SSE: Security Service Engines to Scale Enclave Parallelism for System Interactive Applications

Jared Nye, Usman Ali, and Omer Khan
{jared.nye, usman.ali, khan}@uconn.edu
University of Connecticut, Storrs, CT, USA

2024 International Symposium on Secure and Private Execution Environment Design (SEED) | 979-8-3315-0565-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SEED61283.2024.00019

***Abstract*—Secure processor technologies leveraging enclaves as their architectural security primitive are frequently deployed in cloud environments. However, enclave-based systems incur performance penalties due to architectural limitations arising from costly enclave exits that are incurred to interact with system-level software. Exitless calling aims to improve enclave-based performance by spawning additional responder threads alongside the enclave threads to execute system calls on their behalf, obviating costly enclave exits. However, the responder threads in exitless must use self-governed timers to operate truly asynchronously to the enclave threads to uphold security isolation guarantees. These self-governed timers induce polling stalls that degrade performance when enclave and responder threads saturate the available cores in the system. This paper addresses the polling challenge in exitless by introducing Security Service Engines (SSE) to offload responder threads using dedicated on-chip or off-chip hardware resources. Evaluations show that for highly interactive workloads, SSE-equipped secure multicores achieve performance scaling at par with a baseline system that implements no security primitives.**

I. Introduction

Cloud computing offers a compelling alternative to costly on-site computing, but requires remote clients to rely on third-party cloud providers to process their code and data [1]. This reliance on untrusted third parties raises security concerns addressed in part by the deployment of secure processor technologies in cloud computing infrastructures [2]. Prevailing secure processor technologies such as Intel Software Guard Extensions (SGX) [2] and AMD Secure Encrypted Virtualization (SEV) [3], as well as upcoming technologies such as RISC-V Keystone [4] and Arm Confidential Compute Architecture (CCA) [5], are fundamentally similar in that they attempt to implement isolation property for software security by leveraging the *enclave* as their essential architectural security primitive. Enclaves provide isolated execution environments protected from co-resident user- and system-level software. They enable additional security features, such as physical memory confidentiality, integrity, authentication, and remote attestation. Consequently, enclaves are a staple of secure processor technologies, and processors incorporating enclave-based execution have remained prevalent across academia [6]–[9] and industry [10].

Enclaves offer enhanced security but introduce performance overheads intrinsic to their architectural implementations. For example, on each enclave memory access, memory encryption and integrity checking overheads are incurred to enforce the integrity and confidentiality of enclave memory. Since system-level software is not allowed to execute inside enclaves, any time an application requires system-level software service, an enclave exit must be performed that incurs an additional overhead due to core serialization, state purging, and security checks. To illustrate how frequently applications that are commonly used to characterize enclave performance [11]–[14] interact with system-level software, we measure their interactivity. An 8-core SGX-enabled Intel machine is used to measure the number of per-core interactions per second. SGX and SEV consider the operating system (OS) and virtual machine monitor (VMM) to be untrusted, and thus incur enclave exits upon each system call and hypercall, respectively. For a 4 server and 4 client thread configuration, each core incurred an average of 128,716 system calls and 56,024 hypercalls per second, with similar numbers reported for overall interactivity in [11]. This leads to frequent enclave exits for system calls in SGX and hypercalls in SEV, thus making enclave-based processing expensive.

To mitigate these costly enclave exits, an *exitless* calling is introduced that avoids enclave exits while upholding isolation guarantees between the untrusted system-level software and enclave [11], [12]. Exitless avoids enclave exits through the utilization of an asynchronous calling mechanism which spawns two types of threads: (i) *workers* that execute application code inside enclaves and (ii) *responders* that execute system calls (or hypercalls) on behalf of enclaves [11], [12]. However, the challenge for exitless calling is that when worker threads saturate the available cores in a system, the additional responder threads incur polling-induced stalls due to their asynchronous execution [11], [12], [15]. This performance limitation has been quantified extensively in literature [11], [12], [15], [16] and leads to limited adoption of enclave-based execution.

Prior works [17], [18] improve exitless by pinning worker-responder thread pairs on the same cores and introducing hardware support for lightweight context switching between them. However, they do not fundamentally address the security-centric polling limitations of exitless. In this paper, we make a key observation that if worker and responder threads execute on their dedicated hardware resources, then one can fundamentally address the polling problem of exitless. This observation can be realized using the inherent heterogeneity available in existing architectures, or through added offload hardware. We propose *Security Service Engines (SSE)* that offer each enclave core a dedicated hardware context to offload responder threads. In our execution model, each enclave core maps application

979-8-3315-0565-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SEED61283.2024.00019

worker threads, but at any given time the corresponding SSE engine serves to operate in parallel and asynchronously. The lightweight SSE engines provide dedicated hardware for the responder threads. Furthermore, when exitless calling is not beneficial, the programmable SSE engines can be utilized for other offload services.

SSE mitigates the performance bottlenecks of exitless execution for highly interactive applications that effectively utilize the available cores in a multicore processor. Thus, with responders executing exclusively on SSE engines, workers operate uninterrupted and exploit parallelism. Additionally, polling stalls and context switches are avoided, and costly memory overheads are no longer incurred. To demonstrate the performance benefits of SSE, we evaluate web server and database management workloads representative of applications commonly used to characterize enclave performance. The performance evaluation is done using the MIT Graphite multicore simulator updated with the RISC-V instruction set architecture [19], [20]. The simulator is modified with enclave system call and hypercall overheads, and our proposed SSE engines. In comparison to exit-based and exitless enclave execution models, SSE improves throughput when worker threads match the number of enclave cores in the system. Consequently, the dedicated SSE engines for corresponding responder threads deliver at par performance scaling with a baseline system that implements no security primitives.

II. Background and Motivation

A. Enclaves

The Intel Software Guard Extensions (SGX) [2] introduced enclaves as an architectural security primitive to address cloud computing security concerns. Since then, academia and industry have aggressively adopted enclaves to enhance cloud computing security services, as they provide (i) isolation of code/data at the hardware level, (ii) remote attestation and authentication of code/data for remote users, and (iii) encryption of code/data that is only decrypted while being used inside the enclave.

1) Enclave Threat Model: The enclave threat model protects against the entire software stack and an attacker with physical access to the system, such as a rogue cloud employee. Thus, the following components of the software stack are considered untrusted: firmware, VMM, host operating system, and other mutually untrusted applications. Protection against each of these threats is maintained via the isolation of enclave code and data at the hardware level and through the purging of the entire enclave state upon each context switch. Meanwhile, protection against a physical attacker is upheld through the confidentiality and integrity checking of enclave data.

2) Diversity of Enclave Implementations: Two classes of enclaves have been introduced, *user-level* and *VM-based*, which are differentiated by the scope of applications they are intended to protect. *User-level* enclaves, such as Intel SGX and RISC-V Keystone, consider the operating system to be untrusted and thus only encapsulate individual secure computations or a single user-level application. *VM-based* enclaves, such as AMD SEV and Arm CCA, consider the VMM to be untrusted

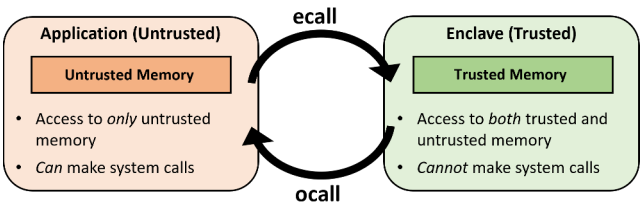


Fig. 1: Exit-based System Call Flow

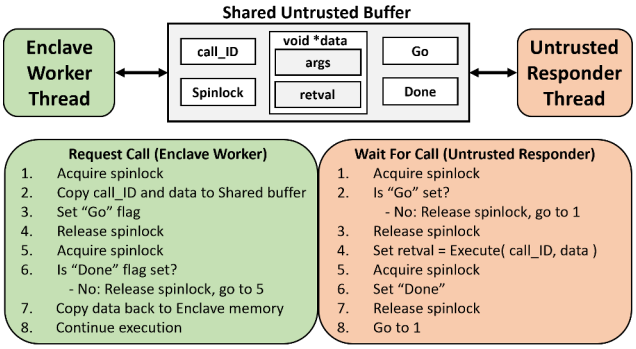


Fig. 2: Exitless Worker-Responder Architecture

and thus encapsulate entire virtual machine (VM) instances. VM-based enclaves trust the guest operating system, enabling code execution across multiple privilege levels. Both user-level and VM-based enclaves introduce performance overheads since both require expensive secure switches outside (and back inside) enclaves to enforce isolation from system-level software.

