2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Secure and Dynamic Core and Cache Partitioning for Safe and Efficient Server Consolidation

Myeonggyun Han
School of ECE, UNIST
hmg0228@unist.ac.kr

Seongdae Yu
School of ECE, UNIST
sd3392@unist.ac.kr

Woongki Baek
School of ECE, UNIST
wbaek@unist.ac.kr

***Abstract*—With server consolidation, latency-critical and batch workloads are collocated on the same physical servers. The resource manager dynamically allocates the hardware resources to the workloads to maximize the overall throughput while providing the service-level objective (SLO) guarantees for the latency-critical workloads. As the hardware resources are dynamically allocated across the workloads on the same physical server, information leakage can be established, making them vulnerable to micro-architectural side-channel attacks. Despite extensive prior works, it remains unexplored to investigate the efficient design and implementation of the dynamic resource management system that maximizes resource efficiency without compromising the SLO and security guarantees.**

To bridge this gap, this work proposes SDCP, secure and dynamic core and cache partitioning for safe and efficient server consolidation. In line with the state-of-the-art dynamic server consolidation techniques, SDCP dynamically allocates the hardware resources (i.e., cores and caches) to maximize the resource utilization with the SLO guarantees. In contrast to the existing techniques, however, SDCP dynamically sanitizes the hardware resources to ensure that no micro-architectural side channel is established between different security domains. Our experimental results demonstrate that SDCP provides high resource sanitization quality, incurs small performance overheads, and achieves high resource efficiency with the SLO and security guarantees.

I. Introduction

Server consolidation is a highly-effective technique for maximizing the resource efficiency of large-scale cloud-computing systems and datacenters. With server consolidation, latency-critical and batch workloads are collocated on the same physical servers. The main design goal of the resource manager for efficient server consolidation is to maximize the resource utilization while satisfying the service-level objective (e.g., the 99th tail latency of 200ms) of the latency-critical workloads.

Resource partitioning is a widely-used technique to provide the SLO guarantees for the latency-critical workloads by isolating the potential performance interference between the collocated latency-critical and batch workloads. Prior works have extensively investigated the dynamic resource partitioning techniques [16, 21, 24, 30]. Their common approach is to continuously monitor the performance and load of the latency-critical workloads and dynamically partition the hardware resources between the latency-critical and batch workloads.

While effective, the existing dynamic resource partitioning techniques are vulnerable to micro-architectural side-channel attacks. This is because they reallocate hardware resources between workloads in different security domains, leaking the sensitive information stored in the reallocated hardware resources. Considering that micro-architectural side-channel attacks are actively developed against various applications (e.g., cryptographic [13] and genome-sequencing applications [25]) that run on cloud-computing platforms and datacenters, it is crucial to develop the system software support for dynamic server consolidation, which manages the hardware resources in a secure and efficient manner.

To bridge this gap, this work proposes SDCP, secure and dynamic cache partitioning for safe and efficient server consolidation. The key difference of SDCP from the state-of-the-art dynamic resource management systems is that SDCP dynamically *sanitizes* the hardware resources when they are reallocated between different security domains to ensure secure and efficient server consolidation.

Specifically, this paper makes the following contributions:

- We propose SDCP, a dynamic resource management system that enables safe and efficient server consolidation through secure and dynamic core and cache partitioning. SDCP dynamically partitions the hardware resources between secure and non-secure containers and performs resource sanitization to prevent any information leakage.
- We design and implement a prototype of SDCP by extending the Docker framework [20], which is a widely-used container manager. We demonstrate that SDCP can be effectively implemented on commodity server systems.
- We quantify the effectiveness of SDCP on a real system using widely-used latency-critical and batch applications [2, 5]. Through quantitative evaluation, we show that SDCP provides high resource sanitization quality, incurs small performance overheads, and achieves high resource efficiency with the SLO and security guarantees.

II. Background and Motivation

A. Container-Based Virtualization

Container-based operating-system virtualization is a widely-used technique in cloud and datacenter computing environments [11]. In comparison with the virtual machine managers that provide *hardware virtualization*, container-based OS virtualization is lightweight and performance-efficient because it requires no complex system software stacks involved with hardware virtualization such as virtual CPUs and second-level address translation (e.g., extended page tables) [11].

Containers provide two major functionalities for OS virtu-

978-1-5386-5815-4/18/$31.00 ©2018 IEEE
DOI 10.1109/CCGRID.2018.00046

alization. First, containers provide namespace virtualization, which allows for each of the containers on the same host OS to have its own root file system. Namespace virtualization is highly effective for program development, debugging, and deployment as each container maintains its own well-tested libraries, tools, and directory structures without affecting the root file system of the host OS.

Second, containers provide resource partitioning among the containers on the same host OS based on the well-established Linux control group (Cgroups) functionality. With resource partitioning, each container is allocated with its own dedicated hardware resources (e.g., cores and NUMA memory nodes), which is highly effective for reducing the performance interference between the workloads [16].

B. Micro-Architectural Side-Channel Attacks

Micro-architectural side-channel attacks are security attacks, which exploit the vulnerabilities that the victim may leak the sensitive information to the attacker through the side channel established by the use of shared hardware resources. For instance, the PRIME+PROBE attack on the last-level cache (LLC), which is a highly powerful micro-architectural side-channel attack [13], works as follows. First, the attacker *primes* cache lines in the target cache set in the LLC. Second, the victim accesses a cache line in the target cache set, which evicts one of the cache lines that was primed by the attacker. Third, the attacker *probes* the primed cache lines in the target cache set again and infers whether the victim has accessed the sensitive code or data with high accuracy.

Prior works have extensively developed various micro-architectural side-channel attacks on private caches [22], branch target buffers (BTBs) [4, 27], and LLCs [22, 25, 29]. In addition, prior works have shown that various sensitive applications such as cryptographic [13] and genome-sequencing applications [25] are vulnerable to micro-architectural side-channel attacks. To exacerbate the problem, even the latest processors with hardware security extensions such as ARM TrustZone [12] and Intel SGX [25] are shown to be vulnerable to micro-architectural side-channel attacks.

Fundamentally, there are three conditions for micro-architectural side-channel attacks to occur. First, the states of some hardware components (e.g., LLC) must be shared between the victim and attacker. Second, the victim and attacker are allowed to change the states of the shared hardware component by performing their normal operations. For instance, if the victim executes a sequence of instructions, they are cached in the LLC. Third, the attacker must be able to precisely detect the state changes made by the victim. The attacker usually employs accurate timing tools such as the time stamp counter (TSC) or hardware performance counters (e.g., cycles) to detect the state changes.

