BabelFish: Fusing Address Translations for Containers

Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas
University of Illinois at Urbana-Champaign
{skarlat2, darbaz2, gopired2, nskim, torrella}@illinois.edu

***Abstract*—Cloud computing has begun a transformation from using virtual machines to containers. Containers are attractive because multiple of them can share a single kernel, and add minimal performance overhead. Cloud providers leverage the lean nature of containers to run hundreds of them on a few cores. Furthermore, containers enable the serverless paradigm, which leads to the creation of short-lived processes.**

In this work, we identify that containerized environments create page translations that are extensively replicated across containers in the TLB and in page tables. The result is high TLB pressure and redundant kernel work during page table management. To remedy this situation, this paper proposes *BabelFish*, a novel architecture to share page translations across containers in the TLB and in page tables. We evaluate BabelFish with simulations of an 8-core processor running a set of Docker containers in an environment with conservative container co-location. On average, under BabelFish, 53% of the translations in containerized workloads and 93% of the translations in serverless workloads are shared. As a result, BabelFish reduces the mean and tail latency of containerized data-serving workloads by 11% and 18%, respectively. It also lowers the execution time of containerized compute workloads by 11%. Finally, it reduces serverless function bring-up time by 8% and execution time by 10%–55%.

***Index Terms*—Virtual Memory, Containers, Address Translation, TLB, Page Tables.**

I. Introduction

Cloud computing is ubiquitous. Thanks to its ability to provide scalable, pay-as-you-go computing, many companies choose cloud services instead of using private infrastructure. In cloud computing, a fundamental technology is virtualization. Virtual Machines (VMs) allow users to share resources, while providing an isolated environment to each user.

Recently, cloud computing has been undergoing a radical transformation with the emergence of *Containers*. Like a VM, a container packages an application and all of its dependencies, libraries, and configurations, and isolates it from the system it runs on. However, while each VM requires a guest Operating System (OS), multiple containers share a single kernel. As a result, containers require significantly fewer memory resources and have lower overheads than VMs. For these reasons, cloud providers such as Google's Compute Engine [27], Amazon's ECS [3], IBM's Cloud [31], and Microsoft's Azure [52] now provide container-based solutions.

Container environments are typically oversubscribed, with many more containers running than cores [56]. Moreover, container technology has laid the foundation for *Serverless* computing [65], a new cloud computing paradigm provided by services like Amazon's Lambda [2], Microsoft's Azure Functions [53], Google's Cloud Functions [26], and IBM's Cloud Functions [32]. The most popular use of serverless computing is known as Function-as-a-Service (FaaS). In this environment, the user runs small code snippets called functions, which are triggered by specified events. The cloud provider automatically scales the number and type of functions executed based on demand, and users are charged only for the amount of time a function spends computing [39, 66].

Our detailed analysis of containerized environments reveals that, very often, the same Virtual Page Number (VPN) to Physical Page Number (PPN) translations, with the same permission bit values, are extensively replicated in the TLB and in page tables. One reason for this is that containerized applications are encouraged to create many containers, as doing so simplifies scale-out management, load balancing, and reliability [12, 33]. In such environments, applications scale with additional containers, which run the same application on different sections of a common data set. While each container serves different requests and accesses different data, a large number of the pages accessed is the same across containers.

Another reason for the replication is that containers are created with forks, which replicate translations. Further, since containers are stateless, data is usually accessed through the mounting of directories and the memory mapping of files. The result is that container instances of the same application share most of the application code and data pages. Also, both within and across applications, containers often share middleware. Finally, the lightweight nature of containers encourages cloud providers to deploy many containers in a single host [30]. All this leads to numerous replicated page translations.

Unfortunately, state-of-the-art TLB and page table hardware and software are designed for an environment with few and diverse application processes. This has resulted in per-process tagged TLB entries, separate per-process page tables, and lazy page table management — where, rather than updating the page translations at process creation time, they are updated later on demand. In containerized environments, this approach causes high TLB pressure, redundant kernel work during page table management and, generally, substantial overheads.

To remedy this problem, we propose *BabelFish*, a novel architecture to *share translations across containers* in the TLB and page tables — without sacrificing the isolation provided by the virtual memory abstractions. BabelFish eliminates the replication of translations in two ways. First, it modifies the

TLB to dynamically share identical {VPN, PPN} pairs and permission bits across containers. Second, it merges page table entries of different processes with the same {VPN, PPN} translations and permission bits. As a result, BabelFish reduces the pressure on the TLB, reduces the cache space taken by translations, and eliminates redundant minor page faults. In addition, it effectively prefetches shared translations into the TLB and caches. The end result is higher performance of containerized applications and functions, and faster container bring-up.

We evaluate BabelFish with simulations of an 8-core processor running a set of Docker containers in an environment with conservative container co-location. On average, under BabelFish, 53% of the translations in containerized workloads and 93% of the translations in FaaS workloads are shared. As a result, BabelFish reduces the mean and tail latency of containerized data-serving workloads by 11% and 18%, respectively. It also lowers the execution time of containerized compute workloads by 11%. Finally, it reduces FaaS function bring-up time by 8% and execution time by 10%–55%.

II. Background

A. Containers

Containers are a lightweight software virtualization solution that aims to ease the deployment of applications [44]. A container is defined by an image that specifies all of the requirements of an application, including the application binary, libraries, and kernel packages required for deployment. The container environment is more light-weight than a traditional VM, as it eliminates the guest operating system. All of the containers share the same kernel and, consequently, many pages can be automatically shared among containers. As a result, containers typically exhibit better performance and consume less memory than VMs [67]. Docker containers [21] is the most prominent container solution. In addition, there are management frameworks, such as Google's Kubernetes [28] and Docker's Swarm [20], which automate the deployment, scaling, and maintenance of containerized applications.

The lightweight nature of containers has led to the Function-as-a-Service (FaaS) paradigm [32, 24, 26, 2, 53]. In FaaS, the user provides small code snippets, called *Functions*, and providers charge users only for the amount of time a function spends computing. In FaaS environments, many functions can be running concurrently, which leads to high consolidation rates.

Containers rely on OS virtualization and, hence, use the process abstraction, rather than the thread one, to provide resource isolation and usage limits. Typically, a containerized application scales out by creating multiple replicated containers, as this simplifies load balancing and reliability [12, 33]. The resulting containers usually include one process each [22], and run the same application but use different sections of data. For example, a graph application exploits parallelism by creating multiple containers, each one with one process. Each process performs different traversals on the shared graph. As another example, a data-serving workload such as Apache HTTPD, creates many containers, each one with one process. Each process serves a different incoming request. In both examples, the containers share many pages.

B. Address Translation in x86 Linux

Figure 1 shows a conventional TLB organization. Each entry includes a Valid bit (V), a Virtual Page Number (VPN), a Physical Page Number (PPN), flags, and a Process Context Identifier (PCID). The PCID identifies the process, and is shorter than the pid that the OS assigned to the process. For an access to hit in the TLB, both VPN and PCID have to match. In modern processors, a core typically has an L1 instruction TLB, an L1 data TLB, and a unified L2 TLB.

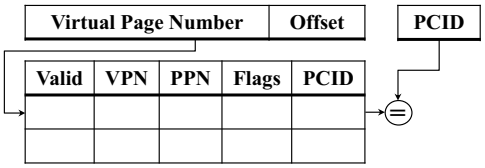


Fig. 1: Conventional TLB organization.