B. Exit-based Calling

The *exit-based* enclave model shown in Figure 1 implements special security procedures for the enclave code to access the system-level software. For an untrusted application to begin execution inside an enclave, an ecall function is used to make a secure context switch into trusted enclave code. Similarly, to return to untrusted code, an outside call function (ocall) is executed within an enclave to return control back to the untrusted application.

During an ecall, several hardware security checks are performed, the entire untrusted application state is backed up, the state of the previous enclave application is restored (if applicable), the core pipeline and translation look-aside buffers (TLBs) are flushed, and the processor core is switched out of enclave mode. During an ocall, the entire trusted enclave state is backed up, the state of the untrusted application is restored, the core pipeline and TLBs are flushed, and the processor core is switched out of enclave mode. If an ocall is invoked to utilize system-level services, all information located in enclave memory needed by system-level software must first be copied from enclave to untrusted memory. After the required services are complete an ecall is performed to resume execution inside an enclave. Consequently, ecalls and ocalls are costly and take 83–113x longer than typical system calls [11].

C. Exitless Calling

Prior works [11], [12] proposed an alternative *exitless* calling model that mitigates expensive enclave exits when system software services are required in the exit-based model without violating isolation guarantees between the enclave and untrusted responder. This is accomplished using an asynchronous worker-responder calling mechanism between worker threads (executing enclave code) that make system call requests, and responder threads (executing untrusted code) that process system call requests [11], [12]. Figure 2 illustrates the architecture of this asynchronous worker-responder calling mechanism, where the enclave code is the worker, and the untrusted code is the responder. The worker is the party who requests a call, while the responder stands by, waiting for a call to process by constantly polling a shared memory location. To uphold isolation guarantees, responders are not allowed to directly access enclave code/data. Therefore, exitless utilizes an asynchronous polling mechanism to create a communication channel between the worker and responder threads using a shared buffer located in untrusted memory that is synchronized using a spinlock.

When system software services are needed, workers send system call requests to responders by executing *RequestCall* which loads system call information into a shared buffer, and then indicates a request is pending by setting a *Go* shared flag. After issuing a request, the corresponding worker waits for its pending system call request to complete by monitoring the shared *Done* flag until it is set by the responder (indicating system call processing is complete). Once completed, the worker thread copies the result back into trusted memory and continues execution. Asynchronously, responders execute *WaitForCall*, which continuously polls the *Go* flag for pending requests. Each responder thread implements a self-governing polling method to periodically check, receive, and process pending system call requests. Once *Go* is set, responders process the corresponding request by invoking the requested system call with its provided arguments. After the system call is complete, responders indicate a request has been processed by setting the *Done* flag. Responders then continue polling until the next request arrives. For applications that frequently interact with system-level software, prior works [11], [12], [15] have shown exitless calling significantly improves performance by avoiding exit-based overheads.

III. Limitations of Exitless Calling

Exitless calling scales up to the system threading capabilities and avoids exit-based calling overheads. However, it incurs unnecessary polling and context-switching overheads induced by the responder threads when the total number of threads exceeds the scaling capabilities of a machine. Polling overheads are incurred due to the asynchronous interface between workers and responders and are greatest at increased parallelism. This is because responders continuously poll, even without requests to process. Polling consumes an entire hardware context without any meaningful work to perform until the responder's self-governed timer expires. This results in a performance bottleneck

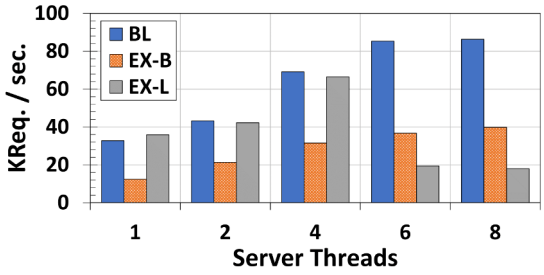


Fig. 3: Evaluation of Lighttpd with each configuration spawning an equal number of servers and responders (in exitless). Each server and responder occupy a single thread. As the system only has eight cores, in exitless it becomes overloaded in the 6-server configuration as twelve total threads are needed but cannot execute simultaneously.

when a responder thread is forced to share a core with other worker or responder threads. In such a scenario, responder polls without meaningful work to process and thus reduces the utilization of that core. Moreover, when there are not enough cores for all workers and responders to execute, the system scheduler is frequently invoked to swap workers for responders and vice versa, resulting in costly context switching overheads.

We evaluate the popular application lighttpd [21] to demonstrate the performance gap between exitless (EX-L), exit-based (EX-B), and an insecure baseline (BL). These measurements are performed on an 8-core SGX-enabled Intel Machine using the Graphene-SGX framework [13]. While the total number of servers, clients (which occupy only a single thread), and responders are less than or equal to the number of logical cores, the performance of exitless scales, as shown in Figure 3. However, once the total number of server, client, and responder threads exceeds the number of logical cores, the performance of exitless reduces significantly. This significant degradation in the performance of exitless is the result of two primary overheads: (i) polling and (ii) context switching.

Polling overheads are incurred due to the asynchronous interface between workers and responders and are most significant at increased parallelism because responders continuously poll, even when there are no requests to process. Continuous polling consumes an entire hardware context without any meaningful work until the next system call request arrives. While this is not an issue when hardware contexts are abundant, the negative effects of polling are glaring as the total number of workers and responders approach the number of available cores on a machine. In such cases, responders may poll without requests to process, stalling workers since there are no remaining hardware contexts for workers to execute.

When there are not enough cores for all workers and responders to execute, the OS scheduler must be frequently invoked to swap workers for responders and vice versa. Such frequent swapping of threads results in costly context-switching overheads. Frequent context switches further introduce cache thrashing, as the memory locality utilized by a thread executing uninterrupted may no longer be exploitable since another thread accesses its unique data structures using the same private caches

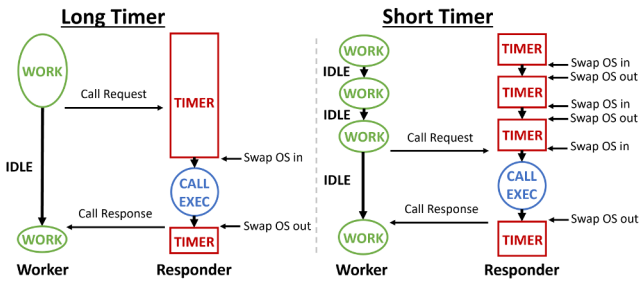


Fig. 4: System Calling in Exitless

as the thread it was swapped. This causes increased memory access latency and reduces performance.

