. The template ensured that each load and store used the appropriate SpxRead or SpxWrite assembly instruction, respectively. This approach minimizes source code modifications, but results in poor register utilization, since the template chooses which registers to use for data movement, instead of the compiler. As we will show, SPX64 is still able to improve performance, even though this strategy increases instruction counts and register pressure. For the persistent workloads, a transactional compiler inserts the SPX64 calls using the same static register allocation policy. The total instruction counts are reported in Table 7 for each instrumented application. The impact of the increased instruction counts on performance will be discussed for each group of applications.

8.1 Performance

8.1.1 Security Applications. To determine the effectiveness and performance impact of SPX64 for security-critical workloads, we developed microbenchmarks that stress-test five algorithms from the OpenSSL and Libgcrypt libraries: AES, DES, and RSA. For each benchmark, we protect memory access patterns by placing sensitive data in the SD$. We consider two variants of SPX64: one in which SpxRead instructions are allowed to issue speculatively and one in which they are not.

We begin with the case in which SpxRead is allowed to issue speculatively. Figure 4 presents the performance of five systems, normalized to a baseline 32 KB L1-DCache (Table 5). The spx-2, spx-3, and spx-4 systems have a 32 KB SD$ with 2-, 3-, and 4-cycle access latency, respectively. These configurations let us study the best, common, and worst-case latencies for SPX64. Since these systems have 64 KB of memory close to the pipeline, we also compare against two systems with larger L1-DCaches. The base64-4 and base64-5 systems have a 64 KB L1-DCache and with 4-cycle (optimistic) or 5-cycle (more realistic) access latency.

As Figure 4 shows, doubling the size of the L1-DCache to 64 KB does not boost performance. This is because the selected workloads have relatively small working sets, and the sensitive data exhibits high spatial and temporal locality. Thus, the sensitive data is never evicted from the 32 KB L1-DCache of the baseline. The remaining data is accessed in a streaming fashion, and thus increasing cache capacity does not provide any benefit. We confirmed this by measuring miss rates: The miss rate remains the same between 32 KB L1-DCache (baseline) and 64 KB L1-DCache (base64-4). This result also indicates that our SPX64 design is not benefiting from larger space for these workloads.

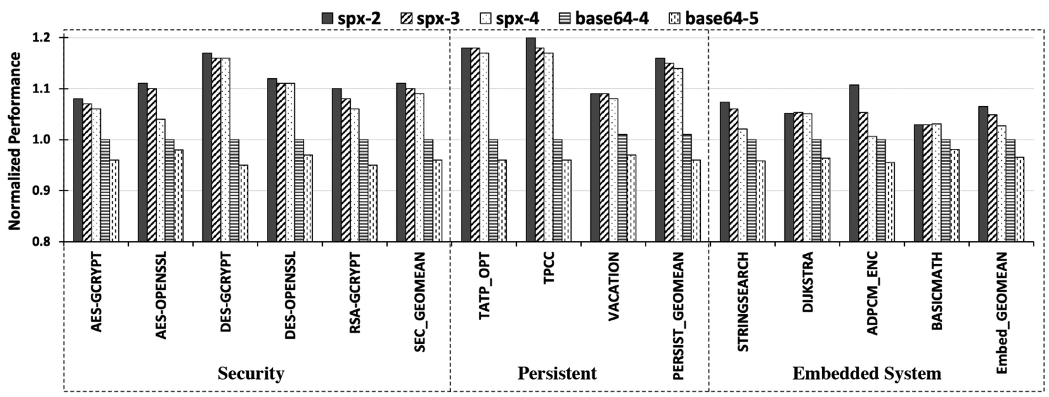

Fig. 4. Performance normalized to a baseline with a 32 KB L1-DCache; spx-n: SPX64 design with n-cycle SD$ latency; base64-4: a baseline with a 64 KB L1-DCache with impractical 4-cycle latency; base64-5: a more realistic baseline with a 64 KB L1-DCache with 5-cycle latency.

The normalized performance results (higher is better) shown in Figure 4 indicate that for each of the evaluated security applications, the SPX64 design improves performance, with gains proportional to the access latency. Short latency (2 cycles) is conceivable because of the fact that the SD$ is only a single-port cache design, does not reply to cache coherence snoop calls from a lower level cache, and does not need to maintain any replacement logic for evictions on set conflict. Note that while we configured the SD$ with a 32 KB capacity, in these workloads the sensitive data is only 4 KB. Furthermore, it is contiguous and thus can fit in the SD$ even when the thread is only allocated a single 4 KB way or when the SD$ is only 4 KB.