When an access misses in both L1 and L2 TLBs, a page table walk begins. This is a multistep process performed in hardware. Figure 2 shows the page walk for an address in the x86-64 architecture. The hardware reads the CR3 control register, which contains the physical address of the Page Global Directory (PGD) of the currently-running process. The hardware adds the 40-bit CR3 register to bits 47-39 of the virtual address. The result is the physical address of an entry in the PGD. The hardware reads such address from the memory hierarchy — accessing first the data caches and, if they declare a miss, the main memory. The data in that address contains the physical address of the Page Upper Directory (PUD), which is the next level of the translation. Such physical address is then added to bits 38-30 of the virtual address. The contents of the resulting address is the physical address of the next-level table, the Page Middle Directory (PMD). The process is repeated using bits 29-21 of the virtual address to reach the next table, the Page Table (PTE). In this table, using bits 20-12 of the virtual address, the hardware obtains the target physical table entry (pte_t). The pte_t provides the PPN and additional flags that the hardware uploads into the L1 TLB to proceed with the translation of the virtual address.

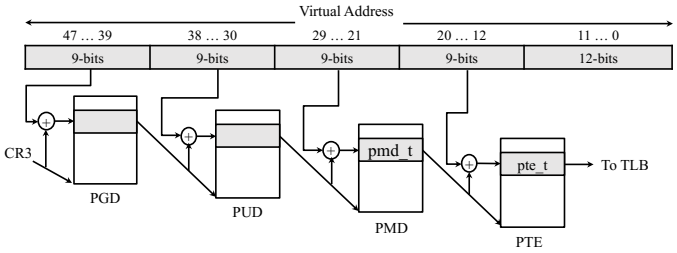


Fig. 2: Page table walk.

In theory, a page walk involves four cache hierarchy accesses. In practice, each core has a translation cache called the Page Walk Cache (PWC) that stores a few recently-accessed entries of the first three tables (PGD, PUD, and PMD). The hardware checks the PWC before going to the cache hierarchy. If it hits there, it avoids a cache hierarchy access.

When this translation process fails, a page fault occurs and the OS is invoked. There are two relevant types of page faults: major and minor. A major one occurs when the page for one of these physical addresses requested during the walk is not in memory. In this case, the OS fetches the page from disk into memory and resumes the translation. A minor page fault occurs when the page is in memory, but the corresponding entry in the tables says that the page is not present in memory. In this case, the OS simply marks the entry as present, and resumes the translation. This happens, for example, when multiple processes share the same physical page. Even though the physical page is present in memory, a new process incurs a minor page fault on its first access to the page.

C. Replicated Translations

The Linux kernel avoids having multiple copies of the same physical page in memory. For example, when a library is shared among applications, only a single copy of the library is brought into physical memory. As a result, multiple processes may point to the same PPN. The VPNs of the different processes may be the same, and have identical permission bits. Furthermore, on a fork operation, pages are only copied lazily and, therefore, potentially many VPNs in the parent and child are the same and point to the same PPNs. Finally, file-backed mappings created through *mmap* lead to further sharing of physical pages. In all of these cases, there are multiple copies of the same {VPN, PPN} translation with the same permission bits, in the TLB (tagged with different PCIDs) and across the page tables of different processes.

III. BabelFish Design

BabelFish has two parts. One enables TLB entry sharing, and the other page table entry sharing. In the following, we describe each part in turn, and then present a simple example.

A. Enabling TLB Entry Sharing

Current TLBs may contain multiple entries with the same {VPN, PPN} pair, the same permission bits, and tagged with different PCIDs. Such replication is common in containerized environments, and can lead to TLB thrashing. To solve this problem, BabelFish combines these entries into a single one with the use of a new identifier called *Container Context Identifier* (CCID). All of the containers created by a user for the same application are given the same CCID. It is expected that the processes within the same CCID group will want to share many TLB entries and page table entries.

BabelFish adds a CCID field to each entry in the TLB. Further, when the OS schedules a process, the OS loads the process' CCID into a register — like it currently does for the process' PCID. Later, when the TLB is accessed, the hardware will look for an entry with a matching VPN tag and a matching CCID. If such an entry is found, the translation succeeds, and the corresponding PPN is read. Figure 3 shows an example for a two-way set-associative TLB. This support allows all the processes in the same CCID group to share entries.

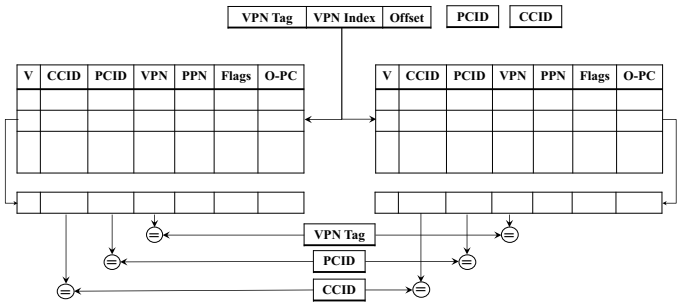


Fig. 3: Two-way set-associative BabelFish TLB.

The processes of a CCID group may not want to share some pages. In this case, a given VPN should translate to different PPNs for different processes. To support this case, we retain the PCID in the TLB, and add an *Ownership* (O) bit in the TLB. If O is set, it indicates that this page is owned rather than shared, and a TLB hit *also* requires a PCID match.

We also want to support the more advanced case where many of the processes of the CCID group want to share the same $\{VPN_0, PPN_0\}$ translation, but a few other processes do not, and have made their own private copies. For example, one process created $\{VPN_0, PPN_1\}$ and another one created $\{VPN_0, PPN_2\}$. This situation occurs when a few of the processes in the CCID group have written to a Copy-on-Write (CoW) page and have their own private copy of the page, while most of the other processes still share the original clean page. To support this case, we integrate the Ownership bit into a new TLB field called *Ownership-PrivateCopy* (O-PC) (Figure 3).

Ownership-PrivateCopy Field.

The O-PC field is expanded in Figure 4. It contains a 32-bit *PrivateCopy (PC)* bitmask, one bit that is the logic OR of all the bits in the PC bitmask ($ORPC$), and the Ownership (O) bit. The PC bitmask has a bit set for each process in the CCID group that has its own private copy of this page. The rest of the processes in the CCID group, *which can be an unlimited number*, still share the clean shared page. We limit the number of private copies to 32 to keep the storage modest. Before we describe how BabelFish assigns bits to processes, we describe how the complete TLB translation in BabelFish works.

The BabelFish TLB is indexed as a regular TLB, using the VPN Tag. The hardware looks for a match in the VPN and in the CCID. All of the potentially-matching TLB entries will be in the same TLB set, and more than one match may occur. On a match, the O-PC and PCID fields are checked, and two cases are possible. First, if the O bit is set, this is a private entry. Hence, the entry can be used only if the process' PCID matches the TLB entry's PCID field.

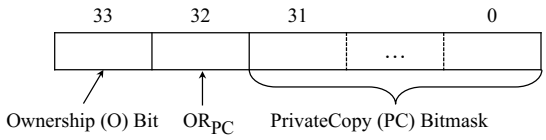


Fig. 4: Ownership-PrivateCopy (O-PC) field. The PrivateCopy (PC) bitmask has a bit set for each process in the CCID group that has its own private copy of the page. The OR_{PC} bit is the logic OR of all the bits in the PC bitmask.

Alternately, if O is clear, this is a shared entry. In this case, before the process can use it, the process needs to check whether the process itself has its own private copy of the page. To do so, the process checks its own bit in the PC bitmask. If the bit is set, the process cannot use this translation because the process already has its own private copy of the page. An entry for such page may or may not exist in the TLB. Otherwise, since the process' bit in the PC bitmask is clear, the process can use this translation.