If a security countermeasure reliably breaks any of these conditions, it can reliably defeat micro-architectural side-channel attacks. It is generally impractical to break the second or third condition because it may significantly degrade the performance (e.g., making all the memory operations uncacheable) or compromise the correctness and/or capability (e.g., accurate timing information is required for real-time data processing systems) of the applications.

C. Need for Safe and Efficient Server Consolidation

Server consolidation is a highly effective technique for maximizing the resource utilization in large-scale cloud and datacenter computing environments [16, 21, 24, 30]. With server consolidation, latency-critical and batch workloads are collocated on the same physical servers. The main design goal of the resource manager is to maximize the resource utilization while providing the service-level objective (SLO) guarantees.

Resource partitioning is a widely-used technique to provide the SLO guarantees for latency-critical workloads. There are mainly two approaches for resource partitioning. The first approach is *static partitioning*, which statically partitions the hardware resources between the latency-critical and batch workloads. While simple, static partitioning techniques fail to achieve the SLO guarantees for latency-critical workloads and/or high resource utilization when the load for latency-critical workloads fluctuates.

To address the limitations of static partitioning, prior works have extensively investigated the dynamic resource partitioning techniques for efficient server consolidation with SLO guarantees [16, 21, 24, 30]. With dynamic resource partitioning, the resource manager dynamically monitors the performance and load intensity of the latency-critical workloads and allocates the hardware resources between the latency-critical and batch workloads.

While effective, the state-of-the-art dynamic resource partitioning techniques are vulnerable to micro-architectural side-channel attacks. This is mainly because the hardware states can be fully shared between workloads in different security domains when the hardware resources are dynamically reallocated across security domains. To enable safe and efficient server consolidation, it is crucial to investigate the system software support for dynamic resource partitioning, which manages the hardware resources in a secure and effective manner.

III. THREAT AND SECURITY MODELS

A. Threat Model

The threat model assumed in this work is as follows. First, the OS kernel and the container manager are trusted. Second, containers are untrusted and have the access to accurate timing tools (e.g., Time Stamp Counter (TSC)). Therefore, if any hardware component is shared between the victim and attacker containers, it is vulnerable to micro-architectural side-channel attacks. Third, containers cannot directly change the resources allocated to other containers. To receive more hardware resources, containers must explicitly make requests to the container manager, which is common in the current practice of cloud and datacenter computing [20].

B. Security Model

Our security model assumes that there are two types of containers in terms of security – *secure* and *non-secure* containers. Secure containers are assumed to contain sensitive data that must not be revealed outside themselves. In contrast, non-

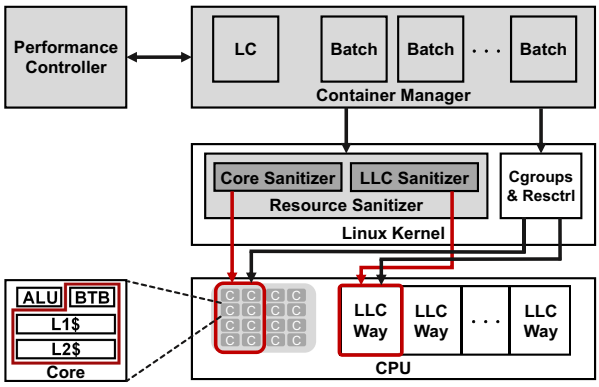


Fig. 1. Overall architecture of SDCP

secure containers are assumed to contain no sensitive data.

The security level of each container is set by the service provider with the mutual agreement between the owner of the container and the service provider. Once the security level of the container is set, it cannot be manipulated by any user (even the owner of the container).

The security policy enforced by SDCP is as follows. First, no information leakage from a secure container to other containers through micro-architectural side-channel attacks is allowed because secure containers are assumed to contain sensitive data. Second, information leakage from non-secure containers to other containers is acceptable as non-secure containers are assumed to contain no sensitive data.

IV. Design and Implementation

This section discusses the design and implementation of SDCP, which consists of three main components – (1) the resource sanitizer, (2) the performance controller, and (3) the container manager. Figure 1 shows the architecture of SDCP.

The design principles applied to SDCP are as follows. First, SDCP requires no hardware modifications and employs the hardware features available in recent commodity processors. Second, SDCP aims to minimize the potential performance overheads, especially the code related to resource sanitization.

While we aim to discuss the design and implementation of SDCP in an architecture-agnostic manner, we describe some of the techniques implemented in SDCP in the context of the `x86-64` architecture because the current implementation of SDCP employs some of the `x86-64` features. Section IV-E discusses alternative designs on other architectures.

A. Resource Sanitizer

The resource sanitizer sanitizes the private caches (i.e., L1 and L2 caches), the branch target buffer (BTB) of cores, and LLC ways when they are reallocated from a secure container to prevent any information leakage. The resource sanitizer comprises three components – the private cache sanitizer, the BTB sanitizer, and the LLC sanitizer.

One of the conditions for side-channel attacks is that the attacker must be able to observe the states produced by the victim. For instance, the Prime+Probe attack builds on the property that the attacker is able to precisely determine whether a cache line of interest has been accessed by the victim. The rationale behind resource sanitization is to defeat micro-architectural side-channel attacks by breaking this condition. SDCP ensures that the attacker only observes the states produced by the resource sanitizer but not by the victim, eliminating the side channels that can be established through the private caches, BTB, and LLC.

We design and implement the resource sanitizer as a Linux kernel module mainly because it uses privileged instructions (e.g., `WBINVD`). The user-level components of SDCP communicate with the resource sanitizer through the `IOCTL` interface.

Private Cache Sanitizer: It sanitizes the private caches (i.e., L1 and L2 caches) of a core. The current implementation of SDCP employs the `WBINVD` instruction available in the `x86-64` architecture, which invalidates all the cache lines in the private caches of the core where the instruction is executed [9]. If there are cache lines that hold the same data in the LLC, they are also invalidated.

BTB Sanitizer: It sanitizes the BTB of a core. Since there is no `x86-64` instruction that invalidates the BTB, we synthesize the assembly code that comprises a sequence of instructions to sanitize the BTB. This technique is inspired from the reverse-engineering technique proposed to experimentally determine the BTB capacity of recent Intel CPUs [27].