Prior works [11], [13], [17], [18] have introduced thread scheduling and sleeping mechanisms to reduce the time responders poll when a few call requests are made. More specifically, [11], [13] reduce responder poll time by having responders set a timer and sleep until it expires. Communication between workers and responders must be truly asynchronous to uphold the isolation property, responders must operate on their autonomous timers without requiring the workers to invoke them. This requirement makes setting the timer length difficult as responders must sleep often and long enough to reduce polling and context switches, but not so much that wake-up penalties or unnecessary context switching occur, as shown in Figure 4.

We consider cases where workers and responders share a single core when the number of threads exceeds the number of cores, as this is the best-performing configuration in such a situation. The first scenario, where responders sleep too long, is denoted *Long Timer*. In Long Timer, a wake-up penalty is incurred after a period where no system calling occurred since the responder went to sleep and is still asleep after a worker finally makes a request. As a result, the worker continues to poll until the responder wakes up and is swapped in. This scenario is not ideal as system call processing does not begin until the timer set by the responder expires, increasing the latency of system calls.

The second scenario, where responders sleep too often is denoted *Short Timer*. In Short Timer, significant context-switching penalties are incurred as responders are invoked unnecessarily. Figure 4 demonstrates this, as after sleeping for a short period, the responder's timer expires, and it is swapped in without any system calls to process. Even in the best-case scenario where the responder sleep time is configured to wake up as soon as system call requests arrive, obviating any worker polling, such periodic interruptions increase overall latency as workers are frequently interrupted by unnecessary responder invocations.

Thus, while exitless calling allows for improved performance of highly interactive applications compared to exit-based calling under certain conditions, it suffers from significant performance challenges when polling becomes the bottleneck and can also be worse than exit-based. This demonstrates that overheads intrinsic to exitless calling can be more costly than the overheads incurred on each enclave exit incurred in exit-based and is not by itself an appropriate bottleneck mitigation scheme in the presence of increased parallelism.

IV. Security Service Engines

To take advantage of multicore parallelism, we propose *Security Service Engines (SSE)*, a technique that overcomes the polling challenge with exitless by coupling each general-purpose enclave core in a shared-memory multicore processor with a lightweight *SSE engine*. SSE engines can be integrated using existing cores which we denote as *SSE-Software* or one of two ways using dedicated hardware: either (i) on the CPU, within the tile of its coupled enclave core, which we denote *SSE-ONCPU*, or (ii) off the CPU using available off-chip resources, such as an FPGA, which we denote *SSE-OFFCPU*. In all configurations, SSE engines are fully programmable, have their own dedicated private L1 instruction and data caches, and access the shared cache hierarchy. SSE engines are designed to perform two tasks: first, the execution of responders spawned by an exitless enclave application and offloaded to an SSE; second, serve asynchronous system call requests that are issued by workers. The worker-responder architecture and API used in SSE are similar to that of exitless but with a key difference. The workers execute exclusively on enclave cores, while responders are offloaded to SSE engines to avoid timer limitations and consequently performance degradation.

A. SSE-Software

To operate in parallel and asynchronously, SSE offers each enclave core a dedicated context to offload syscalls execution on the responder core. SSE-Software is a software-only approach and requires no additional hardware. In SSE-Software, all available cores are spatially distributed between worker and responder threads in *1-worker-1-responder* or *n-worker-1-responder* configurations depending on the interactivity of workloads. A shared memory buffer allows inter-process communication (IPC) between the worker and the responder. For non-interactive workloads, all available cores are assigned to workers to maximize parallelism and performance. For low-interactivity workloads, a small amount of responder cores is sufficient, whereas, for high-interactivity applications, a 1-1 configuration is optimal to maximize parallelism and performance.

Although SSE-Software has the inherent benefit of no hardware overhead and provides asynchronous syscalls processing, it has two major limitations. 1) For n-worker and 1-responder configuration, the responder core serializes syscalls processing for multiple workers which results in additional performance penalties. 2) For 1-worker and 1-responder configurations, fewer cores are available for worker threads, which results in less parallelism, leading to performance degradation.

B. SSE-ONCPU

To overcome SSE-Software limitations, SSE-ONCPU proposes a lightweight dedicated core to implement SSE. Figure

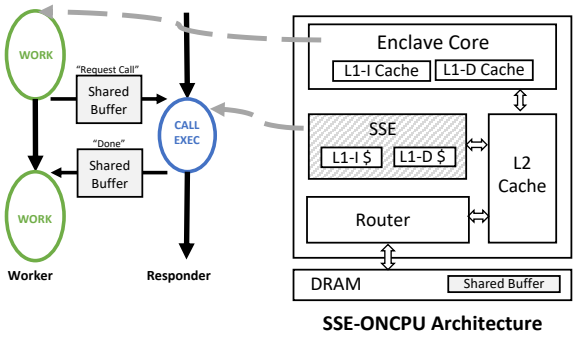


Fig. 5: SSE-ONCPU architecture with SSE engines, enclave core, cache hierarchy, and untrusted shared buffer.

5 demonstrates how each tile is architected using the SSE-ONCPU approach. It also shows how threads interact and execute with SSE engines. In SSE-ONCPU, an application may only spawn as many responder threads as there are SSE engines on a machine to avoid unwanted polling bottlenecks. Each SSE engine operates such that its responder thread strictly serves the worker thread(s) executing on its corresponding enclave core. Thus, while an application may spawn more worker threads than there are enclave cores, any additional performance penalty incurred under SSE because of over-saturating enclave cores will also be incurred by a baseline system without SSE engines. By providing workers and responders distinct execution resources, unnecessary stalls, and context switching are obviated as enough hardware contexts are provided for all workers and responders to execute simultaneously.

1) Lightweight Implementation: As offload hardware is typically responsible for the execution of a set of specified tasks, there are opportunities to make optimizations to improve performance or reduce area and/or power. For example, the work performed by responder threads, i.e., checking the shared buffer and executing system calls, is not compute-intensive. Therefore, each SSE engine utilizes a simple in-order shallow pipeline with no out-of-order execution support. Additionally, during a system call an SSE engine only performs a one-time copy of data located in untrusted memory. With little data reuse, SSE implements small and simple private caches. Furthermore, due to the layer of indirection added when workers and responders interact, shared data must be marshaled back and forth between private caches of enclave cores and SSE engines. For simplicity, SSE engines are designed not to cache much of the system call data, but instead evict it to the shared L2 cache. Consequently, the enclave core can move the requested data back into its enclave memory without needing to invalidate its SSE engine's private cache, thus avoiding costly sharing misses. These implementation choices result in cost-effective caches in SSE without reducing performance.

C. SSE-OFFCPU

Heterogeneous architectures featuring specialized hardware accelerators are increasingly being deployed by leading CPU vendors. Among various heterogeneous devices, FPGA fabric

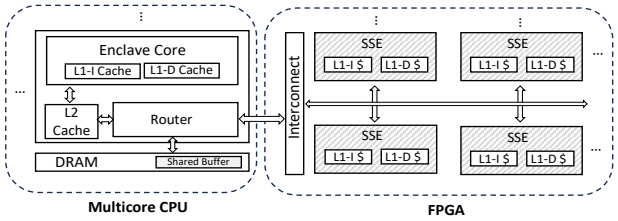


Fig. 6: SSE-OFFCPU architecture with SSE engines located off the CPU of a CPU-FPGA platform.

is actively being integrated within Intel CPUs using shared memory QPI cache interface [22]. As such, we also consider an implementation of SSE that leverages an existing heterogeneous architecture with an FPGA located off the CPU, which we denote *SSE-OFFCPU*. SSE engines can be mapped to the FPGA either as (i) synthesizable logic blocks or (ii) burned as programmable RISC-V cores, with our implementation performing the latter to take the more programmable approach. Figure 6 demonstrates how SSE engines can be implemented using an FPGA, with workers and responders executing exclusively on the enclave cores (located on the CPU) and SSE engines (located on the FPGA), respectively.