The O-PC information of a page is part of a TLB entry, but only the O and OR_{PC} bits are stored in the page table entry. The PC bitmask is *not* stored in the page table entry to avoid changing the data layout of the page tables. Instead, it is stored in an OS software structure called the *MaskPage* that is described in the Appendix. Each MaskPage also includes an ordered list (*pid_list*) of up to 32 pids of processes from the CCID group. The order of the pids in this list encodes the mapping of PC bitmask bits to processes. For example, the second pid in the pid_list is the process that is assigned the second bit in the PC bitmask.

In BabelFish, a MaskPage contains the PC bitmasks and pid_list for all the pages of a CCID group mapped by a set of PMD tables (details in the Appendix).

Actions on a Write to a Copy-on-Write (CoW) Page.

To understand the operation of the pid_list, consider what happens when a process writes to a CoW page. The OS checks whether the process is already in the pid_list in the MaskPage for this PMD table set. If it is not, this is the process' first CoW event in this MaskPage. In this case, the OS performs a set of actions. Specifically, it adds the process' pid to the end of the pid_list in the MaskPage, effectively assigning the next bit in the corresponding PC bitmask to the process. This assignment will be used by the process in the future, to know which bit in the PC bitmask to check when it accesses the TLB. In addition, the OS sets that PC bitmask bit in the MaskPage to 1. Then, the OS makes a copy of a page of 512 *pte_t* translations for the process, sets the Ownership (O) bit for each translation, allocates a physical page for the single page updated by the process, and changes the translation for that single page to point to the allocated physical page. The other 511 pages will be allocated later on demand as needed. We choose to copy a page of 512 translations rather than only one translation to reduce the bookkeeping overhead.

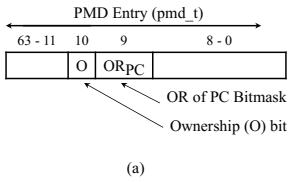
In addition, irrespective of whether this was the process' first CoW event in this MaskPage, the OS has to ensure that the TLB is consistent. Hence, similar to a conventional CoW, the OS invalidates from the local and remote TLBs, the TLB entry for this VPN *that has the O bit equal to zero*. The reason is that this entry has a stale PC bitmask. Note that only this single entry needs to be invalidated, while the remaining (up to 511) translations in the same PTE table can still safely remain in the TLBs. Finally, when the OS gives control back to the writing process, the latter will re-issue the request, which will miss in the TLB and bring its new *pte_t* entry into the TLB, with the O bit set and the updated PC bitmask.

Writable pages (e.g., data set) and read-only pages (e.g., library code) can have an unlimited number of sharers. However, CoW pages, which can be read-shared by an unlimited number of sharers, cannot have more than 32 unique writing processes — since the PC bitmask runs out of space. We discuss the case when the number of writers goes past 32 in the Appendix.

Overall, with this support, BabelFish allows multiple processes in the same CCID group to share the TLB entry for a page — even after other processes in the group have created their own private copies of the page. This capability reduces TLB pressure. This mechanism works for both regular pages and huge pages.

Role of the OR_{PC} Bit in the O-PC Field.

Checking the PC bitmask bits on a TLB hit adds overhead. The same is true for loading the PC bitmask bits into the TLB on a TLB miss. To reduce these overheads, BabelFish uses the OR_{PC} bit (Figure 4), which is the logic OR of all the PC bitmask bits. This bit is present in the O-PC field of a TLB entry. It is also present, together with the O bit, in each *pmd_t* entry of the PMD table. Specifically, bits O and OR_{PC} use the currently-unused bits 10 and 9 of *pmd_t* in the x86 Linux implementation [34, 45] (Figure 5(a)).



(a)

O bit	OR$_{PC}$ bit	Check PC bitmask on TLB hit?	Fetch PC bitmask on TLB miss?
0	0	No	No
0	1	Yes	Yes
1	X	No	No

(b)

Fig. 5: OR_{PC} bit: position in the *pmd_t* (a) and impact (b).

The OR_{PC} bit is used to selectively avoid reading the PC bitmask on a TLB hit, and to avoid loading the PC bitmask to the TLB on a TLB miss. The logic is shown in Figure 5(b). Specifically, consider the case when the O bit is clear. Then, if the OR_{PC} bit is clear, the two operations above can be safely skipped; if OR_{PC} is set, they need to be performed. Consider now the case when the O bit is set. In this case, the two operations can also be skipped. The reason is that an access to a TLB entry with the O bit set relies only on the PCID field to decide on whether there is a match. In all cases, when the PC bitmask bits are not loaded into the TLB, the hardware clears the corresponding TLB storage. Overall, with OR_{PC}, the two operations are skipped most of the time.

Rationale for Supporting CoW Sharing

Supporting CoW sharing within a CCID group, where a page is read-shared by a potentially unlimited number of processes, and a few other processes have private copies of the page adds complexity to the design. However, it is important to support this feature because it assists in accelerating container bring-up — and fast bring-up is a critical requirement for FaaS environments. Specifically, during bring-up, containers first read several pages shared by other containers. Then, they write to some of them. This process occurs gradually. At any point, there are some containers in the group that share the page read-only, and others that have created their own copy. Different containers may end-up writing different sets of pages. Hence, not all containers end-up with a private copy of the page.

B. Enabling Page Table Entry Sharing

In current systems, two processes that have the same {VPN, PPN} mapping and permission bits still need to keep separate page table entries. This situation is common in containerized environments, where the processes in a CCID group may share many pages (e.g., a large library) using the same {VPN, PPN} mappings. Keeping separate page table entries has two costs. First, the many *pte_t* requested from memory could thrash the cache hierarchy [50]. Second, every single process in the group that accesses the page may suffer a minor page fault, rather than only one process suffering a fault. Page fault latency has been shown to add significant overhead [15].

To solve this problem, BabelFish changes the page table structures so that processes with the same CCID can share one or more levels of the page tables. In the most common case, multiple processes will share the table in the last level of the translation. This is shown in Figure 6. The figure shows the translation of an address for two processes in the same CCID group that map it to the same physical address. The two processes (one with $CR3_0$ and the other with $CR3_1$) use the same last level page (PTE). They place in the corresponding entries of their previous tables (PMD) the base address of the same PTE table. Now, both processes together suffer only one minor page fault (rather than two), and reuse the cache line that contains the target *pte_t*.

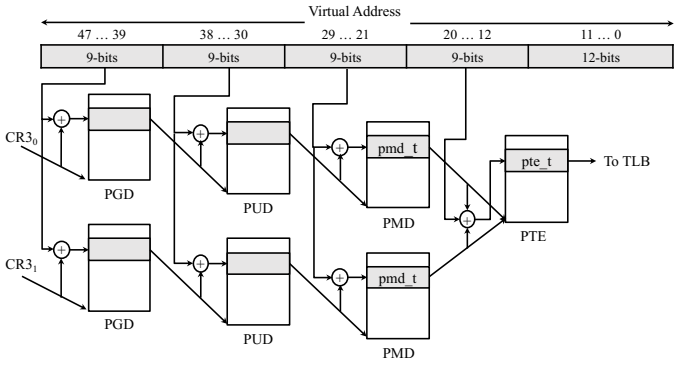


Fig. 6: Page table sharing in BabelFish.