Listing 1. The `x86-64` assembly code of the BTB sanitizer

```
1000            <sanitizeBTB>:
...             ...
1004            <B_0>:
1004            jmp <B_1>
1006            nop
...             ...
1004+(N-1)*D    <B_N-1>:
1004+(N-1)*D    jmp <fini>
1006+(N-1)*D    nop
...             ...
```

Algorithm 1 Pseudocode for BTB sanitization code generator

```
procedure BTBSanitizationCodeGenerator
    sanitizeBTB ← NULL; maxBTBHits ← 0
    for N in candidateValuesN ▷ {512, 1024, 1536, ···, 8704}
        for D in candidateValuesD ▷ {2, 4, 8, 16, 32, 64}
            candidateSanitizeBTB ← genSanitizeCode(N, D)
            beginPerfCounters()
            for i in {0, 1, ···, M − 1} ▷ M = 100
                candidateSanitizeBTB()
            resteers ← endPerfCounters() / M
            currBTBHits ← N − resteers
            if currBTBHits > maxBTBHits
                maxBTBHits ← currBTBHits
                sanitizeBTB ← candidateSanitizeBTB
    return sanitizeBTB
```

Listing 1 shows the `x86-64` assembly code of the BTB sanitizer. The BTB sanitizer executes N unconditional jump instructions, each of which is separated with a distance of D bytes in the virtual address space. The BTB sanitizer ensures that the attacker only observes the states generated by the BTB sanitizer, defeating BTB-based side-channel attacks.

As an offline process, we aim to experimentally find the values for the N and D parameters, with which the BTB

sanitizer fills as many BTB entries as possible by executing the `sanitizeBTB` code in Listing 1. Algorithm 1 shows the code generation algorithm for the BTB sanitizer, which experimentally determines the optimal values for N and D.

For given values N and D, the code generation algorithm generates the `candidateSanitizeBTB` function in the `x86-64` assembly (Line 5). It then repeatedly executes the `candidateSanitizeBTB` function (Lines 7–8 in Algorithm 1) and measures the number of the `BTB resteers` performance events using the hardware performance counters available in the `x86-64` architecture (Lines 6 and 9).[1]

If the measured BTB resteer ratio (i.e., BTB resteers per branch instruction) is $r_{N,D}$, the BTB generates $N \cdot (1 - r_{N,D})$ BTB hits, indicating that the BTB consists of at least $N \cdot (1 - r_{N,D})$ entries. To maximize the BTB sanitization quality, the code generation algorithm iterates the candidate values of N and D (Lines 3–4), finds N and D such that $N \cdot (1 - r_{N,D})$ is maximized (Lines 10–13), and returns the `sanitizeBTB` code generated with the optimal N and D values (Line 14).

LLC sanitizer: It sanitizes an LLC way. The challenges for LLC sanitization is as follows. First, there is no instruction that invalidates an LLC way in the `x86-64` architecture. Thus, we need to synthesize the LLC sanitization code that executes a sequence of memory operations that overwrite the target LLC way with high coverage.

Second, the exact set-indexing function of the LLC in recent Intel CPUs is undisclosed [13]. The LLC is physically divided into *slices*. To ensure that each slice is equally utilized, the LLC set-indexing logic employs some physical address bits as an input to the hash function, which determines the slice ID where the address is mapped. This hash function is undisclosed and requires non-trivial efforts for reverse engineering [13].

To address these challenges, we propose an LLC sanitization technique based on the hardware performance counters. Our LLC sanitization technique is inspired from the technique proposed in [17], which is developed to investigate the undocumented LLC indexing functions used in recent Intel CPUs.

Specifically, we employ the hardware performance event, which counts the number of accesses to each slice [9]. For an arbitrary virtual address, we can precisely determine its associated slice ID by repeatedly executing the `clflush` instruction and identifying the frequently-accessed slice ID using this performance event. Further, since the lower S bits of the cache-block address is used as the set ID, we can precisely determine the slice and set IDs associated with the address. Based on this information, the LLC sanitizer automatically constructs the *address set* that comprises virtual addresses, which cover all the slices and sets of the target LLC way.

Algorithm 2 shows the pseudocode for the address-set construction algorithm. During the start-up time of SDCP, the LLC sanitizer allocates a memory buffer (Line 1 in Algorithm 2). Each of the cache line-sized memory objects in the memory buffer is used to sanitize a cache line of the target LLC way. The memory buffer size should be at least the size of an LLC way to ensure high LLC sanitization quality.

[1]The `BTB resteers` performance event increments when the control flow is resteered due to a BTB misprediction.

Algorithm 2 Pseudocode for address-set construction

```
1: procedure CONSTRUCTADDRSETS(baseAddr, size)
2:   addr ← baseAddr; endAddr ← baseAddr + size
3:   initialize(addrSetPerSlice)
4:   while addr < endAddr
5:     sliceId ← getSliceId(addr)
6:     setId ← getSetId(addr)
7:     if addrSetPerSlice[sliceId].contains(setId) = false
8:       addrSetPerSlice[sliceId].insert(setId, addr)
9:     addr ← addr + llcLineSize
```

Algorithm 3 Pseudocode for LLC sanitization

```
1: procedure SANITIZELLC(wayId)
2:   for sliceId in SliceIds
3:     for addr in addrSetPerSlice[sliceId]
4:       clflush(addr)
5:   setAllocatedWays(wayId)
6:   for sliceId in SliceIds
7:     for addr in addrSetPerSlice[sliceId]
8:       write(addr, v_Dummy)
```

For each cache line-sized memory object in the memory buffer, the LLC sanitizer precisely determines its associated slice and set IDs (Lines 5–6). The LLC sanitizer maintains an array of maps (Line 3), which is indexed by the slice ID. The map is a collection of key-value pairs, where the key is the set ID and the value is the address of the memory object.

The LLC sanitizer iterates every cache line-sized memory object in the buffer (Lines 4–9) and inserts the address of the memory object if the corresponding map does not contain any item with the key, which is the set ID associated with the memory object (Lines 7–8). Note that the address-set construction is performed only once during the start-up time of SDCP, incurring no performance overheads at runtime.

The LLC sanitizer sanitizes the target LLC way using the constructed address set. Algorithm 3 shows the pseudocode for the LLC sanitization, which mainly consists of the two steps. During the first step, the LLC sanitizer executes the `clflush` instruction against all the addresses in the constructed address set (Lines 2–4 in Algorithm 3). This step is necessary to ensure that no unexpected LLC cache hits occur during the second step, which actually sanitizes the target LLC way using the addresses in the address set. Note that the way partitioning technique implemented in Intel CPUs allows cache hits even when the requested cache line is in the LLC way that is not owned by the requester (i.e., a core or process) [7].

During the second step, the LLC sanitizer sets its LLC partition to the target LLC way (Line 5), reiterates the constructed address set, and writes a dummy value to each cache line (Lines 6–8). Since the LLC sanitizer ensures that the attacker only observes the states generated by the LLC sanitizer, SDCP defeats LLC-based side-channel attacks.