1) Existing CPU-FPGA Platforms: We develop SSE-OFFCPU considering recent CPU-FPGA platforms, such as Intel's Xeon+FPGA [23], which co-packages the CPU and FPGA to deliver higher memory bandwidth and lower memory latency for code that executes on the FPGA. In particular, the FPGA has access to the shared memory of the CPU which allows it to make memory accesses via the CPU. Communication between the CPU and FPGA is supported by both PCIe and Quick Path Interconnect (QPI) physical links. The PCIe interconnect is used for reads and writes from the FPGA memory directly from and to DRAM, respectively, while the QPI interconnect maintains data coherency between the last-level cache of the CPU and the memory located on the FPGA. Using both types of links allows for the CPU and FPGA to share a unified address space between the CPU and FPGA which obviates costly data replication.

2) Code Execution Off-Chip: The untrusted and trusted buffer still exists in the CPU space since enclave cores are considered trusted and thus workers execute exclusively on enclave cores. Each enclave core has its private L1 caches and shared L2 slice, hence both enclave and insecure code are stored in each enclave core's private caches and shared L2 cache slice. Meanwhile, since responders are considered untrusted, they execute exclusively on the SSE engines located on the FPGA and can only access untrusted data through the CPU interface without requiring further modifications to the memory hierarchy. The only code executed on the SSE engines in this implementation is the responder code which only consists of polling a shared flag and the capability to execute system calls. Thus, data transfers between enclave cores and SSE engines are leveraged through the last-level cache.

D. Discussion of SSE Tradeoffs

Overall, an SSE-enhanced secure multicore processor provides a novel method to address the polling bottleneck in exitless calling. The responders operate truly asynchronous to the workers, and SSE can avoid the performance challenges associated with the security-centric polling to service system calls. Consequently, performance gains are achieved by avoiding unwanted thread switches and reducing system call waiting times for workers.

1) Generality of SSE: Similar to SSE-Software, multiple enclave cores may share a single SSE engine to reduce hardware overhead. However, for applications where each worker frequently interacts with system-level software, a one-to-one coupling of enclave cores and SSE engines is necessary to achieve optimal performance. For less interactive applications, a one-to-one coupling of enclave cores and SSE engines is excessive and fewer SSE engines suffice. To optimize resource utilization under such scenarios, the programmable SSE engines can be reused to perform other useful services that have been well-studied in literature, such as worklist-directed prefetching of tasks in graph acceleration [24], cache optimizations [25], and many others [26]–[28]. Compared to SSE-ONCPU, SSE engines in SSE-OFFCPU have the added benefit of being reconfigurable since they are implemented within the FPGA fabric of a CPU-FPGA platform and thus allow for further hardware optimizations to improve the performance of less interactive applications.

2) Benefits of SSE-ONCPU over SSE-OFFCPU: As offload engines typically execute a set of specified tasks, there are opportunities to make optimizations to the engines to further improve performance or reduce area and/or power. The tasks executed on SSE engines are summarized as polling, synchronization, and system call processing. Given the compute-bound and low arithmetic-intensive nature of each of these tasks, SSEs are implemented using simpler in-order cores, and only need a single hardware context per core to execute a single responder. This allows applications to execute in trusted enclaves that are more arithmetic intensive and require the full capability of an optimized superscalar core to maximize performance to execute exclusively on such cores. Meanwhile, responders, which do not require such capabilities, instead occupy SSE engines and ensure optimal utilization of system resources.

Compared to SSE-OFFCPU, the proximity of enclave cores and SSE engines in the SSE-ONCPU configuration allows for more efficient data communication and consequently shorter L2 cache sharing and serialization latencies. Additionally, as SSE-ONCPU places the SSE engines on-chip, SSE engines operate at a higher frequency than they could if they were implemented on an FPGA fabric as is done in SSE-OFFCPU.

3) Benefits of SSE-OFFCPU over SSE-ONCPU: In addition to being reconfigurable compared to SSE-ONCPU, SSE-OFFCPU has the added benefit of not requiring hardware modifications. This allows users to better exploit the off-CPU hardware for applications that are less interactive and hence do not benefit as much from deploying many responder threads. However, SSE-OFFCPU comes with its tradeoffs. Most notably, SSE engines in SSE-OFFCPU operate at a lower frequency and incur a longer shared memory access latency since the shared L2 slices are all located on the CPU. This prevents SSE-OFFCPU from completely matching the performance of SSE-ONCPU.

Algorithm 1 Server

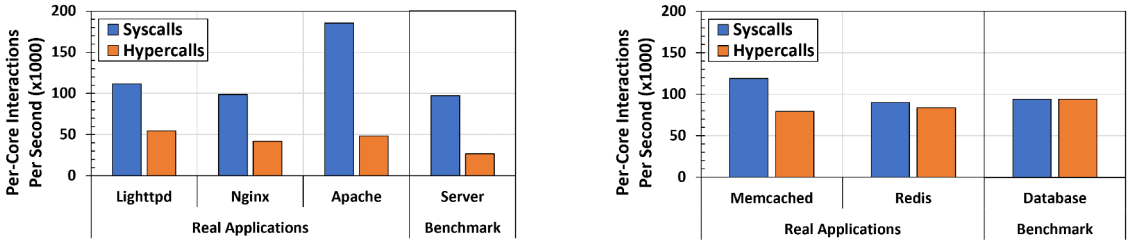
```
1:  procedure WORKER
2:      conn_fd ← ocall_accept(socket_fd)
3:      for i ← 1 to num_requests do
4:          req_size ← ocall_ioctl(conn_fd)              ▷ Request size
5:          rd_queue ← malloc(req_size)                  ▷ Read buffer
6:          ocall_read(conn_fd, rd_queue, req_size)
7:          req ← parse_req(rd_queue)                    ▷ Parse request
8:          file_fd ← ocall_open(req.name)               ▷ Open web page
9:          info ← ocall_fstat(file_fd)                  ▷ Web page info
10:         resp ← gen_resp(info)                        ▷ Generate response
11:         ocall_send(conn_fd, resp)                    ▷ Send response
12:         ocall_fstat(file_fd)                         ▷ Check for changes
13:         ocall_lseek(file_fd, 0)                      ▷ Move file offset
14:         wr_queue ← malloc(info.size)                 ▷ Send buffer
15:         ocall_read(file_fd, wr_queue, info.size)
16:         ocall_send(conn_fd, wr_queue, info.size)
17:         ocall_close(file_fd)                         ▷ Close web page
```

V. Workloads

The proposed SSE architecture is simulated on an application-level RISC-V simulator. To understand the performance implications, highly interactive applications are surveyed including Lighttpd [21], Nginx [29], Apache [30], Memcached [31], and Redis [32]. Each application makes hundreds of thousands of syscalls per second and executes Linux kernel functions (i.e., Linux kernel API [33]) to perform operating system-level operations. It includes *recv*, *send*, *read*, *socjet*, and other syscalls that perform I/O, network, and memory operations. Highly interactive applications are broadly categorized into Server (Lighttpd, Nginx, Apache) and Database (Memcached, Redis) applications. To evaluate the SSE architecture, the Server and Database benchmark applications are developed that implement the steady-state behavior of real applications. To implement syscalls at the application level, wrapper functions are implemented, and code from the Linux kernel is re-used [33] to implement operating system functionality.

A. Server Benchmark

Server is a highly interactive, lightweight web server application. It moves a large amount of data (up to 16 KB) through the *send* and *read* system calls between clients and servers. It moves smaller amounts of data (less than 200 bytes) through all other system calls. Each of these system calls stresses worker-responder interactions in exitless calling. The high-level pseudocode of the Server is shown in Algorithm 1. Server spawns *N* workers that serve *M/N* web page requests from clients that are divided evenly between workers. *M* is the total number of web page requests issued. After accepting a connection request, workers iterate over a loop that begins with *ioctl* syscall to get the request size, then allocate a buffer of that size. The request is read using *recv* syscall, parsed to



(a) Server (b) Database

Fig. 7: Number of system-level software interactions (OS for syscalls and VMM for hypercalls) each core incurs per second.