In lieu of a complete place and route of all the core components, we conducted a sensitivity study that varied the SD$ access latency from 2 cycles to 4 cycles (4 cycles matches the baseline latency). Even when SPX64 latency is as high as the baseline, it still performs better. For example, spx-4 reduces execution cycles by 8.5%. This is because the miss penalty on a cold miss encountered by SPX64 is smaller than the miss penalty for the L1-DCache. The lower miss penalty stems from two main factors. First, an SPX64 cold miss on its refill/response path from DRAM bypasses the L1-DCache and L2-Cache, because SD$ does not change the status of the lower levels of the memory hierarchy and does not need to support coherence, as discussed in Section 4. Second, recall that the **Write-Queue or Eviction Queue (WQ)** is an optimization in modern architectures to allow demand reads to be scheduled before writes. It delays some writes to main memory, since they are not on the critical path. SD$ does not perform any write-back or evictions, and thus SD$ is independent of a WQ that is required for looking up L1-DCache misses prior to accessing the L2-Cache. Note that this performance increase is despite an increase in the instruction count (Table 7).

Our current approach to code generation for SPX64 instructions uses the base addressing mode, whereas the baseline (non-SPX) code generation is able to use all of the x86 addressing modes (i.e., any variation of *base+(index*scale)+displacement*). Consequently, our SPX64 binaries have a higher instruction count due to additional instructions for calculating addresses. In addition, these address computation instructions can result in more register spills than the baseline. Table 8 shows the correlation between the percentage of SPX64 instructions and the increased number of instructions versus the baseline. Each conversion from a `mov` to a `SPX64 mov` instruction causes one to four additional instructions. For the security workload, preloading data exploited by cache side channels (i.e., lookup tables) into the scratchpad also adds additional instructions, but the percentage of additional instructions due to preloading is less than 0.3%. Despite the increased

Table 8. Correlation between Number of SPX64 Instructions and Instruction Overhead for SPX64 Benchmarks

Benchmarks	SPX64 instructions	Instruction overhead
AES-GCRYPT	4.12%	5%
AES-OPENSSL	1.49%	4%
DES-GCRYPT	6.71%	21%
DES-OPENSSL	4.51%	13%
RSA-GCRYPT	1.61%	2%

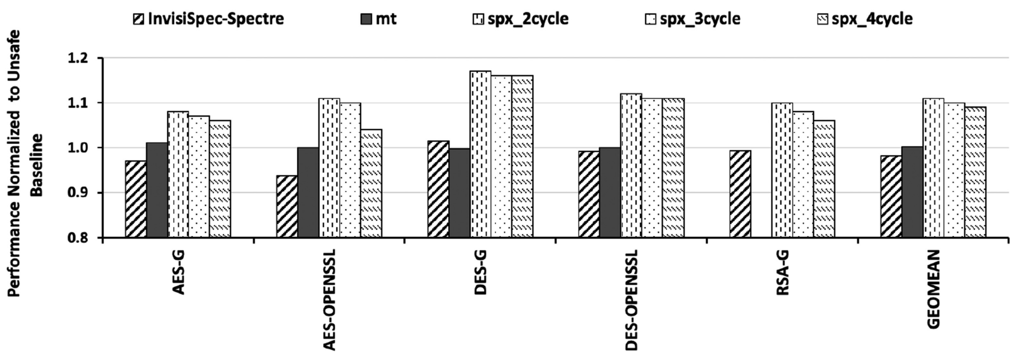

Fig. 5. Security microbenchmark performance normalized to an unsafe baseline; InvisiSpec-Spectre: InvisiSpec design with data filter cache; mt: MuonTrap design variant for Spectre protection; spx-2: SPX64 design with 2-cycle SD$ latency; spx-3: SPX64 design with 3-cycle SD$ latency; spx-4: SPX64 design with 4-cycle SD$ latency.

instruction count, we see that the lower latency of the SD$, and the simpler design, still produces an advantage relative to the baseline.