B. Performance Controller

The performance controller periodically collects the runtime information such as performance, load intensity, and resource utilization and dynamically controls the hardware resources based on the runtime information. The main goal of the

performance controller is to maximize the resource efficiency of the underlying server while satisfying the SLO of the latency-critical container in a coordinated and secure manner.

The hardware resources that are mainly controlled by the performance controller of SDCP are cores (and their private caches) and LLC, which are highly performance critical [16]. The performance controller builds on the dynamic core partitioning capabilities available in the Linux container [11, 20] and the Intel Cache Allocation Technology (CAT) [7] to dynamically partition the LLC across the containers.

Since there is no commodity processor with support for memory bandwidth partitioning [16], the current version of the performance controller does not explicitly perform the memory bandwidth partitioning across the containers. However, to provide the SLO guarantees for the latency-critical container, the performance controller dynamically monitors the memory bandwidth utilization and throttles the batch containers when the memory bandwidth utilization is high.

The metadata of the container is extended to contain the additional fields – (1) *secure*, which indicates whether the corresponding container is secure or not (i.e., non-secure) and (2) *latency critical*, which indicates whether the corresponding container is latency critical or not (i.e., batch). These additional fields can be only manipulated by the service provider but not by the users (even the owner of the container).

The performance controller enforces the following policies. First, hardware resources can be only shared by non-secure and batch containers. Latency-critical or secure containers are not allowed to share hardware resources with other containers to satisfy the SLO or security guarantees. We define the container group as a group of one or more containers that can share the same hardware resources. The performance controller groups all the non-secure and batch containers into a single container group. For consistency, we mainly use this term even if a container group only consists of a single container, which is the case for secure and/or latency-critical containers.

Second, the performance controller admits only a single latency-critical container group at any point of time to the underlying server. However, the performance controller may admit multiple batch container groups as long as it can satisfy the SLO guarantees for the latency-critical container group.

Inspired from the design proposed in one of the state-of-the-art techniques for dynamic server consolidation [16], the performance controller of SDCP comprises two types of controllers – (1) the long-term performance controller and (2) the short-term performance controller. The long-term performance controller mainly collects the performance and load-intensity data of the latency-critical container group and makes coarse-grain control decisions such as enabling/disabling batch container groups. The short-term performance controller mainly performs fine-grain resource management such as allocating more cores or LLC ways to batch container groups. In contrast with the prior works [16, 21, 24, 30], the performance controller of SDCP ensures that no information leakage occurs when hardware resources are dynamically reallocated between container groups in different security domains by robustly sanitizing them.

Algorithm 4 Pseudocode for the long-term controller

```
1:  procedure LONGTERMCONTROLLER
2:      flagGrowBatch ← false; createThread(shortTermController)
3:      while true
4:          tailLatency ← getTailLatency(); load ← getLoad()
5:          slack ← (targetLatency − tailLatency) / targetLatency
6:          if slack < 0
7:              flagGrowBatch ← false; disableCGs(enBatchCGs)
8:              disBatchCGs ← disBatchCGs ∪ enBatchCGs
9:              enBatchCGs ← ∅
10:         else if slack < s_T or load > l_{T,H}
11:             flagGrowBatch ← false
12:             reallocResources(CORE, enBatchCGs, lcCGs, 1)
13:             reallocResources(LLC, enBatchCGs, lcCGs, 1)
14:         else if load < l_{T,L}
15:             if disBatchCGs.size > 0
16:                 flagGrowBatch ← false
17:                 batchCG ← getNextDisabledCG(disBatchCGs)
18:                 allocMinResources(batchCG) ▷ 1 core and 2 LLC ways
19:                 enableCG(batchCG)
20:                 disBatchCGs ← disBatchCGs \ {batchCG}
21:                 enBatchCGs ← enBatchCGs ∪ {batchCG}
22:             else if enBatchCGs.size > 0
23:                 flagGrowBatch ← true
24:         sleep(T_Long)
```

Algorithm 4 shows the pseudocode for the long-term performance controller. The long-term controller operates at a rather long period (i.e., 15 seconds) to collect a sufficient amount of the performance data for statistical significance [16]. At every long-term controlling period, the long-term controller collects the tail latency and load statistics of the latency-critical container group (Line 4–5 in Algorithm 4). If the latency slack is negative, the long-term controller immediately disables all the batch container groups and allocates all the resources to the latency-critical container group (Lines 6–9).

If the latency slack is small or the load is high, the long-term performance controller triggers the safe guard mechanism by disabling the subsequent resource allocation to the batch container groups (Lines 10–13). Further, the long-term performance controller gradually reclaims the resources from one of the batch container groups and allocates them to the latency-critical container group.[2]

If the load is low,[3] the long-term performance controller allows for the resources to be allocated from the latency-critical container group to the batch container groups (Lines 14–23). If there is any batch container group that has been disabled due to insufficient resources, the long-term performance controller gradually re-enables each of them (Lines 15–21). If there is any enabled batch container group but no disabled batch container group, the long-term performance controller allows for the resources to be reallocated to the enabled batch container groups (Lines 22–23).

The long-term performance controller reallocates the resources between the latency-critical and batch container groups by invoking the `reallocResources` function (Al-

[2]For fairness, the long-term performance controller reclaims the resources from the container group allocated with the most resources per container.

[3]In this work, s_T, $l_{T,H}$, and $l_{T,L}$ are set to 0.35, 0.85, and 0.8, respectively

Algorithm 5 The `reallocResources` function

```
1: procedure REALLOCRESOURCES(type, from, to, N)
2:   while N--
3:     fromCG ← getCGwithMaxResource(type, from)
4:     if resCount(type, fromCG) = minResource(type)
5:       break
6:     resourceNum ← removeResource(type, fromCG)
7:     toCG ← getCGwithMinResource(type, to)
8:     if type = CORE
9:       addCore(toCG, resourceNum)
10:    else if type = LLC
11:      addLLCWay(toCG, resourceNum)
```

Algorithm 6 The `addCore` function

```
1: procedure ADDCORE(toCG, coreNum)
2:   ownerCG ← coreAllocTable[coreNum]
3:   if ownerCG.secure = true
4:     sanitizeCore(coreNum)
5:   setCoreOwner(coreNum, toCG)
```

gorithm 5). The `reallocResources` function invokes the `addCore` (Algorithm 6) or `addLLCWay` (Algorithm 7) function depending on the hardware resource type.