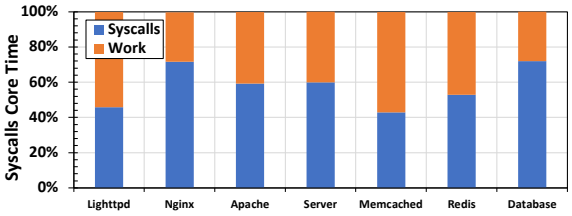


Fig. 8: Per core time utilization to execute syscalls in the exit-based system for real applications and benchmark applications.

Algorithm 2 Database

```
1:  procedure WORKER
2:      buf ← malloc(BUF_SIZE)                          ▷ Request buffer
3:      for i ← 1 to num_requests do
4:          ocall_read(fd, buf, BUF_SIZE)               ▷ Read request
5:          cmd ← read_cmd(buf)                         ▷ Get command
6:          key, data ← cmplt_read(cmd, buf)            ▷ Parse key/data
7:          idx ← key%HASH_TABLE_SIZE                   ▷ Table index
8:          spin_lock_acquire(key_locks[idx])           ▷ Acquire lock
9:          if cmd == set then
10:             hash_table[idx] ← data                  ▷ Update entry
11:             resp ← add_set_header()                 ▷ Get header
12:             ocall_send(fd, resp)                    ▷ Send response
13:         else
14:             res ← hash_table[idx]                   ▷ Response data
15:             resp ← add_get_header(res)              ▷ Get header
16:             resp ← add_data(resp, data)             ▷ Get data
17:             ocall_send(fd, resp)                    ▷ Send response
18:         spin_lock_release(key_locks[idx])           ▷ Release lock
```

determine the requested file's name, then the requested file is opened using *open* call. *Fstat* is then called to get the requested file stats needed that generate the response sent to the client using a *send* call. *Fstat* is called again to ensure the file has not been modified, then *lseek* is called to move the file offset to the beginning of the file. A buffer is allocated to store the file contents that are read and then sent using *read* and *send* calls, respectively. Finally, the worker closes the file using *close* call, then repeats this process for the next request.

B. Database Benchmark

Database is a highly interactive key-value RAM database. It moves a moderate amount of data (2 KB) through all system calls, stressing worker-responder interactions in exitless calling. The high-level pseudocode of the Database is shown in Algorithm 2. Database spawns N workers that serve M/N queries that are divided evenly between workers, where M is the total number of queries issued. Workers iterate over a loop that begins with a *read* call to read a query and then parses the query. After determining the command type (get or set), the lock for the entry that is being accepted is acquired. If the command is a get, then the data in the entry is sent back to the client using a *send* call. If the command is a set, then the entry is updated with the data received from the client, and a response acknowledging the update is sent back to the client using a *send* call. Finally, the worker releases the entry's lock.

C. Benchmark Validation

Two sensitivity studies are conducted to validate that benchmark applications align with real workloads for SSE architecture performance characterization. Figure 7 shows the per-core interactivity of real applications and *Server* and *Database* benchmarks on an 8-core SGX-enabled Intel machine. Figure 7 highlights that benchmark applications perform hundreds of thousands of syscalls similar to real applications. To validate time distribution to process syscalls and work in benchmark and real applications, the time spent using *strace* utility is measured on the SGX machine. Further, Figure 8 shows per core time utilization of syscalls in real applications and the benchmark applications. The results validate that the benchmark applications show a similar time distribution compared to real applications.

VI. Methodology

The baseline (BL), exit-based (EX-B), exitless (EX-L), and our proposed SSE-enhanced architectures (SSE-ONCPU and SSE-OFFCPU) are implemented using the MIT Graphite simulator enhanced with the needed performance models, and RISC-V ISA [19], [20]. A 64-core tiled multicore processor with a two-level coherent private L1 shared L2 cache hierarchy per core, and a 2D mesh on-chip network with X-Y routing is evaluated. The default architectural parameters used for evaluation are shown in Table I. In BL and EX-B, workers are spawned from 4–64 (depending on the configured number of workers) on the corresponding core number. The spawning of workers for EX-L, SSE-ONCPU, and SSE-OFFCPU is more complex and thus is discussed in a later subsection.

CPU Subsystem	
Number of Cores	64 RISC-V, Out-of-Order @ 1 GHz
Reorder Buffer Entries	192
Store Queue Entries	32
Memory Subsystem	
L1–I, L1–D Cache per core	32 KB, 4–way Assoc., 1 cycle
L2 Inclusive Cache per core	256 KB, 8–way Assoc., 4 cycles
Cache Line Size	64 bytes
Directory Protocol	Invalid–based MESI, ACKwise$_4$
DRAM Controllers	4, 10 GBps per Contr./ 100ns
Encryption Overhead	10 cycles
Electrical 2–D Mesh with XY Routing	
Hop Latency	2 cycles (1–router, 1–link)
Contention Model	Only link contention, 64 bit Flits (Infinite input buffers)
Flit Width	64 bits

TABLE I: Architectural Simulator Parameters for Evaluation.

A. Enclave Modeling

To model memory encryption and integrity checking performance overheads, a constant 10-cycle latency [34] is added to each main memory data access performed across each implementation excluding BL.

1) Exit-based (EX-B): Tian et. al [17] quantify the overhead of each ecall/ocall in Intel SGX to be 9,000-cycles, respectively. Thus, a 9,000-cycle latency is added to each ecall/ocall performed in EX-B.

2) Exitless (EX-L): When the number of threads exceeds the core count (at 64 workers), each worker-responder pair is pinned to a single core as this yields the highest performance under such conditions. Worker and responder threads are swapped out for one another based on a timer set by the responder. Initially, responders check for requests, and if there are none, set a timer and sleep until it expires. After a responder goes to sleep, a worker is swapped in and executes until the timer set by a responder expires.

The fixed cost of a context switch is set to a conservative 500ns based on measurements performed in [35]. To maximize performance when the number of threads exceeds the core count, responder sleep times are configured to 10μs which was experimentally determined to achieve optimal performance for the evaluated workloads.

3) SSE-Software: SSE-Software is modeled by pinning worker and responder threads on dedicated cores. All available cores are spatially distributed in n-clusters where each cluster contains 1 responder core while the remaining cores act as worker cores. This configuration is adopted to avoid additional network-on-chip latencies for worker-responder communications. A shared memory that is accessible within the cluster is used for worker-responder communication similar to EX-L.

4) SSE-ONCPU: SSE-ONCPU is modeled by adding an in-order simple pipeline and its private 1KB L1 instruction and data caches to each of the 64 tiles. Each SSE engine placed in a tile shares the L2 cache slice with its corresponding enclave core. Two variants of SSE-ONCPU are evaluated: (i) *SSE-ONCPU-NHP* which has a reduced 224KB L2 slice on each tile to compensate for the SSE-ONCPU area overhead and ensure resources comparable to BL, EX-B, and EX-L are used, and (ii) *SSE-ONCPU-HP* that has a full 256KB L2 slice on each tile. Threads are spawned such that workers are mapped exclusively to enclave cores, while responders are mapped exclusively to SSE engines.