Figure 5 compares our SPX64 design with MuonTrap [2] and InvisiSpec-Spectre [42]; results for the open-source InvisiSpec and MuonTrap are produced for the security microbenchmarks on the same X86 system. RSA-G microbenchmark did not run on MuonTrap code, therefore the result for RSA-G is not included. All the results are normalized to the unsafe baseline (higher is better). InvisiSpec performed about 0.98× compared to unsafe Baseline, and MuonTrap performed the same as baseline.

The above discussion considered an SPX64 implementation in which SpxRead instructions could be executed speculatively. This design is vulnerable to Spectre [25] (Variant 1) attacks (within-process attacks). Since we have already isolated the sensitive data of the application into the SD$, we can trivially prevent this Spectre variant by introducing a non-speculative SpxRead instruction.

MuonTrap [2] and InvisiSpec [42] do not provide complete protection within the same process. MuonTrap uses software mitigation, which requires to clear the filter cache when entering and exiting a sandboxed region of code. In the case of InvisiSpec, an attacker within the same process can request the non-speculative data followed by victim's speculative load. The attacker's speculative load within the same process will lead to a hit in the speculative buffer for victim's load, which may result in new covert channel.

We further extend the SPX64 design to protect against Spectre [25] (Variant 1) within process attacks by allowing SpxRead access SD$ only when it reaches the head of the ROB. We call this version spx-2-Spec-Safe. It has on average a 12% slowdown compared to the unsafe baseline in Figure 6. However, this performance is still more than 2× faster than the Speculation-Safe, in which

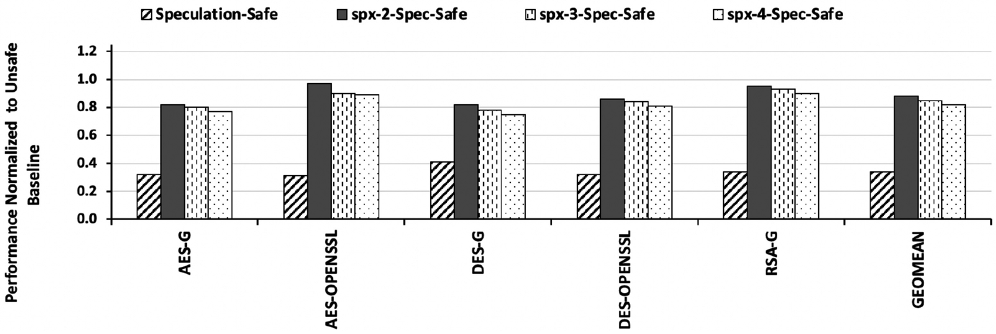

Fig. 6. Security microbenchmark performance normalized to an unsafe baseline; Speculation-Safe: baseline with non-speculative load instructions; spx-2-Spec-Safe: SPX64 design with non-speculative SpxRead instructions and with 2-cycle SD$ latency; spx-3-Spec-Safe: SPX64 design with non-speculative SpxRead instructions and with 3-cycle SD$ latency; spx-4-Spec-Safe: SPX64 design with non-speculative SpxRead instructions and with 4-cycle SD$ latency.

all the load instructions are non-speculative. The proposed SPX64 can strike a balance between performance and security by selectively making certain loads non-speculative.

Using SPX64 in this manner is a remarkably simple way to protect against cache-based side-channel attacks. In our initial benchmark implementation (Figure 4), we only needed to change one line of code to allocate the sensitive data in the SD$; to use nonspeculative loads, we only had to change that same line. Our libraries and compiler extensions did the rest of the work of replacing loads and stores with SPX64 instructions. Thus, the effort to use spx-2-nonSpec is equivalent to the effort to use safe_baseline. For this effort, the performance degradation in the microbenchmark is 12%, versus 60% for Speculation-safe baseline. We expect that it will be possible to reduce this slowdown further. One promising strategy is to use a hybrid approach (similar to the one used in MuonTrap [2]) that combines the isolation provided by the default SPX64 design with software mitigation techniques.

8.1.2 Persistent Applications. We used the open-source LLVM persistent transaction plugin [47] to instrument and optimize our persistent workloads. The plugin provides a suite of different PTM algorithms, to which we added variants that employ SPX64 to reduce the overhead of redo logging. As described in Section 3.2, each write is performed twice, to the SD$ and to a lightweight redo vector, and each read passes through the SD$. This ensures that reads have processor consistency, and, since there are no lookups in the redo vector, it can use a simpler implementation.