Before reallocating the hardware resource from the secure container group, the `addCore` or `addLLCWay` function sanitizes it using the resource sanitizer. While the `addCore` function simply sanitizes the core to be reallocated, the `addLLCWay` function shifts and sanitizes multiple ways (Lines 8–18 in Algorithm 7) even if it essentially reallocates a single LLC way. This is because the Intel CAT requires that each LLC partition must comprise consecutive LLC ways [7].

Algorithm 8 shows the pseudocode for the short-term performance controller. The short-term performance controller operates at a shorter period (i.e., 2 seconds) than the long-term performance controller to reallocate the hardware resources between the container groups in a fast and efficient manner [16].

At every short-term controlling period, the short-term performance controller measures the memory bandwidth utilization using the hardware performance counters (Lines 7–11 in Algorithm 8). If the current memory bandwidth utilization is higher than a predefined threshold (i.e., B_T), it reduces the amount of the resources allocated to one of the batch container groups. This is to ensure that the latency-critical container group receives sufficient memory bandwidth by throttling the memory-bandwidth usage of the batch container groups [16].

If the current memory bandwidth utilization is below the threshold and the `flagGrowBatch` is set to true, the short-term performance controller gradually reallocates the hardware resources from the latency-critical container group to the batch container groups (Lines 14–25). The short-term performance controller reallocates the resources to the batch container groups in an alternating manner in that it first reallocates a core, then an LLC way, and so on.

At every short-term controlling period, the short-term performance controller checks (Lines 26–27) if the amount of the available resources is below the thresholds (i.e., C_T, and L_T).[4]

[4]In this work, B_T, C_T, and L_T are set to 27GB/s, 3, and 3, respectively.

Algorithm 7 The `addLLCWay` function

```
1: procedure ADDLLCWAY(toCG, wayNum)
2:   if toCG.latencyCritical = true
3:     wayNumList ← {wayNum, ···, wayNumMin+1}
4:     step ← −1
5:   else
6:     wayNumList ← {wayNum, ···, wayNumMax−1}
7:     step ← +1
8:   for currWayNum in wayNumList
9:     nextWayNum ← currWayNum + step
10:    currCG ← wayAllocTable[currWayNum]
11:    nextCG ← wayAllocTable[nextWayNum]
12:    if currCG = toCG
13:      break
14:    else
15:      if currCG.secure = true and currCG ≠ nextCG
16:        setWayOwner(currWayNum, sanitizer)
17:        sanitizeLLC(currWayNum)
18:      setWayOwner(currWayNum, nextCG)
```

Algorithm 8 Pseudocode for the short-term controller

```
1: procedure SHORTTERMCONTROLLER
2:   phase ← CORE
3:   while true
4:     availCores ← getAvailableCores()
5:     availLLC ← getAvailableLLCWays()
6:     availResources ← availCores + availLLC
7:     memBandwidth ← getMemBandwidth()
8:     if memBandwidth > B_T
9:       reallocResources(CORE, enBatchCGs, lcCGs, 1)
10:      reallocResources(LLC, enBatchCGs, lcCGs, 1)
11:      goto label_sleep
12:    if flagGrowBatch = false or availResources = 0
13:      goto label_sleep
14:    if phase = CORE
15:      if availCores > 0
16:        reallocResources(CORE, lcCGs, enBatchCGs, 1)
17:        availCores ← availCores − 1
18:      if availLLC > 0
19:        phase ← LLC
20:    else if phase = LLC
21:      if availLLC > 0
22:        reallocResources(LLC, lcCGs, enBatchCGs, 1)
23:        availLLC ← availLLC − 1
24:      if availCores > 0
25:        phase ← CORE
26:    if availCores < C_T or availLLC < L_T
27:      flagGrowBatch ← false
28:  label_sleep:
29:    sleep(T_Short)
```

If so, it sets `flagGrowBatch` to false. This safe guard is to ensure that the hardware resources are reallocated to the batch container groups in a more gradual manner when the available hardware resources become scarce.

C. Container Manager

The container manager manages all the containers in the system. We currently use the Docker framework [20] as a baseline container management system and extend it to design and implement SDCP because of its widespread use and well-established development environment. However, we believe that SDCP can be also applied to other container management systems such as Apache Mesos [8] and Kubernetes [3].

Specifically, we have extended the Docker framework to revise some of the Docker commands and their implementation in order to allow for the performance controller to control the resources allocated to containers in a secure manner. For instance, the `update` command of Docker that is used to update the resources allocated to a container has been extended to annotate which resources must be sanitized before they are allocated to the container.

In addition, we have extended the Docker framework to establish the communication with the resource sanitizer, which is implemented as a Linux kernel module. It issues commands to the resource sanitizer through the `IOCTL` interface.

D. Implementation

The resource sanitizer is implemented as a Linux kernel module in the C programming language. The total lines of the code for the resource sanitizer are approximately 560 lines.

The performance controller is implemented as a user-level process in the Python programming language because the Docker framework provides a well-established API for the Python programming language. The total lines of the code for the performance controller are approximately 770 lines.

The container manager (i.e., the Docker framework) is implemented as a user-level process in the Go programming language. The total lines of the code that have been added or modified to implement SDCP are approximately 280 lines.

E. Discussion

Other architectures: The current implementation of SDCP employs the synthesized code to sanitize the BTB and the LLC because the `x86-64` architecture lacks dedicated instructions to invalidate them. On other architectures that provide instructions for the BTB and/or LLC way invalidation, SDCP can employ such instructions if they improve performance.

In contrast, the current implementation of SDCP employs the `WBINVD` instruction to sanitize the private caches. On other architectures that lack such instructions, SDCP can use the synthesized code to sanitize the private caches, similarly to the case with LLC sanitization.

SDCP employs the way partitioning functionality, which is widely supported in the recent commodity processors of Intel and ARM [7, 28]. If the underlying processor lacks the way partitioning functionality, SDCP can employ the page coloring technique [26], which essentially provides the set partitioning functionality in software.

V. Experimental Methodology

Benchmarks: To quantify the performance of SDCP, we use two latency-critical benchmarks and three batch benchmarks. We run each benchmark in a separate container. All the data reported in this work is the average of five experiments.

We use the `memcached` and `websearch` benchmarks from CloudSuite [5, 23] as the latency-critical workloads. The `memcached` benchmark is adapted from a production-quality in-memory key-value store [6] and uses the Twitter dataset with a scaling factor of 30 [5, 23]. The `websearch` benchmark is adapted from a production-quality web-search engine (Apache Solr [1]) and uses the 12GB index [5, 23].