5) SSE-OFFCPU: SSE-OFFCPU is also implemented on the simulator as opposed to an FPGA for a consistent evaluation and comparison with all other configurations. SSE-OFFCPU is modeled by adding 64 simple tiles, each with an in-order core that operates at a clock frequency of 250MHz and has its private 1KB L1 instruction and data caches that are coherent with the last-level cache of the CPU. To emulate existing CPU-FPGA platforms [22], no L2 cache slice is included in any of the FPGA tiles, and instead are only located on the CPU tiles. The added 64 FPGA tiles are positioned beneath the 64 CPU tiles and the location of the memory controllers remains around the original 64 CPU tiles, which causes an increase in shared and main memory access times for the SSE engines compared to the SSE-ONCPU. Like SSE-ONCPU, threads are spawned such that workers are mapped exclusively to enclave cores, while responders are mapped exclusively to SSE engines.

B. Evaluation Metrics

1) Throughput: Performance is measured in throughput specific to the operations the benchmark performs. Further details about how throughput is measured are described in the next subsection.

2) Execution Time Breakdown: The performance of each implementation is measured by tracking the average execution time across all workers. To gain further insights, execution time measurements are also tracked as follows:

- **Work** is the time spent executing workload-specific tasks that are not system calls or hypercalls.
- **Pack** is the time spent copying data to/from enclave memory from/to untrusted memory.
- **Responder-Syscall** is the time spent processing system calls or hypercalls by SSE cores.
- **Responder-Communication** is the time spent copying data to/from SSE memory from/to untrusted shared buffer.
- **Poll** is the time (i) workers wait for a responder's timer to expire after making a request and (ii) responders spend checking the shared buffer for requests.
- **Ctx Switch** is the time spent executing a context switch.

C. Workload Configurations

1) Server: Server is evaluated under two configurations: (i) an *SGX-like configuration* that incurs an enclave exit upon each system call mentioned in the previous paragraph, and (ii) an *SEV-like configuration* that only incurs enclave exits upon *send* and *recv* system calls as these communicate directly with devices that are managed by the VMM. 1,000 requests for a 16KB web pages are issued evenly between clients, with each worker serving its respective client's requests. Throughput is measured in thousands of requests served per second (KReq) and all evaluations are performed with 4 to 64 worker threads.

2) Database: Database is evaluated under one configuration that incurs enclave exits upon *send* and *read* system calls as these communicate directly with devices that are managed by

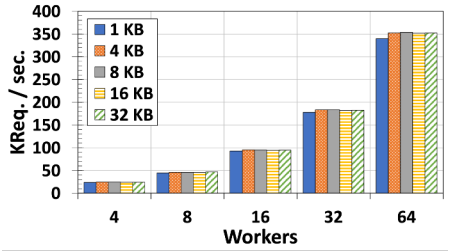


Fig. 9: L1 Instruction and Data Cache Size

the VMM (and thus the host OS). 4,000 requests for a 2*KB* query are issued evenly between clients, with each worker serving its respective client's requests. A get-to-set ratio of 1:1 is used. Throughput is measured in millions of queries served per second (MOPs) and all evaluations are performed with 4 to 64 worker threads.

VII. Evaluation

This section first outlines the evaluation used to determine the SSE engine microarchitecture tradeoffs. Next, the performance comparisons among the different configurations, BL, EX-B, EX-L, SSE ONCPU Hardware Penalty (SSE-ONCPU-HP), SSE ONCPU No Hardware Penalty (SSE-ONCPU-NHP), SSE-OFFCPU, and SSE-Software with 1-1, 3-1, and 7-1 settings are discussed.

A. SSE Microarchitecture Sizing Studies

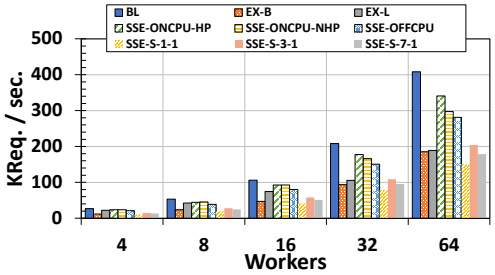
Figure 9 shows the performance of SSE with varying L1 instruction and data cache sizes. With private cache sizes ranging from 1–32*KB*, an overall difference in throughput of 2.7% is observed, indicating that SSE engines do not require large private caches to achieve optimal performance due to a significant number of sharing misses that are incurred upon each worker-responder interaction. Therefore, L1 instruction and data cache sizes of just 1*KB* are used to minimize SSE engine area and power overheads.

To determine out-of-order/in-order setting for SSE, the size of reorder buffer (ROB) is varied from 1–192 (data not shown). An overall difference in throughput of just 0.7% is observed, indicating that SSE engines do not require an out-of-order core to obtain optimal performance.

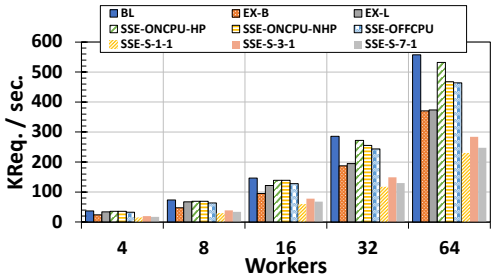
The store queue in SSE is also varied from 1–32 (data not shown). An overall difference of 4% in throughput is observed due to a significant number of writes that are performed during each worker-responder interaction. A single-entry store queue causes stalls that delay the worker from resuming execution. However, the SSE engines perform 0.5% better with only 8 store queue entries instead of 32 (default on the enclave cores).

B. Performance Evaluation of Server Benchmark

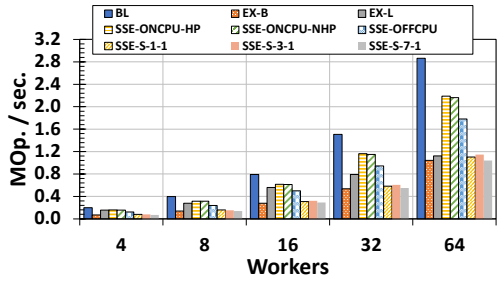
Figure 10 shows that baseline (BL) achieves the highest throughput across both SGX and SEV configurations for the Server benchmark as the number of workers is increased from 4 to 64. EX-B achieves performance scaling under both configurations but consistently performs worse due to costly



(a) Server: SGX-like Configuration



(b) Server: SEV-like Configuration



(c) Database: SGX-like Configuration

Fig. 10: Worker Throughput Performance

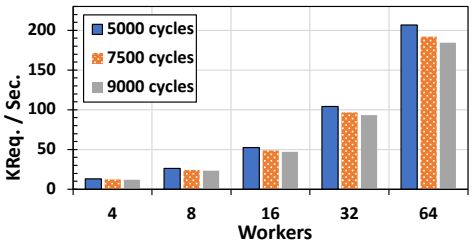
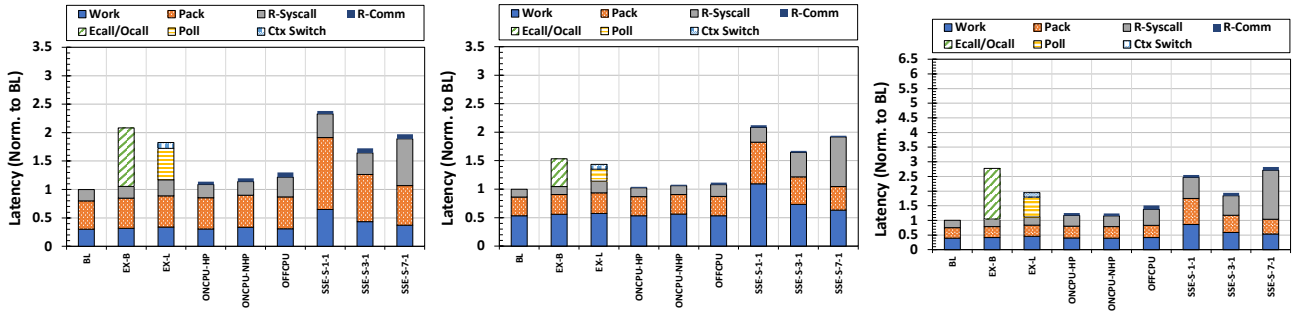


Fig. 11: SGX Exit-Based ecall/ocall Latency Overhead Sweep

ecalls/ocalls. The ecall/ocall overheads lead to a 2.3× and 1.5× performance loss for the SGX-like and SEV-like configurations of Server, respectively. Figure 11 shows the varying ecall/ocall overheads sweep study for EX-B. EX-L achieves performance scaling relatively to EX-B but suffers a decrease in performance at 32 and 64 workers due to polling and context switching overheads.