In our implementation, the SPX64 instructions are inserted into the PTM library via in-line assembly. Unlike Section 8.1.1, we are not concerned with the impact on code generation, since the PTM library code already produces tens of instructions per load or store of nonvolatile memory.

We consider persistent benchmarks with different redo log sizes from DudeTM [31]: the TPCC transaction processing benchmark and the TATP telecom application benchmark. We ran the New Order TPCC benchmark and tested Update Location transactions for TATP. Both benchmarks use a persistent hash table for the index. In addition to a standard application of SPX64, we also consider an optimized use in TATP-OPT, where we use SpxZero to avoid fetching data for blocks that are only written. We also measure the Vacation benchmark from the Whisper benchmark suite [33]. (Note that Vacation is the only benchmark from Whisper to use a transactional interface.) Vacation simulates a travel reservation system; the clients and server run in the same process, and clients make requests from a collection of persistent data structures managed by the server.

Table 9. Speedup for TATP Variants

TATP Variation	SD$ Hit Rate %	Speedup (spx-2 cycle)	spx-3 cycle	spx-4 cycle
TATP	50	1.06	1.05	1.05
TATP_OPT	79	1.18	1.18	1.17

As shown in Figure 4, the performance of TATP-OPT, TPCC, and Vacation is improved by using SPX64. One of the key factors to understand the performance improvement in persistent benchmarks is the size of the redo log. In TPCC, the write sets are much larger than they are in TATP, and there is more opportunity for a SpxRead to hit in the SD$. Even with a small redo log, the hit rate in TATP is 50%. The optimized TATP shows a significant improvement, reducing latency by 18% and increasing the hit rate by 29% (Table 9).

To have a fair comparison with TATP-OPT, we developed TATP-Prefetch-Baseline version, which adds 4,096 software prefetch instructions before transactions start to avoid demand misses on variables being placed in SD$. We found out that TATP-Prefetch-Baseline performs same as baseline. We also compared them on real machine [Intel Xeon Silver 4214 CPUs] and found the same outcome. This is because prefetches still move data, which adds bandwidth overhead even if they are early enough to avoid demand misses.

The redo log size for Vacation is in between the redo log sizes of TATP and TPCC. In spite of having a 85% SD$ hit rate, which is more than it is in TATP and TPCC, the performance improvement for Vacation is about 9% lower. This is because there are fewer SPX64 accesses on the critical path in Vacation. Table 7 shows that the number of committed instructions decreases for each benchmark: For all the persistent workloads, replacing hash table lookups with SD$ accesses reduces the instruction count (Section 3.2).

8.1.3 Embedded Systems Applications. To evaluate the performance improvements for workloads with high data locality, we evaluated four embedded system applications from MiBench [20]. **Adaptive differential pulse code modulation (ADPCM)** is a popular algorithm used by wireless communication devices for compressing speech samples. Basicmath evaluates various kernels involving mathematical operations such as cubic function solving, square root, and angle conversions. Embedded processors in network devices perform shortest path calculations, which are evaluated by Dijkstra's algorithm. Finally, stringsearch is used in text manipulation algorithms for searching given words in the phrases of text. We evaluated these benchmarks with "small" input data sets. We manually identified the data structures that exhibited high spatial and temporal locality and managed them in the SD$. For example, the node data structure and the adjacency matrix are frequently accessed during shortest path calculations and therefore can benefit from fast SD$ accesses. Similarly, tables for manipulating strings or step index calculations for PCM exhibit frequently accessed and co-located data, which we managed in the SD$.

Figure 4 shows execution cycles for our design, normalized to the baseline. We observe that our design efficiently handles various access patterns for different data types and data structures with a low access latency and improves the performance up to 10%. For example, the spx-2 design outperforms application executions on cache-based designs and improves overall performance by 1.06× (geomean). As noted before for security workloads, performance for embedded systems applications is also improved by SPX64 with even higher access latencies due to reduced miss penalty and independence from a WQ. Thus, as compared to cache-based executions, our SPX64 design offers low access latency and data management capabilities to the programmer, just like software-managed executions on conventional scratchpads. However, as compared to scratchpads, it alleviates the programming complexity by providing programmers or the compiler with a flexible way to manage high-locality data and even data of various structures while retaining the generality of caches and avoiding the need for DMA transfers.

As with the security benchmarks, we expect these results to improve as we refine our compilation toolchain. Presently, the generated code for these workloads (featuring SPX64 instructions) exhibits on average 11% more total committed instructions (Table 7) than the baseline. Integrating awareness of SPX64 into the code generation process would decrease this overhead.