TABLE I. System configuration

Component	Description
Processors	4× Intel Xeon Processor E5-4660 v4 CPUs @ 2.2GHz, 16 cores per CPU
L1 I-cache	Private, 32KB, 8 ways
L1 D-cache	Private, 32KB, 8 ways
L2 cache	Private, 256KB, 8 ways
L3 cache	Shared, 40MB, 20 ways
Memory	128GB (8 × 16GB DDR4)
OS	Ubuntu 16.04, Linux Kernel 4.10.0
Docker	17.06.0

The two latency-critical benchmarks include their own load generator. We configure the load generator of each latency-critical benchmark to generate a constant load and a diurnal load that is observed in real-world datacenters [15, 19].

The SLO for the `memcached` benchmark is that the 95th percentile request latency should be less than 1ms [24]. With this target latency, the maximum requests per second (RPS) that the server system evaluated in this work can handle is 450,000. The SLO for the `websearch` benchmark is that the 99th percentile query latency should be less than 200ms [5, 23]. With this target latency, the maximum queries per second (QPS) that the evaluated server system can handle is 150.

We use three benchmarks (i.e., `blackscholes` (`BL`) [2], `stream` (`ST`) [18], and `swaptions` (`SW`) [2]) as the batch workloads. They exhibit various architectural characteristics such as high compute intensity (e.g., `BL` and `SW`), high memory intensity (e.g., `ST`), and high sensitivity to the allocated LLC capacity (e.g., `SW`).

System Configuration: To quantify the performance of SDCP, we employ a 64-core NUMA system with four 16-core CPUs (Table I). We allocate a dedicated CPU to run the latency-critical container group and the batch container groups along with SDCP. To minimize performance interference, we also allocate a dedicated CPU to the load generator that generates the load to the latency-critical container group.

VI. Evaluation

A. Resource Sanitization Quality

We quantify the BTB and LLC sanitization quality. We define the sanitization quality as follows – $Quality(\%) = \frac{E_{Sanitized}}{E_{Total}} \times 100$, where E_{Total} and $E_{Sanitized}$ denote the total number of entries in the hardware component (i.e., BTB or an LLC way) to be sanitized and the number of entries that are actually sanitized by the resource sanitizer. Since the detailed BTB and LLC organizations of the recent Intel architectures are undisclosed, we use the customized code to sanitize them, which may fail to achieve the sanitization quality of 100%.

Using the hardware performance counters, we measure the front-end resteer ratio ($r_{N,D}$) for each jump instruction executed by the BTB sanitizer with different N and D values (Section IV-A). For given N and D, $E_{Sanitized}$ is at least $N \cdot (1 - r_{N,D})$. Our experimental results show that $E_{Sanitized}$ is maximized (i.e., 4405) when $N = 4608$ and $D = 16$.

Since the number of the BTB entries of the Intel Broadwell architecture is undisclosed, we are unable to calculate the BTB sanitization quality. To ensure that the BTB sanitizer achieves high quality, we also ran the BTB sanitization code

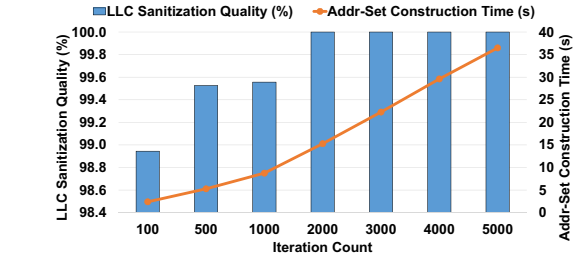
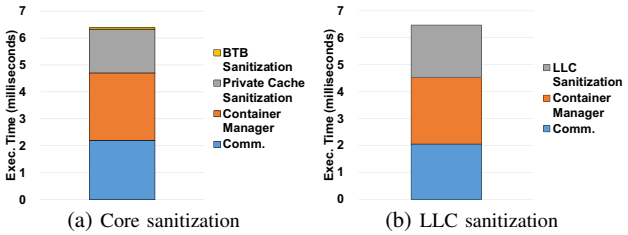


Fig. 2. LLC sanitization quality



(a) Core sanitization (b) LLC sanitization

Fig. 3. Performance overheads of resource sanitization

on an Intel Haswell CPU and checked that it achieves the BTB sanitization quality of 99.8% ($= \frac{4088}{4096}$), which is high. Assuming that the BTB organization of the Intel Broadwell architecture has not been significantly changed from that of the Intel Haswell architecture, we conjecture that the BTB sanitizer also achieves high quality on the Intel Broadwell architecture, which is equipped in our target server system.

We now quantify the LLC sanitization quality. Figure 2 shows the LLC sanitization quality and the address-set construction time with different iteration counts to determine the slice ID associated with each memory object in the 2MB memory buffer allocated for the LLC sanitizer (Section IV-A).

First, perhaps surprisingly, we observe that the LLC sanitizer achieves the sanitization quality of 100% with sufficiently large iteration counts when the 2MB memory buffer is used. Since the size of each LLC way is 2MB, this implies that the hash function that determines the slice ID for each address is perfect in that it incurs no conflicts. We tested with 50 different 2MB memory buffers and checked that the LLC sanitizer achieves the 100% sanitization quality in all the cases.[5]

Second, with small iteration counts, the LLC sanitization quality is slightly degraded. This is because the signal-to-noise ratio decreases when experimentally determining the slice ID of each address using the hardware performance counters with small iteration counts. However, since the LLC sanitizer performs a smaller number of the `clflush` instructions with small iteration counts, the address-set construction time[6] is significantly reduced, which may be a good design trade-off.

B. Performance Overheads

We investigate the performance overheads of the resource sanitizer. Figure 3a shows the core sanitization time breakdown, which comprises four segments – (1) the communication time between the performance controller and the container manager, (2) the command process time by the container manager, (3) the private cache sanitization time, and (4) the BTB sanitization time. We observe the following data trends.

First, the overall performance overheads of core sanitization are small (i.e., 6.4ms). Since the core sanitization occurs at the period (i.e., 2 seconds) of the short-term controller, the performance impact of the core sanitization is insignificant. Second, the private cache sanitization overheads are significantly larger than the BTB sanitization overheads. This is because the `WBINVD` instruction that the private cache sanitizer executes to sanitize the private caches incurs numerous write-backs for dirty cache lines to memory. In contrast, BTB sanitization is significantly faster because it incurs no write-backs.

We now investigate the performance overheads of LLC sanitization. Figure 3b shows the LLC sanitization time breakdown. Similarly to core sanitization, the overall performance overheads of LLC sanitization are small (i.e., 6.48ms) and its performance impact is insignificant, considering the period (i.e., 2 seconds) of the short-term controller.

Overall, our experimental results show that the resource sanitizer of SDCP is highly effective in that it achieves high sanitization quality with small performance overheads.