The default sharing level in BabelFish is a PTE table, which maps 512 4KB pages in x86-64. Sharing can also occur at other levels. For example, it can occur at the PMD level—i.e., entries in multiple PUD tables point to the base of the same PMD table. In this case, multiple processes can share the mapping of 512×512 4KB pages or 512 2MB huge pages. Further, processes can share a PUD table, in which case they can share even more mappings. We always keep the first level of the tables (PGD) private to the process.

Examples of large chunks of shared pages are libraries, and data accessed through mounting directories or memory mapping of files. Note that Figure 6 does not imply that all the pages in the shared region are present in memory at the time of sharing; some may be missing. However, it is not possible for two processes to share a table and want to keep private some of pages mapped by the table.

C. Example of BabelFish Operation

To understand the impact of BabelFish, we describe an example. Consider three containers (*A*, *B*, and *C*) that have the same $\{VPN_0, PPN_0\}$ translation. First, *A* runs on Core *0*, then *B* runs on Core *1*, and then *C* runs on Core *0*. Figure 7 shows the timeline of the translation process, as each container, in order, accesses VPN_0 for the first time. The top three rows of the figure correspond to a conventional architecture, and the lower three to BabelFish. To save space, we show the timelines of the three containers on top of each other; in reality, they take place in sequence.

We assume that PPN_0 is in memory but not yet marked as present in memory in any of the *A*, *B*, or *C pte_t*s. We also assume that none of these translations is currently cached in the page walk cache (PWC) of any core.

Conventional Architecture. The top three rows of Figure 7 show the conventional process. As container *A* accesses VPN_0, the translation misses in the L1 and L2 TLBs, and in the PWC. Then, the page walk requires a memory access for each level of the page table (we assume that, once the PWC has missed, it will not be accessed again in this page walk). First, as the entry in the PGD is accessed, the page walker issues a cache hierarchy request. The request misses in the L2 and L3 caches and hits in main memory. The location is read from memory. Then, the entry in the PUD is accessed. The process repeats for every level, until the entry in the PTE is accessed. Since we assume that PPN_0 is in memory but not marked as present, *A* suffers a minor page fault as it completes the translation (Figure 7). Finally, *A*'s page table is updated and a $\{VPN_0, PPN_0\}$ translation is loaded into the TLB.

After that, container *B* running on another core accesses VPN_0. The hardware and OS follow exactly the same process as for *A*. At the end, *B*'s page table is updated and a $\{VPN_0, PPN_0\}$ translation is loaded into the TLB.

Finally, container *C* running on the same core as *A* accesses VPN_0. Again, the hardware and OS follow exactly the same process. *C*'s page table is updated, and another $\{VPN_0, PPN_0\}$ translation is loaded into the TLB. The system does not take

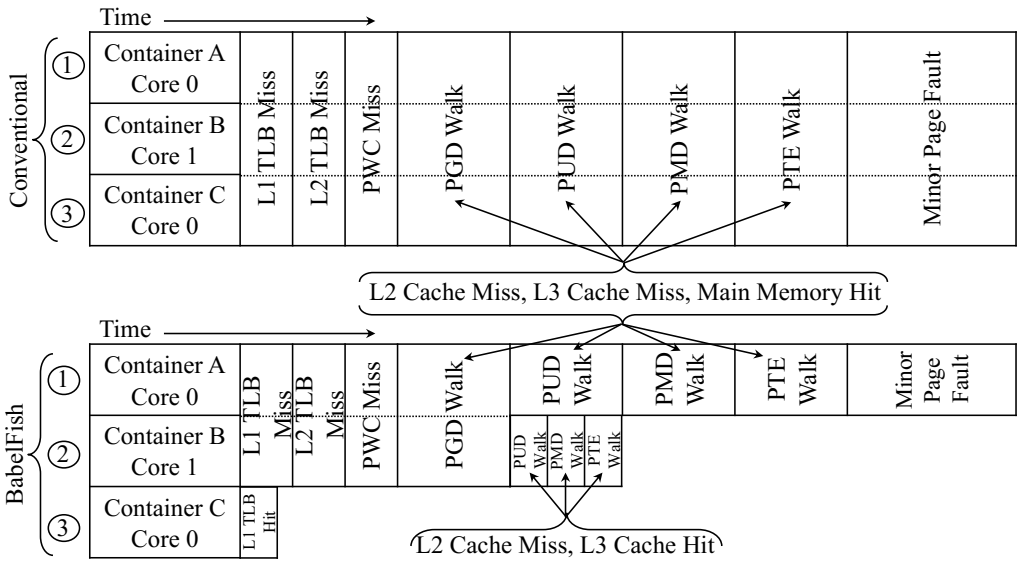


Fig. 7: Timeline of the translation process in a conventional (top) and BabelFish (bottom) architecture. In the figure, container *A* runs on Core *0*, then container *B* on Core *1*, and then container *C* on Core *0*.

advantage of the state that *A* loaded into the TLB, PWC, or caches because the state was for a different process.

BabelFish Architecture. The lower three rows of Figure 7 show the behavior of BabelFish. Container *A*'s access follows the same translation steps as in the conventional architecture. After that, container *B* running on another core is able to perform the translation substantially faster. Specifically, its access still misses in the TLBs and in the PWC; this is because these are per-core structures. However, during the page walk, the multiple requests issued to the cache hierarchy miss in the local L2 but hit in the shared L3 (except for the PGD access). This is because BabelFish enables container *B* to reuse the page-table entries of container *A* — at any level except at the PGD level. Also, container *B* does not suffer any page fault.

Finally, as *C* runs on the same core as *A*, it performs a very fast translation. It hits in the TLB because it can reuse the TLB translation that container *A* brought into the TLB. Recall that, in the x86 architecture, writes to CR3 do not flush the TLB. This example highlights the benefits in a scenario where multiple containers are co-scheduled on the same physical core, either in SMT mode, or due to an over-subscribed system.

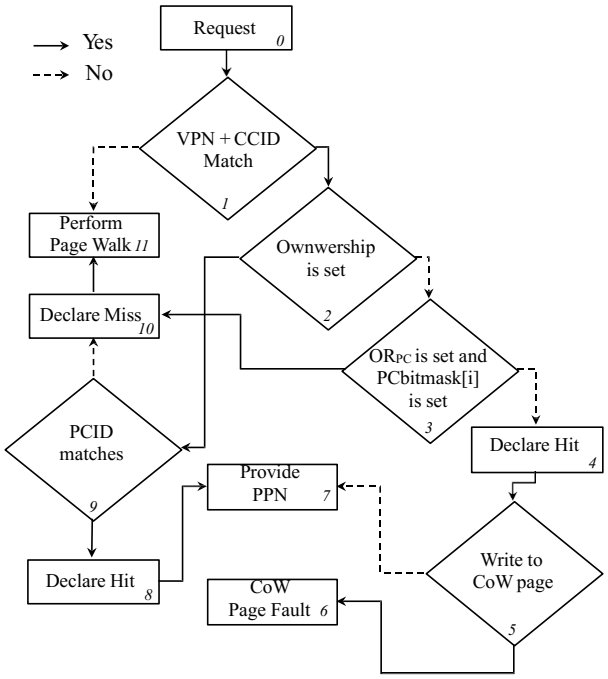


Fig. 8: Flowchart of a TLB access.

IV. Implementation

We now discuss the implementation of BabelFish.

A. Resolving a TLB Access

Figure 8 shows the algorithm that the hardware uses in a TLB access. For simplicity, the figure shows the flowchart assuming a single TLB level; in practice, the checks are performed sequentially in the L1 and L2 TLBs.