To demonstrate managing large footprint of workloads on SPX64 system, we evaluated benchmarks with large input data. Among these benchmarks, for ADPCM and stringsearch, the large input data (25 MB and 100 kB, respectively) did not fit in SD$ and required evictions and accesses to lower-level memories. So, we allowed managing only a fraction of the input stream (e.g., a tile of 16,000 samples of short integers in ADPCM) in SD$ along with highly reused data structures (e.g., two small lookup tables in ADPCM). After processing each tile, the program execution issued SpxClear instruction to flush entire SD$. Then, it continued to access the next tile of the data stream along with the reusable data arrays. With low latency of accessing isolated data from SD$ and high SD$ hit-rate (99% for both), we observed about 9.4% and 5% performance improvement, respectively, as compared to the baseline.

8.2 Power Consumption

As mentioned in Section 8.1.1, SPX64 can improve performance by migrating frequent accesses from a large and complicated L1-DCache with a relatively long access latency to a simple SD$ with a relatively short access latency while providing security benefits. The total number of L1-DCache accesses and the power consumed by the L1-DCache ought to decrease accordingly. For the security applications, we observed that L1-DCache accesses decreased by 37%, and L1-DCache energy consumption decreased by 16%. Core and L1-ICache power increased, because of the increased instruction count. Overall, as compared to the baseline, the SPX64 design with a 2-cycle access latency speeds up the execution by 1.12× (geometric mean) with a 3.6% increase in total power consumption (Figure 7).

For persistent applications, we observed an average reduction of 5% in committed instructions, and thus a 1.3% reduction in L1-ICache power consumption. Additionally, the frequency of L1-DCache accesses decreased by 10%, causing a 12% reduction in its power consumption. Overall, the SPX64 design with a 2-cycle access latency speeds up the execution by 1.16× (geometric mean) with a 0.7% reduction in energy consumption (Figure 7).

In the embedded workloads, which we used to approximate the impact of SPX64 for applications with high locality, SPX64 increased total power consumption by 0.41% (geomean). Major power reductions are observed for the L1-DCache and L2-Cache, because frequently accessed data arrays (e.g., the adjacency table and tables for string manipulation) were kept stationary in the SD$. L1-DCache accesses decreased by 45% on average, resulting in a 44% geomean reduction in L1-DCache power. When executing embedded system applications on SPX64, the geomean power consumption of SD$ was only 7% of the L1-DCache power. The L1-DCache and SD$ collectively consumed 61% less power (geomean) than the baseline L1-DCache. Moreover, the improved ability to exploit locality reduced total L2-Cache accesses by 91% on average and reduced L2-Cache power consumption by 64%. We expect additional power savings if the SD$ were power-gated when not in use.

8.3 Area Overhead

In addition to requiring space for the SD$ itself, SPX64 requires changes to the LSQ (SPX64-LSQ) for processing both SPX64 and regular instructions; and changes to MSHRs, which must be shared between the L1-DCache and the SD$. As shown in Table 10, this leads to a 1.75% area increase for