C. Performance and Resource Efficiency

We quantify the performance and resource efficiency of SDCP in terms of the tail latency (i.e., the SLO guarantees for the latency-critical workloads), the effective machine utilization (EMU) [16], and the resource (i.e., CPU and memory bandwidth) utilization. Specifically, EMU is defined as follows – $EMU = \frac{T_{LC}}{T_{LC,max}} + \frac{T_{Batch}}{T_{Batch,max}}$, where $T_{LC,max}$, $T_{Batch,max}$, T_{LC}, and T_{Batch} denote the solo-run throughput of the latency-critical container group with the full hardware resources, the solo-run throughput of the batch container group with the full hardware resources, the throughput of the latency-critical container group with the collocated batch container group, and the throughput of the batch container group with the collocated latency-critical container group, respectively.

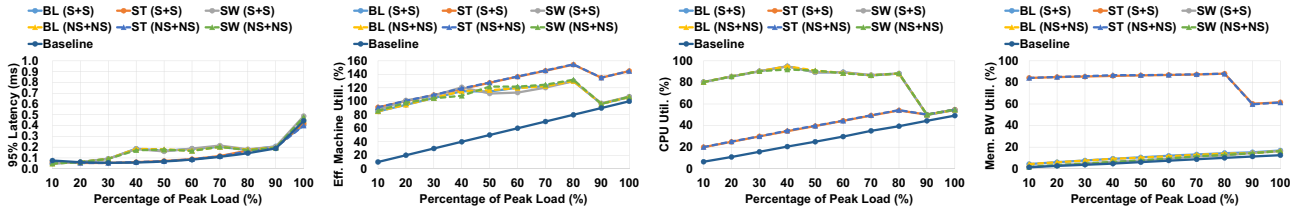
Figure 4 shows the performance and resource efficiency results of the `memcached` workload when it is collocated with each of the batch workloads. We execute each combination of the `memcached` and batch workloads in the following two security configurations – each of the two workloads runs in (1) a secure container group (i.e., S+S) and (2) a non-secure container group (i.e., NS+NS). The first configuration represents the worst-case performance scenario for SDCP because hardware resources are always sanitized when they are dynamically reallocated. In contrast, the second configuration represents the best-case performance scenario for SDCP because no hardware resource sanitization occurs.

For comparison, Figure 4 also shows the performance and resource efficiency results (i.e., baseline) when SDCP only executes `memcached` alone. To investigate the performance and resource-efficiency sensitivity to the load, we sweep it from 10% to 100% of the peak load. We observe the following.

First, SDCP achieves the similar performance and resource efficiency across the workloads in the two aforementioned security configurations (i.e., S+S and NS+NS). This is mainly

[5]In this work, the iteration count is set to 2500 (100% sanitization quality).

[6]Note that SDCP only performs the address-set construction during the start-up time, incurring no overheads at runtime.



(a) Latency (b) Effective machine utilization (c) CPU utilization (d) Memory bandwidth utilization

Fig. 4. Performance and resource efficiency results with the `memcached` workload

because the resource sanitizer of SDCP is highly efficient and optimized. Our experimental results demonstrate that the security features of SDCP incur little or no impact on performance and resource efficiency.

Second, Figure 4a shows that SDCP robustly provides the SLO guarantees (i.e., the 95th tail latency of 1ms) for `memcached` across the batch workloads and load levels. At low load levels, the tail latency of `memcached` with `ST` is lower than the other batch workloads. This is because SDCP throttles the amount of the hardware resources allocated to `ST` as it consumes a significant portion of the memory bandwidth of the system. Since SDCP allocates fewer hardware resources to `ST` than the other batch workloads, `memcached` is allocated with more resources, resulting in the reduced tail latency.

Third, Figure 4b shows that SDCP achieves significantly higher effective machine utilization (e.g., 2.4× higher at the 50% load on average) than the baseline (i.e., executing only `memcached`) across the batch workloads and load levels by effectively collocating the latency-critical and batch workloads on the same physical server. We observe that the EMU often exceeds 100%. This is mainly because the minimum amount of hardware resources required to satisfy the SLO is not strictly proportional to the load level. SDCP dynamically detects the excessive hardware resources allocated to the latency-critical workload and gradually reallocates them to the batch workload, achieving the EMU higher than 100%.

Fourth, Figure 4c shows that SDCP achieves higher CPU utilization (e.g., 2.7× higher at the 50% load on average) than the baseline. Especially, SDCP achieves significantly higher CPU utilization when collocating the `memcached` and core-intensive (e.g., `SW`) workloads. Since core-intensive workloads incur little or no performance interference to `memcached`, SDCP allocates as many cores to them as possible, achieving high CPU utilization.

In contrast, SDCP achieves lower CPU utilization when collocating the `memcached` and `ST` workloads. Since `ST` is a memory-intensive benchmark, SDCP throttles the hardware resources allocated to `ST` to provide the SLO guarantees for `memcached`. Therefore, SDCP allocates fewer cores to `ST`, achieving lower CPU utilization.

Fifth, Figure 4d shows that SDCP achieves higher memory bandwidth utilization (e.g., 3.2× higher at the 50% load on average) than the baseline. In particular, SDCP achieves significantly higher memory bandwidth utilization when collocating `memcached` and `ST`. This is because `ST` is a memory-intensive workload that demands high memory bandwidth. In contrast, SDCP achieves lower memory bandwidth utilization when collocating `memcached` and `SW` because they demand low memory bandwidth.

Figure 5 shows the performance and resource efficiency results of SDCP when the `websearch` and batch workloads are collocated. Similarly to the results with `memcached`, we observe that SDCP robustly provides the SLO guarantees (i.e., the 99th tail latency of 200ms) for `websearch`, achieves high effective machine utilization, and significantly improves the resource utilization. Further, the resource sanitization of SDCP incurs little or no impact on performance and resource efficiency when executing the `websearch` and batch workloads in secure container groups.

We now investigate the performance and resource efficiency of SDCP when a non-constant load is applied to the latency-critical workloads. In line with the prior works [15, 19], we use the diurnal load because it closely represents the realistic load observed in datacenters. The load generator for each latency-critical workload is configured to simulate a period of 24 hours. To keep the total experiment time manageable, we configure each hour in the original diurnal load pattern to correspond to 15 seconds.