As the TLB is accessed, each way of the TLB checks for a VPN and CCID match (①). If none of the ways matches, a page walk is initiated (⑪). Otherwise, for each of the matching ways, the following process occurs. The hardware first checks if the Ownership bit is set (②). If it is, the hardware checks for a PCID match (⑨). If the PCID matches (and assuming that all the permissions checks pass) we declare a TLB hit (⑧) and provide the PPN (⑦). Otherwise, it is a TLB miss (⑩).

If the Ownership bit is clear, the hardware checks if the requesting process already has a private copy of the page. It does so by checking first the OR_{PC} bit and, if it is set, checking the bit of the process in the PC bitmask (③). If both are set, it means that the process has a private copy, and a miss is declared (⑩). Otherwise, a hit is declared (④). In this case, the hardware checks whether this is a write to a CoW page

(⑤). If it is, a CoW page fault is declared (⑥). Otherwise, assuming that all the permissions checks pass, the PPN is provided (⑦). After all the TLB ways terminate their checks, if no hit has been declared (⑩), a page walk is initiated (⑪).

B. Implementing Shared Page Table Entries

The page walker of current systems uses the contents of a control register to initiate the walk. In x86, it is the CR3 register. CR3 points to the beginning of the PGD table (Figure 6), and is unique per process. To minimize OS changes, BabelFish does not change CR3 and, therefore, does not support sharing PGD tables. This is not a practical limitation because processes rarely share the whole range of mapped pages.

BabelFish adds counters to record the number of processes currently sharing pages. One counter is assigned to each table at the translation level where sharing occurs. For example, if sharing occurs at the PTE level like in Figure 6, then there is one counter logically associated with each PTE table. When the last sharer of the table terminates or removes its pointer to the table, the counter reaches zero, and the OS can unmap the table. Such counters do not take much space or overhead to update, and are part of the virtual memory metadata.

C. Comparing BabelFish and Huge Pages

Both BabelFish and huge pages attempt to reduce TLB pressure, and they use orthogonal means. Huge pages merge the translations of multiple pages belonging to *the same* process to create a huge page; BabelFish merges the translations of *different* processes to eliminate unnecessary replication. As a result, BabelFish and huge pages are complementary techniques that can be used together.

If an application uses huge pages, BabelFish automatically tries to combine huge-page translations that have identical {VPN, PPN} pairs and permission bits. Specifically, if the application uses 2MB huge pages, BabelFish automatically tries to merge PMD tables; if the application uses 1GB huge pages, BabelFish automatically tries to merge PUD tables.

D. Supporting ASLR in BabelFish

Address Space Layout Randomization (ASLR) is a security mechanism that randomizes the positions of the segments of a process in virtual memory [47, 58, 48]. When a process is created, the kernel generates a random virtual address (VA) offset for each segment of the process, and adds it to the base address of the corresponding segment. Hence, the process obtains a unique segment layout, which remains fixed for the process lifetime. This strategy thwarts attackers that attempt to learn a program's segment layout. In Linux, a process has 7 segments, including code, data, stack, heap, and libraries.

BabelFish supports ASLR, even while sharing translations between processes. We envision two alternative configurations for ASLR, a software-only solution called *ASLR-SW* that requires minimal OS changes, and a hardware-software solution called *ASLR-HW*, that provides stronger security guarantees.

In the *ASLR-SW* configuration, each CCID group has a private ASLR seed and, therefore, gets its own layout randomization. All processes in the same CCID group get the same layout and, therefore, can share TLB and page table entries among themselves. In all cases, different CCID groups have different ASLR seeds.

This configuration is easy to support in Linux. Specifically, the first container in a CCID gets the offsets for its segments, and subsequent containers in the group reuse them. This configuration is likely sufficient for most deployments, especially in serverless environments where groups of containers are spawned and destroyed quickly.

In the *ASLR-HW* configuration, each process has a private ASLR seed and, therefore, its own layout randomization. In this configuration, when a CCID group is created, the kernel generates a randomized VA offset for each of its segments, as indicated above. We call this set of offsets *CCID_offset[]*. Every time that a process *i* is spawned and joins the CCID group, in addition to getting its own set of random VA offsets for its segments (i_offset[]), it also stores the set of *differences* between the CCID group's offsets and its own offsets (i.e., diff_i_offset[] = CCID_offset[] - i_offset[]).

With this support, when a process is about to access the TLB, its VA goes through a logic module with comparators and one adder. This logic module determines which segment is being accessed, and then adds the corresponding entry in diff_i_offset[] to the VA being accessed. The result is the corresponding VA shared by the CCID group. The TLB is accessed with this address, enabling the sharing of translations between same-CCID processes while retaining per-process ASLR. Similarly, software page walks follow the same steps.

The hardware required by this configuration may affect the critical path of an L1 TLB access. Consequently, BabelFish places the logic module in between the L1 TLB and L2 TLB. The result is that BabelFish's translation sharing is only supported from the L2 TLB down; the L1 TLB does not support TLB entry sharing.

In practice, eliminating translation sharing from the L1 TLB only has a minor performance impact. The reason is that the vast majority of the translations are cached in the L2 TLB and, therefore, page walks are still eliminated. The L1 TLB performs well as long are there is locality of accesses within a process, which *ASLR-HW* does not alter.

To be conservative, in our evaluation of Section VII, we model BabelFish with *ASLR-HW* by default.

V. Security Considerations

To minimize vulnerabilities, cloud providers limit page sharing to occur only within a single user security domain. For example, VMware only allows page deduplication within a single guest VM [72]. In Kubernetes, the security domain is called a Pod [29], and only containers or processes within a Pod share pages by default. In the recently-proposed X-Containers [68], the security domain is the shared LibOS, within which all processes share pages.

In this paper, we consider a more conservative container environment, where a security domain contains only the containers of a single user that are running the same application. It is on top of this baseline environment that BabelFish proposes

that the containers in the security domain additionally share address translations.

The baseline environment, where all the containers of a single user running the same application share pages is likely vulnerable to side channel attacks. Adding page translation sharing with BabelFish does not significantly change the security considerations over the baseline environment. This is because the attacker could leverage the sharing of pages to attack, without needing to leverage the sharing of translations. Addressing the issue of securing the baseline environment is beyond the scope of this paper.

VI. Evaluation Methodology

Modeled Architecture. We use cycle-level simulations to model a server architecture with 8 cores and 32GB of main memory. The architecture parameters are shown in Table I. Each core is out-of-order and has private L1 I+D caches, a private unified L2 cache, and a shared L3 cache. Each core has L1 I+D TLBs, a unified L2 TLB, and a page walk cache with a page walker. The table shows that the TLBs can hold pages of different sizes at the same time. With BabelFish, the access times of the L1 TLBs do not change. However, on an L1 TLB miss, BabelFish performs a two-cycle address transformation for ASLR (Section IV-D). Moreover, the L2 TLB has two access times: 10 and 12 cycles. The short one occurs when the O and OR_{PC} bits preempt an access to the PC bitmask (Figure 5(b)); the long one occurs when the PC bitmask is accessed. Section VII-D provides details. We use Ubuntu Server 16.04 [13] and Docker 17.06 [18].