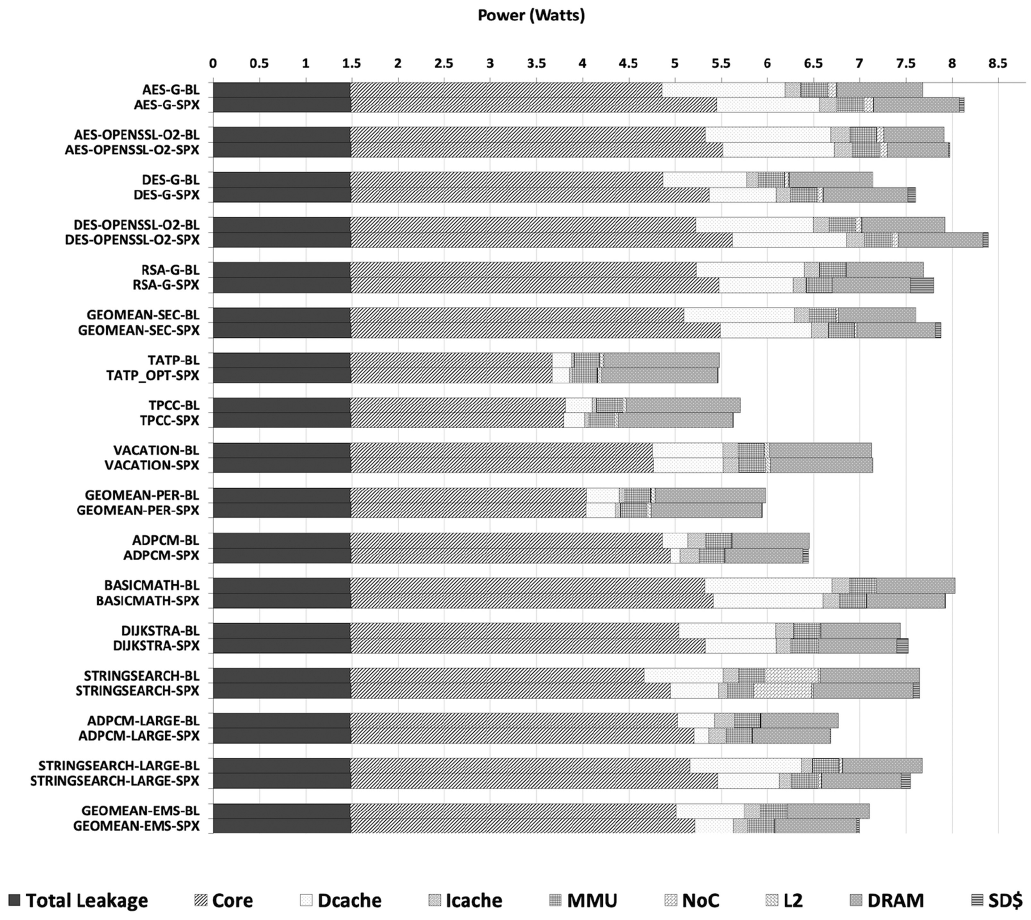

Fig. 7. Power breakdown of SPX64 architecture (with 2-cycle SD$ access latency) for security, persistent, and embedded systems benchmarks.

Table 10. Area Breakdown of SPX64 Architectures

Processor Configuration	**L1-I$**	**L1-D$**	**MMU**	**L2**	**Core**	**NoC**	**SD$**	**L3**
Single core, 2-level caches	1.75%	21.27%	0.56%	27.79%	45.35%	1.49%	1.75%	-
Six cores, 3-level caches	1.47%	17.93%	0.47%	23.42%	38.22%	1.26%	1.47%	15.71%

The first row shows the evaluation for the configurations in Table 5 and Table 6. The second row shows the area breakdown of a system featuring six cores and a 9 MB 16-way set-associative L3 cache, similar to an Intel Skylake [21] processor.

a single-core system with a 2-level cache hierarchy. The SD$ contributes to most of the area overhead. However, the area of SD$ is much smaller than the L1-DCache area, as L1-DCache requires 3 ports in the baseline to satisfy the LSQ issue width and coherence requests, while the proposed SD$ requires a single port. This is because SPX64-LSQ has the same issue width as the baseline LSQ and the SD$ does not need to support coherence. Note that the absolute area overhead will increase with an increase in the number of cores. However, for higher core counts, a system typically is configured with a larger last-level cache and a deeper cache hierarchy. Thus, the relative area overhead is smaller (1.47%) in such a system.

9 RELATED WORK

A traditional scratchpad memory [5, 18, 34] is a software-managed on-chip data storage without any tag array, which provides an address space that is disjoint from and inconsistent with main memory. These properties can reduce energy, area, and latency [5] relative to a hardware-managed cache. While scratchpads offer lower access latency and provide total control to programmers, who manage the movement of data blocks in and out of the scratchpad, they may demand considerable programming effort: The programmer or compiler must determine the timeliness of initiating a data transfer via DMA so the memory access latency can be hidden [18, 24] and computation does not stall. Applications can be accelerated by managing coarse-grained and high-locality data blocks in scratchpad through predictable, regular, and low latency accesses.