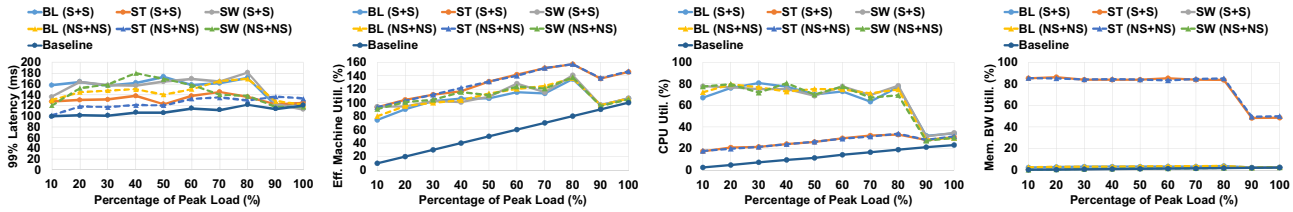
Figure 6 shows the performance and resource efficiency results of SDCP when collocating the `memcached` and `BL` workloads. We execute each workload in a separate secure container group to model the worst-case performance scenario. Even with this setting, SDCP robustly achieves high performance and resource efficiency while providing the SLO and security guarantees for the two workloads.

Overall, our quantitative evaluation demonstrates the effectiveness of SDCP in that it provides high resource sanitization quality, incurs small performance overheads even in the worst-case performance scenario in which all the container groups are secure container groups, and achieves high resource efficiency with the robust SLO and security guarantees.

VII. Related Work

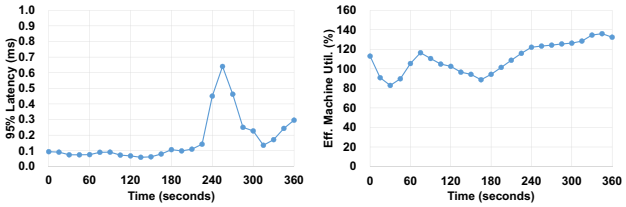
Prior works have extensively investigated the dynamic resource management techniques for efficient server consolidation [16, 21, 24, 30]. Their key theme is to collocate latency-critical and batch workloads on the same physical servers and dynamically allocate hardware resources to improve the resource efficiency with the SLO guarantees.

While effective, the prior works lack the resource partitioning and sanitization between the workloads in different security domains, making them vulnerable to micro-architectural side-channel attacks. In contrast, our work aims to achieve



(a) Latency (b) Effective machine utilization (c) CPU utilization (d) Memory bandwidth utilization

Fig. 5. Performance and resource efficiency results with the `websearch` workload



(a) Latency (b) Effective machine utilization

Fig. 6. Runtime behavior of SDCP

high resource utilization while simultaneously providing the SLO and security guarantees for the collocated workloads through secure and dynamic core and cache partitioning.

Prior works have proposed various techniques to develop and defend micro-architectural side-channel attacks [4, 10, 12, 13, 14, 25, 29]. While insightful, the prior works lack the consideration of the performance impact of enforcing the security policies on server consolidation when latency-critical and batch workloads are collocated on the same physical servers. Our work differs in that it proposes a secure and dynamic resource management system for safe and efficient server consolidation, presents the design and implementation of SDCP by extending a production-quality container management system (i.e., Docker), and quantifies the performance impact of SDCP in various scenarios where the latency-critical and batch container groups are collocated on the same physical server in different security domains.

VIII. Conclusions

This work presents SDCP, secure and dynamic core and cache partitioning for safe and efficient server consolidation. SDCP continuously collects the runtime information such as the performance and load intensity of the latency-critical container group and dynamically allocates the hardware resources between the latency-critical and batch container groups based on the runtime information. When a hardware resource is reallocated between different security domains, SDCP dynamically performs the fast and high-quality resource sanitization to prevent any information leakage, effectively defeating micro-architectural side-channel attacks. Our quantitative evaluation demonstrates the effectiveness of SDCP in that it provides high resource sanitization quality, incurs low performance overheads, and achieves high resource efficiency while robustly providing the SLO and security guarantees for the collocated container groups. As future work, we plan to investigate the secure and dynamic resource management techniques for heterogeneous computing systems whose devices are shared by the workloads in different security domains.

Acknowledgements

This research was partly supported by the National Research Foundation of Korea (No. NRF-2016M3C4A7952587, PF Class Heterogeneous High Performance Computer Development) and the Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-16-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development). Woongki Baek is the corresponding author.

References

[1] *Apache Lucene and Solr*. https://github.com/apache/lucene-solr.
[2] C. Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: PACT '08.
[3] B. Burns et al. "Borg, Omega, and Kubernetes". In: *Commun. ACM* (2016).
[4] D. Evtyushkin et al. "Jump over ASLR: Attacking branch predictors to bypass ASLR". In: MICRO '16.
[5] M. Ferdman et al. "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware". In: ASPLOS '12.
[6] B. Fitzpatrick. "Distributed Caching with Memcached". In: *Linux J.* (2004).
[7] A. Herdrich et al. "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family". In: HPCA '16.
[8] B. Hindman et al. "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center". In: NSDI'11.
[9] *Intel 64 and IA-32 Architectures Software Developer's Manual*.
[10] T. Kim et al. "STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud". In: Security'12.
[11] *Linux Containers*. https://linuxcontainers.org/.
[12] M. Lipp et al. "ARMageddon: Cache Attacks on Mobile Devices". In: Security'16.
[13] F. Liu et al. "Last-Level Cache Side-Channel Attacks are Practical". In: SP '15.
[14] F. Liu et al. "CATalyst: Defeating last-level cache side channel attacks in cloud computing". In: HPCA '16.
[15] D. Lo et al. "Towards Energy Proportionality for Large-scale Latency-critical Workloads". In: ISCA '14.
[16] D. Lo et al. "Heracles: Improving Resource Efficiency at Scale". In: ISCA '15.
[17] C. Maurice et al. "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters". In: RAID '15.
[18] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995).
[19] D. Meisner et al. "Power Management of Online Data-intensive Services". In: ISCA '11.
[20] D. Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* (2014).
[21] R. Nishtala et al. "Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads". In: HPCA '17.
[22] D. A. Osvik et al. "Cache Attacks and Countermeasures: The Case of AES". In: CT-RSA'06.
[23] T. Palit et al. "Demystifying cloud benchmarking". In: ISPASS '16.
[24] V. Petrucci et al. "Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers". In: HPCA '15.
[25] "Software Grand Exposure: SGX Cache Attacks Are Practical". In: WOOT '17.
[26] D. M. Tullsen et al. "Handling Long-latency Loads in a Simultaneous Multithreading Processor". In: MICRO '01.
[27] V. Uzelac et al. "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures". In: ISPASS '09.
[28] X. Wang et al. "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support". In: HPCA '17.
[29] Y. Yarom et al. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack". In: SEC'14.
[30] H. Zhu et al. "Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems". In: ASPLOS '16.