Processor Parameters	
Multicore chip	8 2-issue OoO cores, 128 ROB; 2GHz
L1 (D, I) cache	32KB, 8 way, WB, 2 cycle AT, 16 MSHRs, 64B line
L2 cache	256KB, 8 way, WB, 8 cycle AT, 16 MSHRs, 64B line
L3 cache	8MB, 16 way, WB, shared, 32 cycle AT, 128 MSHRs, 64B line
Per Core MMU Parameters	
L1 (D, I)TLB (4KB pages)	64 entries, 4 way, 1 cycle AT
L1 (D)TLB (2MB pages)	32 entries, 4 way, 1 cycle AT
L1 (D)TLB (1GB pages)	4 entries, FA, 1 cycle AT
ASLR Transformation	2 cycles on L1 TLB miss
L2 TLB (4KB pages)	1536 entries, 12 way, 10 or 12 cyc AT
L2 TLB (2MB pages)	1536 entries, 12 way, 10 or 12 cyc AT
L2 TLB (1GB pages)	16 entries, 4 way, 10 or 12 cycle AT
Page walk cache	16 entries/level, 4 way, 1 cycle AT
Main Memory Parameters	
Capacity; Channels	32GB; 2
Ranks/Channel; Banks/Rank	8; 8
Frequency; Data rate	1GHz; DDR
Host and Docker Parameters	
Scheduling quantum	10ms
PC bitmask; PCID; CCID	32 bits; 12 bits; 12 bits

TABLE I: Architectural parameters. AT is Access Time.

Modeling Infrastructure. We integrate the Simics [49] full-system simulator with the SST framework [62, 9] and the DRAMSim2 [63] memory simulator. Additionally, we utilize Intel SAE [14] on Simics for OS instrumentation. We use CACTI [10] for energy and access time evaluation. For the address translation, each hardware page walker is connected to the cache hierarchy and issues memory requests following the page walk semantics of x86-64 [34]. The Simics infrastructure provides the actual memory and control register contents for each memory access of the page walk. We use the page tables maintained by the Linux kernel during full-system simulation. To evaluate the hardware structures of BabelFish, we model them in detail in SST. To evaluate the software structures, we modify the Linux kernel and instrument the page fault handler.

Workloads. We use three types of workloads: three Data Serving applications, two Compute applications, and three Functions representing Function-as-a-Service (FaaS).

The *Data Serving* applications are the containerized *ArangoDB* [7], *MongoDB* [55], and *HTTPd* [5]. *ArangoDB* represents a key-value store NoSQL database with RocksDB as the storage engine. *MongoDB* is a scalable document-model NoSQL database with a memory mapped engine, useful as a backend for data analytics. *HTTPd* is an efficient open source HTTP server with multiprocess scaling, used for websites and online services. Each application is driven by the Yahoo Cloud Serving Benchmark [16] with a 500MB dataset.

The *Compute* applications are the containerized *GraphChi* [42] and *FIO* [36]. *GraphChi* is a graph processing framework with memory caching. We use the PageRank algorithm which traverses a 500MB graph from SNAP [43]. *FIO* is a flexible I/O benchmarking application that performs in-memory operations on a randomly generated 500MB dataset.

We developed three C/C++ containerized *Functions*: *Parse*, which parses an input string into tokens, a *Hash* function based on the djb2 algorithm [35], and a *Marshal* function that transforms an input string to an integer. All functions are based on OpenFaaS [24] and use the GCC image from Docker Hub [19]. Each function operates on an input dataset similar to [2]. For these functions, we explore dense and sparse inputs. In both cases, a function performs the same work; we only change the distance between one accessed element and the next. In dense, we access all the data in a page before moving to the next page; in sparse, we access about 10% of a page before moving to the next one.

Configurations Evaluated. We model widely-used container environments that exploit replication of the applications for better load balancing and fault tolerance. We conservatively keep the number of containers per core low. Specifically, in Data Serving and Compute workloads, each core is multiplexed between two containers, which run the same application on different input data. Each Data Serving container is driven by a distinct YCSB client and, therefore, serves different requests. Similarly, each Compute container accesses different random locations. As a result, in both types of workloads, each container accesses different data, but there is partial overlap in the data pages accessed by the two containers.

In the Function workloads, each core is multiplexed between three containers, each running a different function. The three containers access different data, but there is partial overlap in

the data pages accessed by the three containers.

In each case, we compare two configurations: a conventional server (*Baseline*), and one augmented with the proposed hardware and software (*BabelFish*). We enable transparent huge pages (THP) for both the *Baseline* and *BabelFish*.

Simulation Methodology. BabelFish proposes software and hardware changes. To model a realistic system, we need to warm-up both the OS and the architecture state. We use two phases. In the first phase, we warm-up the OS by running the containers for Data Serving and Compute for a minute, and all the functions to completion, as they are short.

In the second phase, we bring the applications to steady state, warm-up the architectural state, and measure. Specifically, for Data Serving and Compute, we instrument the applications to track entry to steady state and then execute for 10 seconds. We then warm-up the architectural state by running 500 million instructions, and finally evaluate for four billion instructions. For Functions, there is no warm-up. We run all the three functions from the beginning to completion and measure their execution time.

We also measure container bring-up, as the time to start a Function container from a pre-created image (docker start). We perform full system simulation of all the above steps.

VII. Evaluation

The BabelFish performance improvements come from two sources: page table entry sharing and L2 TLB entry sharing. In this section, we first characterize these two sources, and then discuss the overall performance improvements. Finally, we examine the resources that BabelFish needs.

A. Characterizing Page Table Entry Sharing

Figure 9 shows the shareability of PTE entries (*pte_ts*) when running two containers of the same Data Serving and Compute application, or three containers of the functions. The figure includes the steady-state mappings of all pages, including container infrastructure, and program code and data. The data is obtained by native measurements on a server using Linux Pagemap [46], while running each application for 5 minutes.

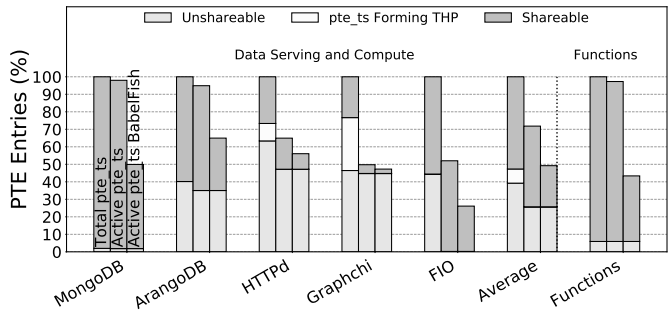


Fig. 9: Page table sharing characterization.

For each application, there are three bars, which are normalized to the leftmost one. The leftmost one is the total number of *pte_ts* mapped by the containers. The central bar is the number of *Active pte_ts*, namely those that are placed by the kernel in the active LRU list. They are a proxy for the pages that are recently touched. The rightmost bar is the number of Active *pte_ts* after enabling BabelFish.

Each bar is broken down into *pte_ts* that are shareable, *pte_ts* that are unshareable, and *pte_ts* that correspond to the huge pages created by THP. The latter are also unshareable. A shareable *pte_ts* has an identical {VPN, PPN} pair and permission bits as another. Unshareable *pte_ts* are either exclusive to a process or not shareable.

Data Serving and Compute Applications. If we consider the average bars, we see that 53% of the total baseline *pte_ts* are shareable. Further, most of them are active. BabelFish reduces the number of shareable active *pte_ts* by about half. Since this plot corresponds to only two containers, the reduction in shareable active *pte_ts* is at most half. Sharing *pte_ts* across more containers would linearly increase savings, as only one copy is required for all the sharers. Overall, the average reduction in total active *pte_ts* attained by BabelFish is 30%.