Hardware-managed caches require less programming effort to hide memory access latency, exploit locality without explicit DMA transfers, and are more amenable to general-purpose executions including irregular or dynamic data structures. To take advantage of both the cache and scratchpad, virtual local store [11] and hybrid cache [10] partition existing L1 caches to allocate some blocks as a software-managed scratchpad, with others managed by hardware. Allowing software to decide which data to keep in the faster-to-access memory can reduce conflict misses. Virtual local store [11] allocates a reserved space in global memory to back-up values in the scratchpad region, which allows easy context switches. Hybrid cache [10] also reserves space in the global memory and flexibly allocates blocks in underutilized cache sets to the scratchpad. Unlike the hybrid cache or virtual local store, SPX64 does not partition an existing L1 cache. Instead, a dedicated hardware storage is used, which is not disjoint from the global space, and simplifies the hardware design while maintaining low access latency.

Stash [27] is another on-chip memory organization that combines features of cache (global address space) and scratchpad (direct addressability). Stash has a map between the global memory address space and the local "stash" address space, which enables global addressing and visibility. The design and implementation of SPX64 differ from stash, because SPX64 emphasizes isolation over global visibility. In contrast to Stash, DAWG [23] is specifically concerned with isolation. It resembles VLS and Hybrid Cache in that it partitions the L1 into an isolated region and a non-isolated region. This strategy can deliver many of the security benefits that SPX64 achieves, but it does not allow the programmer to exploit hardware associativity to accelerate log lookups.

10 CONCLUSIONS AND FUTURE WORK

In this article, we argued that adding a scratchpad memory to CPUs can bring many benefits, including increased security, the ability to avoid hash table lookup overheads for persistent transactions, and higher performance at lower power for applications with high data locality. We presented a comprehensive design, called SPX64, which involved changes to the CPU's microarchitecture, as well as a handful of small extensions to the operating system. We then showed through simulation that SPX64 uses a modest amount of chip area and power and significantly increases the performance of our target workloads while increasing security.

The most significant feature of SPX64 is that it can accomplish all of these goals using a single new hardware structure that is orthogonal to the rest of the core, and hence easier to reason about. While individual solutions have the potential to provide better overall performance for one of these application domains, the SPX64 value proposition is greater, since it is beneficial to many classes of applications. In this article, we show three examples. For security applications, the proposed SPX64 can provide isolation to prevent cache-based side channels. Evaluation results show that different variants of the SPX64 can achieve flexible tradeoffs between performance and security. For persistent transactions, the proposed SPX64 can be used to accelerate log lookups by leveraging the virtual-addressing and set-associative features of the SD$. By changing log lookups to direct SD$ accesses, the total number of dynamic instructions can be reduced. For embedded workloads, the proposed SPX64 can be used to achieve performance improvements similar to other scratchpad designs without explicitly moving data into the scratchpad address space. This is because SD$ can be seen as a shadow address space. Up to 10% performance improvement is observed on the evaluated embedded systems applications. The SD$ is a simpler hardware cache as compared to the L1 data cache, and hence the access latency is relatively low and area overhead is small.

As future work, we plan to explore the value of SPX64 in areas as diverse as generational garbage collectors, real-time workloads, and secure hypervisors. We also intend to refine our implementation, particularly with regard to the integration of SPX64 support into compiler tool-chains and programming languages.

REFERENCES

[1] Intel Corporation. 2018. *White Paper: Retpoline: A Branch Target Injection Mitigation.* Technical Report 337131-003. Retrieved from https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf?source=techstories.org.

[2] Sam Ainsworth and Timothy M. Jones. 2019. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. *arxiv:cs.CR/1911.08384* (2019).

[3] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.*

[4] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Trans. Archit. Code Optim.* 14, 2 (June 2017). DOI: https://doi.org/10.1145/3085572

[5] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02).* 73–78. DOI: https://doi.org/10.1145/774789.774805

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.

[7] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, Louis Goubin and Mitsuru Matsui (Eds.). Lecture Notes in Computer Science, Vol. 4249. Springer Berlin, 201–215.

[8] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Not.* 49 (2014), 433–452.

[9] Jeremy Condit, Edmund Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles.*

[10] Jason Cong, Karthik Gururaj, Hui Huang, Chunyue Liu, Glenn Reinman, and Yi Zou. 2011. An energy-efficient adaptive hybrid cache. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design (ISLPED'11).* IEEE Press, Piscataway, NJ, 67–72. Retrieved from http://dl.acm.org/citation.cfm?id=2016802.2016825.

[11] Henry Cook, Krste Asanovic, and David A. Patterson. 2009. *Virtual Local Sstores: Enabling Software-managed Memory Hierarchies in Mainstream Computing Environments.* Technical Report. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-131 (2009).