SSE-Software overcomes ecall/ocall overheads but is limited due to fewer worker cores. Furthermore, syscalls serialization penalties lead to diminished performance for all settings



(a) Server: SGX-like Configuration (b) Server: SEV-like Configuration (c) Database: SGX-like Configuration

Fig. 12: Worker Latency Breakdown (Normalized to BL)

compared to EX-B and EX-L. Hardware implementations of SSE continue to scale at 32 and 64 workers in both configurations. This is because the additional SSE engines enable SSE to mitigate polling and context-switching overheads. Since the focus is on improving EX-L performance when the number of threads exceeds the number of cores, Figure 12 shows the normalized breakdown of 64-worker configurations.

At 64 workers, EX-L performance is 1.9× and 1.5× worse than BL for the SGX-like and SEV-like configurations of the Server benchmark, respectively. This loss in performance is due to the frequent swapping of workers and responders as there are more threads than cores. Frequent context switching leads to (i) cache thrashing since data lost during a context switch must be re-fetched, and (ii) increased polling time as responders check for system call requests on each invocation. Figure 12 shows polling overheads are the primary cause of performance degradation in EX-L. Even with an ideal 0-cycle context switching overhead, EX-L would still trail BL performance by 1.7× and 1.4× for Server SGX-like and SEV-like setups. This demonstrates that any EX-L configuration that relies on the security-centric timers fails to match BL performance.

L1D cache thrashing is exacerbated by polling in EX-L compared to BL. In Server SGX-like configuration, an increase in L1D and L2 cache misses of 1.4× and 3.7× are incurred compared to BL. This leads to a 12.8% and 41.1% increase in workload and syscall components of worker execution latency, respectively. For Server SEV-like configuration, cache thrashing leads to an increase in L1D and L2 cache misses of 2× and 3× compared to BL. This causes a 7.8% and 49.3% increase in workload and syscall execution times.

SSE-Software eliminates ecall/ocall and polling overheads due to the spatial distribution of enclave and responder cores. For 1-1 configuration, Figure 10 shows that SSE-Software performs worse compared to EX-B and EX-L. This is due to the loss of 50% available worker cores compared to EX-B and EX-L. Figures 12a and 12b show an increase in latencies due to the loss of worker cores. For 3-1 configuration, SSE-Software outperforms EX-B and EX-L for SGX-like server workload due to the availability of additional worker cores but performs worst for SEV-Like workload due to less utilization of responder cores. For the 7-1 configuration, Figure 12a shows that the syscalls serialization penalty outperforms the availability of additional worker cores and SSE-Software performance degrades.

1) SSE-ONCPU Analysis: SSE-ONCPU eliminates ecall/ocall and polling overheads incurred in both EX-B and EX-L (Figure 12) and outperforms EX-L by up to 68.1% and 39.8% for Server SGX-like and SEV-like, respectively. However, the worker–responder interactions that are not present in BL are now split across enclave cores and SSE engines, adding undesirable data movement between them that prevents SSE-ONCPU from matching BL performance.

Both SSE-ONCPU-NHP and SSE-ONCPU-HP incur a greater memory access latency than BL due to added memory system stress caused by data movement in worker-responder interactions. Consequently, compared to BL, an increase in L1D sharing misses of 33.3× and 14.8× are incurred by SSE-ONCPU-NHP, and an increase in L1D sharing misses of 13.6× and 5.2× are incurred by SSE-ONCPU-HP for the SGX-like and SEV-like configurations, respectively. Compared to BL, SSE-ONCPU-NHP (which has a reduced L2 cache per tile), incurs 3.3× and 1.9× more L2 misses, while SSE-ONCPU-HP incurs 0.3% and 0.5% more L2 misses for the SGX-like and SEV-like configurations, respectively. These increases in L1D sharing and L2 misses result in a performance loss in the workload, pack, and syscall processing components of worker execution latency.

2) SSE-OFFCPU Analysis: Like each SSE-ONCPU configuration, SSE-OFFCPU also eliminates ecall/ocall and polling overheads incurred in both EX-B and EX-L but does so with a different architectural implementation that has its tradeoffs. Overall, SSE-OFFCPU offers a 41.8% and 25% improvement over EX-L but performs up to 18.5% and 11.8% worse than SSE-ONCPU for the SGX-like and SEV-like configurations, respectively.

Figure 12 shows that the main reason SSE-OFFCPU performs worse than SSE-ONCPU-HP is due to the 53.6% and 38.9% longer syscall processing times incurred compared to SSE-ONCPU-HP for the SGX-like and SEV-like configurations,

respectively. This is due to two primary reasons: (i) SSE engines operate a lower frequency ($1GHz$ for SSE-ONCPU vs $250MHz$ for SSE-OFFCPU) and (ii) SSE-OFFCPU incurs longer shared memory access latencies because all shared L2 slices are located on the CPU, further away from the SSE engines than in the SSE-ONCPU configuration. Data evicted from the private caches of the SSE engines must travel a longer distance back to enclave cores than in SSE-ONCPU, resulting in 38.9% and 35.3% longer sharing miss latencies for the SGX-like and SEV-like configurations, respectively, from the perspective of the memory system.

C. Performance Evaluation of Database Benchmark

Figure 10c shows EX-B and EX-L incur a 2.8× and 1.9× loss in performance compared to BL, respectively. Figure 12c shows that similar to Server, the primary loss in performance in EX-L for Database comes from polling overheads.

SSE-Software eliminates ecall/ocall and polling overheads. For 1-1 configuration, Figure 10c shows that SSE-Software performs similar EX-B. Figure 12c shows an increase in latencies due to the loss of worker cores. For 3-1 configuration, SSE-Software outperforms EX-B and EX-L for SGX-like database workload due to the availability of additional worker cores. For the 7-1 configuration, Figure 12c shows that the syscalls serialization penalty outperforms the availability of additional worker cores and SSE-Software performance degrades.

1) SSE-ONCPU Analysis: Similar to Server, SSE-ONCPU outperforms EX-L by up to 46.8%, but trails BL performance by as much as 31.3% due to the added data movement between workers and responders (Figure 12c). Each worker–responder interaction in the Database involves transferring a $2KB$ query which places greater stress on each interaction. For the Server only two system calls and one hypercall involve exchanging a large amount of data ($16KB$ web pages) while all other system calls and hypercalls transfer less than 200 bytes of data between workers and responders, resulting in less data being exchanged between workers and responders compared to the Database.

2) SSE-OFFCPU Analysis: SSE-OFFCPU suffers from its performance degradation which causes a 60.6% loss in performance compared to BL. As in Server, this performance loss is caused by the (i) slower frequency that SSE engines operate for this particular configuration and (ii) increased data movement latencies exacerbated by the distance this data must travel between workers and responders. Overall, this leads to SSE-OFFCPU incurring 27× and 79× longer L2 serialization and sharing latencies compared to BL, respectively.