The variation of shareability across applications is primarily due to the differences in usage of shared data versus internal buffering. For example, GraphChi operates on shared vertices, but uses internal buffering for the edges. As a result, most of the active *pte_ts* are unshareable, and we see little gains. In contrast, MongoDB and FIO operate mostly on shared data and see substantial *pte_ts* savings. The other applications have a more balanced mixture of shareable and unshareable *pte_ts*.

Impact of Huge Pages. As shown in Figure 9, THP *pte_ts* are on average 8% of the total *pte_ts*, and are in the unshareable portion. Moreover, only a negligible number of these entries are active during execution. To understand this data, note that MongoDB and ArangoDB recommend disabling huge pages [6, 54]. Further, THP supports only anonymous mappings, and not file-backed memory mapping. The anonymous mappings are commonly used for internal buffering, which are unshareable. Therefore, huge pages are rarely active.

Functions. Functions have a very high percentage of shareable *pte_ts*. Combining them reduces the total active *pte_ts* by 57%. The unshareable *pte_ts* correspond to the function code and internal buffering, which are unique for each function. They account for only ≈6% of *pte_ts*.

One can subdivide the shareable category in the bars into application shared data and infrastructure pages. The latter contain the common libraries that are required by all the functions. It can be shown that they are 90% of the total shareable *pte_ts*, and can be shared across functions. Data *pte_ts* are few, but also shareable across functions.

B. Characterizing TLB Entry Sharing

Figure 10a shows the reduction in L2 TLB Misses Per Kilo Instructions (MPKI) attained by BabelFish. Note that in our evaluation, we conservatively do not share translations at the L1 TLB (Section IV-D). The figure is organized based on workload, and shows data and instruction entries separately.

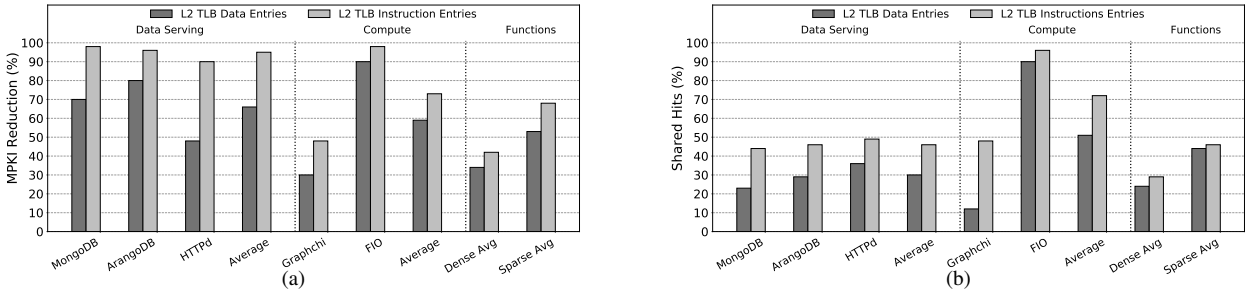


Fig. 10: (a) L2 TLB MPKI reduction attained by BabelFish. (b) Hits on L2 TLB entries that were brought into the TLB by processes other than the one issuing the accesses. We call them Shared Hits and show them as a fraction of all L2 TLB hits.

From the figure, we see that BabelFish reduces the MPKI across the board. For example, for Data Serving, the data MPKI reduces by 66%, and the instruction MPKI reduces even more, by 96%. These are substantial reductions. Good results are also seen for the Compute workloads. Functions see smaller MPKI reductions, as they are short lived and interfered by the docker engine/OS.

The reductions come from various sources. First, a container may reuse the L2 TLB entries of another container and avoid misses. Second, sharing L2 TLB entries effectively increases TLB capacity, which reduces misses. Finally, as a result of these two effects, there is a lower chance that co-scheduled processes evict each other's TLB entries.

To gain more insight into L2 TLB entry sharing, Figure 10b shows the number of hits on L2 TLB entries that were brought into the L2 TLB by processes other than the one performing the accesses. We call them *Shared Hits* and show them as a fraction of all L2 TLB hits. The figure is organized as Figure 10a. As we can see, the percentage of shared hits is generally sizable, but varies across applications, as it is dependent on the applications' access patterns.

For example, GraphChi shows 48% shared hits for instructions and 12% for data. This is because PageRank's code is regular, while its data accesses are fairly random, causing variation between the data pages accessed by the two containers. Overall, BabelFish's TLB entry sharing bolsters TLB utilization and reduces page walks.

C. Latency or Execution Time Reduction

To assess the performance impact of BabelFish, we report different metrics for different applications: reduction in mean and 95th percentile (*Tail*) latency in Data Serving applications; reduction in execution time in Compute applications; and reduction in bring-up time and function execution time in Functions. Figure 11 shows the results, all relative to *Baseline*. To gain further insight, Table II shows what fraction of the performance improvement in Figure 11 comes from TLB effects; the rest comes from page table effects. Recall that transparent huge pages are enabled in all applications but MongoDB and ArangoDB [6, 54].

Consider first the Data Serving applications. On average, BabelFish reduces their mean and tail latencies by a significant

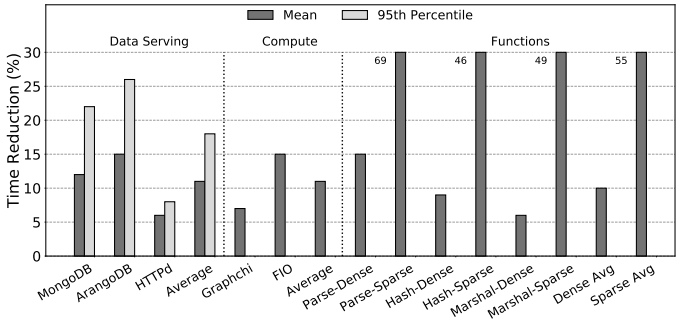


Fig. 11: Latency/time reduction attained by BabelFish.

Data Sharing	Functions
MongoDB: 0.77	Parse-Dense: 0.15
ArangoDB: 0.25	Parse-Sparse: 0.01
HTTPd: 0.81	Hash-Dense: 0.18
Average: 0.61	Hash-Sparse: 0.01
Compute	Marshal-Dense: 0.28
Graphchi: 0.11	Marshal-Sparse: 0.02
FIO: 0.29	Dense Average: 0.20
Average: 0.20	Sparse Average:0.01

TABLE II: Fraction of time reduction due to L2 TLB effects.

11% and 18%, respectively. The reductions are higher in MongoDB and ArangoDB than in HTTPd. It can be shown that this is because, in *Baseline*, address translation induces more stress in the MongoDB and ArangoDB database applications than in the stream-based HTTPd application. Hence, BabelFish is more effective in the former. Table II shows that MongoDB gains more from L2 TLB entry sharing, while ArangoDB more from page table entry sharing. Therefore, both types of entry sharing are helpful.

Compute applications also benefit from BabelFish. On average, their execution time reduces by 11%. GraphChi has lower gains because it performs low-locality accesses in the graph, which makes it hard for one container to bring shared translations that a second container can reuse. On the other hand, FIO has higher gains because its more regular access patterns enable higher shared translation reuse. Table II shows that these applications benefit more from page table effects.

Recall that we run the Functions in groups of three at a time. The leading function behaves similarly in both BabelFish and *Baseline* due to cold start effects. Hence, Figure 11 shows the reduction in execution time for only the other two functions in the group. We see that the reductions are heavily dependent on the access patterns. Functions with dense access patterns access only a few pages, and spend little time in page faults. Hence, their execution time decreases by only 10% on average. In contrast, functions with sparse access patterns access more pages and spend more time servicing page faults. Hence, BabelFish reduces their execution time by 55% on average. In all cases, as shown in Table II, most gains come from page table entry sharing.