[12] Luke Dalessandro, Michael Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming.*

[13] Ning Deng, Weixing Ji, Jaxin Li, and Qi Zuo. 2011. A semi-automatic scratchpad memory management framework for CMP. In *Proceedings of the International Workshop on Advanced Parallel Processing Technologies.* Springer, 73–87.

[14] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22nd USENIX Conference on Security.* 431–446.

[15] Subramanya Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems.*

[16] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias. 2004. An integrated hardware/software approach for runtime scratchpad management. In *Proceedings of the 41st Design Automation Conference*. ACM, 238–243.

[17] Christopher Garman, Xiaochen Guo, and Michael Spear. 2017. A study of unnecessary write backs. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'17)*. ACM, New York, NY, 127–129. DOI: https://doi.org/10.1145/3132402.3132438

[18] Michael Gschwind. 2007. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Prog.* 35, 3 (2007), 233–262.

[19] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games—Bringing access-based cache attacks on AES to practice. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'11)*. 490–505.

[20] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE International Workshop on Workload Characterization (WWC'01)*. IEEE, 3–14.

[21] Intel Inc. 2019. Intel Skylake. Retrieved from https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client).

[22] G. Irazoqui, T. Eisenbarth, and B. Sunar. 2015. S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*. 591–604.

[23] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. 974–987.

[24] Michael Kistler, Michael Perrone, and Fabrizio Petrini. 2006. Cell multiprocessor communication network: Built for speed. *IEEE Micro* 26, 3 (2006), 10–23.

[25] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'19)*. IEEE, 1–19.

[26] Paul C. Kocher. 1996. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In *Proceedings of the Advances in Cryptology Conference (CRYPTO'96)*.

[27] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have your scratchpad and cache it too. *SIGARCH Comput. Archit. News* 43, 3 (June 2015), 707–719. DOI: https://doi.org/10.1145/2872887.2750374

[28] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd IEEE/ACM International Symposium on Microarchitecture*. ACM, 469–480.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. 973–990.

[30] Fangfei Liu, Y. Yarom, Qian Ge, G. Heiser, and R. B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'15)*. 605–622.

[31] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[32] MICRON. 2020. DDR4 SDRAM. Retrieved from https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf.

[33] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*.

[34] NVIDIA. 2013.*Using Shared Memory in CUDA C/C++*. Retrieved from https://devblogs.nvidia.com/using-shared-memory-cuda-cc/.

[35] Dag A. Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: The case of AES. *Topics in Cryptology–CT-RSA 2006* (Jan. 2006). Springer, 1–20.

[36] Colin Percival. 2005. Cache missing for fun and profit. In *Proceedings of the BSDCan Conference*.

[37] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. 199–212.

[38] Muhammad Refaat Soliman and Rodolfo Pellizzoni. 2017. WCET-driven dynamic data scratchpad management with compiler-directed prefetching. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS'17)*

(Leibniz International Proceedings in Informatics (LIPIcs)), Marko Bertogna (Ed.), Vol. 76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 24:1–24:23. DOI: https://doi.org/10.4230/LIPIcs.ECRTS.2017.24

[39] Eran Tromer, DagArne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.* 23, 1 (2010), 37–71.

[40] Sumesh Udayakumaran and Rajeev Barua. 2003. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'03)*. Association for Computing Machinery, New York, NY, 276–286. DOI: https://doi.org/10.1145/951710.951747

[41] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems.*

[42] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. 2018. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. 428–441.

[43] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA nonces using the flush+reload cache side-channel attack. Cryptology ePrint Archive, Report 2014/140.

[44] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security*. 719–732.

[45] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A timing attack on OpenSSL constant time RSA. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES'16) (Lecture Notes in Computer Science)*, Benedikt Gierlichs and Axel Y. Poschmann (Eds.), Vol. 9813. Springer, 346–367. Retrieved from http://dblp.uni-trier.de/db/conf/ches/ches2016.html#YaromGH16.

[46] Richard Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin Lee. 2008. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures.*

[47] Pantea Zardoshti, Tingzhe Zhou, Pavithra Balaji, Michael L. Scott, and Michael Spear. 2019. Simplifying transactional memory support in C++. *ACM Trans. Archit. Code Optim.* 16, 3 (July 2019). DOI: https://doi.org/10.1145/3328796

[48] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the ACM Conference on Computer and Communications Security*. 305–316.

Received May 2020; revised November 2020; accepted November 2020