Overall, the evaluation indicates that as the total number of threads exceeds the core count of a machine, exitless calling becomes less beneficial because of significant polling and context switching overheads. However, by providing enough SSE engines such that the hardware's parallelism is not overburdened as proposed in each of the SSE implementations (SSE-ONCPU and SSE-OFFCPU), polling and context switching overheads can be obviated and continued performance scaling inline with BL is achieved.

SSE Cores	4	8	16	32	64
FPGA LEs	108k	216k	432k	864k	1728k

TABLE II: SSE-OFFCPU Area Estimates using FPGA

D. Area Overhead Analysis

SSE comes at the cost of additional simple in-order pipelines and minimal private caches. For ON-CPU implementation, two multicore implementations are evaluated: (i) SSE-ONCPU-NHP that reduces the size of the shared L2 cache in each tile to compensate for the hardware overhead of SSE, and (ii) SSE-ONCPU-HP that pays for an added cost of SSE in each tile to improve performance. An in-order RISC-V core implementation has been shown to require 15k Logic Elements (LE) on an FPGA [36]. A 1KB direct-mapped L1 cache takes 6k LEs to implement on an FPGA [37]. For SSE-OFFCPU implementation, Table II shows implementation overhead in FPGA LEs based on two L1 caches per core. The latest on-chip CPU+FPGA systems, such as Intel® Stratix® 10 MX FPGA [38], support more than 2 million LEs, which is sufficient to implement up to 64 SSE cores.

VIII. Conclusion

This paper proposes a novel heterogeneous secure multi-core processor architecture that leverages Security Service Engines (SSEs) to improve the performance scaling of highly-interactive security-centric applications. When using state-of-the-art enclave-based execution leveraging exitless calling, these types of applications incur unnecessary stalls and frequent context switches due to the spawning of additional insecure responder threads that compete for the same hardware resources as application worker threads and lead to the saturation of a machine's hardware resources. With SSE, responder threads execute on their distinct hardware resources which obviates resource contention between workers and responders and thus overcomes the overheads present in enclave-based execution with exitless calling. By overcoming these overheads, an increase in performance of up to 68% is achieved. Furthermore, SSE enables performance scaling even when the total number of application threads exceeds the number of cores and comes within 5% of the performance that a security-centric application executing with no hardware security primitives achieves. Overall, SSE offers an efficient solution that makes enclave-based execution practical.

IX. Acknowledgements

This research was supported by the National Science Foundation under Grant No. 1929261.

References

[1] D. C. Chou, "Cloud Computing: A Value Creation Model," *Computer Standards & Interfaces*, vol. 38, pp. 72–77, Feb. 2015.
[2] V. Costan and S. Devadas, "Intel SGX Explained," Tech. Rep. 086, 2016.
[3] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption," 2016.
[4] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović, "Keystone: An Open Framework for Architecting TEEs," *arXiv:1907.10119 [cs]*, Sept. 2019. arXiv: 1907.10119.

[5] D. P. Mulligan, G. Petri, N. Spinale, G. Stockwell, and H. J. M. Vincent, "Confidential computing—a brave new world," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pp. 132–138, 2021.

[6] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," pp. 857–874, 2016.

[7] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: Secure Enclaves in a Speculative Out-of-Order Processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, (Columbus OH USA), pp. 42–56, ACM, Oct. 2019.

[8] H. Omar and O. Khan, "IRONHIDE: A Secure Multicore that Efficiently Mitigates Microarchitecture State Attacks for Interactive Applications," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 111–122, Feb. 2020. ISSN: 2378-203X.

[9] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "CURE: A security architecture with CUstomizable and resilient enclaves," p. 19, 2021.

[10] R. Boivie and P. Williams, "SecureBlue++: CPU Support for Secure Executables," tech. rep., Sept. 2016.

[11] O. Weisse, V. Bertacco, and T. Austin, "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, (Toronto ON Canada), pp. 81–93, ACM, June 2017.

[12] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS Services for SGX Enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, (Belgrade Serbia), pp. 238–253, ACM, Apr. 2017.

[13] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: a practical library OS for unmodified applications on SGX," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pp. 645–658, USENIX Association. ZSCC: 0000263.

[14] S. Kumar, A. Panda, and S. R. Sarangi, "A comprehensive benchmark suite for intel SGX."

[15] B. D'Agostino and O. Khan, "Seeds of SEED: Characterizing Enclave-level Parallelism in Secure Multicore Processors," 2021.

[16] J. Nye and O. Khan, "SSE: Security service engines to accelerate enclave performance in secure multicore processors," vol. 21, no. 2, pp. 129–132, 2022. Conference Name: IEEE Computer Architecture Letters.

[17] H. Tian, Q. Zhang, S. Yan, A. Rudnitsky, L. Shacham, R. Yariv, and N. Milshten, "Switchless Calls Made Practical in Intel SGX," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, (Toronto Canada), pp. 22–27, ACM, Jan. 2018.

[18] O. Shafi and J. Bashir, "Secsched: Flexible scheduling in secure processors," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, (New York, NY, USA), p. 229–240, Association for Computing Machinery, 2020.

[19] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12. ISSN: 2378-203X.

[20] H. Dogan, M. Ahmad, B. Kahne, and O. Khan, "Accelerating synchronization using moving compute to data model at 1,000-core multicore scale," *ACM Trans. Archit. Code Optim.*, vol. 16, pp. 4:1–4:27, Feb. 2019.

[21] lighttpd, "lighttpd. An open-source web server optimized for speed-critical environments.." https://www.lighttpd.net/.

[22] Y.-K. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms," vol. 12, no. 1, pp. 1–20.

[23] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta, "A reconfigurable computing system based on a cache-coherent fabric," in *2011 International Conference on Reconfigurable Computing and FPGAs*, pp. 80–85. ISSN: 2325-6532.

[24] D. Zhang, X. Ma, M. Thomson, and D. Chiou, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 593–607, ACM.

[25] B. C. Schwedock, P. Yoovidhya, J. Seibert, and N. Beckmann, "täkō: a polymorphic cache hierarchy for general-purpose optimization of data movement," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 42–58, ACM.

[26] M. Shan and O. Khan, "HD-CPS: Hardware-assisted drift-aware concurrent priority scheduler for shared memory multicores," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 528–542. ISSN: 2378-203X.

[27] S. Kumar, H. Raj, K. Schwan, and I. Ganev, "Re-architecting VMMs for multicore systems: The sidecore approach,"

[28] J. Liu and B. Abali, "Virtualization polling engine (VPE): using dedicated CPU cores to accelerate i/o virtualization," 2009.

[29] "nginx - advanced load balancer, web server, & reverse proxy." https://nginx.org. Publisher: Nginx, Inc.

[30] "Apache HTTP server project." https://httpd.apache.org/. Publisher: Apache Software Foundation.

[31] "memcached - a distributed memory object caching system." https://memcached.org/.

[32] "Redis - in-memory nosql database." https://redis.io/.

[33] "Linux Kernel 6.1." https://www.kernel.org/, 2023.

[34] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers, "The simon and speck lightweight block ciphers," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2015.

[35] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proceedings of the 2007 workshop on Experimental computer science - ExpCS '07*, ACM Press.

[36] "Ariane: An open-source 64-bit RISC-V Application-Class Processor and latest Improvements." https://riscv.org/wp-content/uploads/2018/05/14.15-14.40-FlorianZaruba_riscv_workshop-1.pdf, 2018.

[37] P. Yiannacouras and J. Rose, "A parameterized automatic cache generator for fpgas," in *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798)*, pp. 324–327, 2003.

[38] "Intel® Stratix® 10 MX 2100 FPGA." https://www.intel.com/content/www/us/en/products/sku/210297/intel-stratix-10-mx-2100-fpga/specifications.html, 2017.