Finally, although not shown in any figure, BabelFish speeds-up function bring-up by 8%. Most of the remaining overheads in bring-up are due to the runtime of the Docker engine and the interaction with the kernel. Overall, BabelFish speeds-up applications across the board substantially, even in our conservative environment where we co-locate only 2-3 containers per core.

BabelFish vs Larger TLB.

BabelFish's main hardware requirements are additional bits in the L2 TLB for the CCID and O-PC fields. It could be argued that this extra hardware could be used instead to make a conventional TLB larger. Hence, we have re-run the workloads with a conventional architecture with this larger L2 TLB. On average, the resulting architecture reduces the mean request latency of Data Serving applications by 2.1%, the execution time of Compute applications by 0.6%, and the execution time of Functions by 1.1% (dense) and 0.3% (sparse).

These reductions are much smaller than those attained by BabelFish (Figure 11). One reason is that BabelFish benefits from both L2 TLB and page table effects. A second reason is that BabelFish also benefits from processes prefetching TLB entries for other processes into the L2 TLB and caches. Overall, this larger L2 TLB is not a match for BabelFish.

D. BabelFish Resource Analysis

We analyze the hardware and software resources needed by BabelFish. We also analyze the resources of a design where, as soon as a write occurs on a CoW page, sharing for the corresponding PMD table set immediately stops, and all sharers get private page table entries. This design does not need a PC bitmask.

Hardware Resources. BabelFish's main hardware overhead is the CCID and O-PC fields in the TLB, and the associated comparison logic (Figure 3). We estimate that this extra hardware adds 0.4% to the area of a baseline core (without L2). If we eliminate the need for the PC bitmask bits, the area overhead falls to 0.07%. These are very small numbers.

Table III shows several parameters of the L2 TLB, both for *Baseline* and BabelFish: area, access time, dynamic energy of a read access, and leakage power. The data corresponds to 22nm, and is obtained with CACTI [10]. The table shows that the difference in TLB access time between *Baseline* and BabelFish is a fraction of a cycle. To be conservative, we add two extra cycles to the access time of the BabelFish L2 TLB when the PC bitmask has to be accessed.

Configuration	Area	Access Time	Dyn. Energy	Leak. Power
Baseline	0.030 mm2	327 ps	10.22 pJ	4.16 mW
BabelFish	0.062 mm2	456 ps	21.97 pJ	6.22 mW

TABLE III: Parameters of the L2 TLB at 22nm.

Memory Space. The memory space of BabelFish is minimal. It includes one MaskPage with PC bitmasks and pid_list (Figure 13) for each 512 pages of *pte_ts*. This is 0.19% space overhead. In addition, it includes one 16-bit counter per 512 *pte_ts* to determine when to de-allocate a page of *pte_ts* (Section IV-B). This is 0.048% space overhead. Overall, BabelFish only adds 0.238% space overhead. If we eliminate the need for the PC bitmask bits, the first item goes away, and the total space overhead is 0.048%.

Software Complexity. We implement BabelFish's page table sharing mechanism in the Linux kernel and in the Simics shadow page tables. We require about 300 Lines of Code (LoC) in the MMU module, 200 LoC in the page fault handler, and 800 LoC for page table management operations.

VIII. Related Work

Huge Pages. BabelFish transparently supports huge pages, which are in fact a complementary way to reduce TLB and cache pressure. Recent work has tackled huge-page bloating and fragmentation issues [41, 57].

Translation Management and TLB Design. Recent work [4, 74, 40] aims to reduce the translation coherence overheads with software and hardware techniques. Other work provides very fine grain protection domains and enables sub-page sharing, but does not support translation sharing [73]. CoLT and its extension [60, 59] propose the orthogonal idea of coalesced and clustered TLB entries within the same process.

MIX-TLB [17] supports both huge page and regular page translations in a single structure, increasing efficiency. In [8], self-invalidating TLB entries are proposed to avoid TLB shoot-downs. Shared last-level TLB [11] aims to reduce translation overhead in multi-threaded applications. Recent work [51] proposes to prefetch page table entries on TLB misses. Other work [61, 69] shows optimizations for CPU-GPU environments. These approaches tackle a set of different problems in the translation process and can co-exist with BabelFish.

Elastic cuckoo page tables [71] propose a hashed page table design based on elastic cuckoo hashing. Such scheme can be augmented with an additional hashed page table where containers in a CCID group share page table entries. Auxiliary structures called Cuckoo Walk Tables could be enhanced to indicate whether a translation is shared. Since the TLB is not affected by elastic cuckoo page tables, BabelFish's TLB design remains the same.

Other work has focused on virtualized environments. POM-TLB and CSALT [64, 50] propose large in-memory TLBs and cache partitioning for translations. DVMT [1] proposes

to reduce 2D page walks in virtual machines by enabling application-managed translations. RMM [37] proposes redundant memory mappings and [25] aims to reduce the dimensionality of nested page walks. PageForge [70] proposes near-memory extensions for content-aware page merging. This deduplication process shares pages in virtualized environments, generating an opportunity to further share translations. These solutions are orthogonal to BabelFish in a scenario with containers inside VMs.

Khalidi and Talluri [38] propose a scheme to share TLB entries between processes. The idea is to tag each TLB entry with a PCID or a bitvector. The bitvector uses one-hot encoding to identify one of 10-16 different collections of shared translations. A TLB is accessed twice: with the PCID and with the bitvector. If the bitvector matches, the shared translation is retrieved. The authors also suggest tagging global hashed page tables with PCIDs or bitvectors. While this scheme allows translation sharing, compared to BabelFish, it has several limitations. First, unlike BabelFish, the scheme is not scalable because the number of different collections of translations that can be shared is limited to the number of bits in the bitvector. Second, to find a shared translation, the TLB is accessed twice. Third, unlike BabelFish, the scheme does not support CoW or selective sharing of a translation. In addition, it does not support ASLR. Finally, BabelFish also proposes the sharing of multi-level page table entries.

Native Execution Environment. A software-only approach that improves zygote fork and application performance in Android by sharing page translations across applications is presented in [23]. This scheme only shares code translations of 4KB pages in a 2-level page table design. This solution requires disabling TLB tagging and marking TLB entries as global, which leads to TLB flushes at context switches. Moreover, this solution does not support the sharing of data translations. In contrast, BabelFish shares both code and data translations in both TLB and page tables, for multiple page sizes, while supporting tagged TLBs and CoW.

IX. Conclusion

Container environments create replicated translations that cause high TLB pressure and redundant kernel work during page table management. To address this problem, we proposed *BabelFish*, a novel architecture to share address translations across containers in the L2 TLB and in the page tables. We evaluated BabelFish with simulations of an 8-core processor running a set of Docker containers. On average, BabelFish reduced the mean and tail latency of containerized data-serving workloads by 11% and 18%, respectively. It also lowered the execution time of containerized compute workloads by 11%. Finally, it reduced serverless function bring-up time by 8% and execution time by 10%–55%. Overall, BabelFish sped-up applications across the board substantially, even in our conservative environment where we co-located only 2-3 containers per core.

Acknowledgment

This research was funded in part by NSF under grants CNS 17-63658, CNS 17-05047, and CCF 16-29431.

Appendix: Storing and Accessing the PC Bitmask

To understand where the PC bitmask is stored and how it is accessed, consider Figure 12(a), which shows the page tables of three processes of a CCID group that share a PTE table. BabelFish adds a single *MaskPage* associated with the set of PMD tables of the processes.

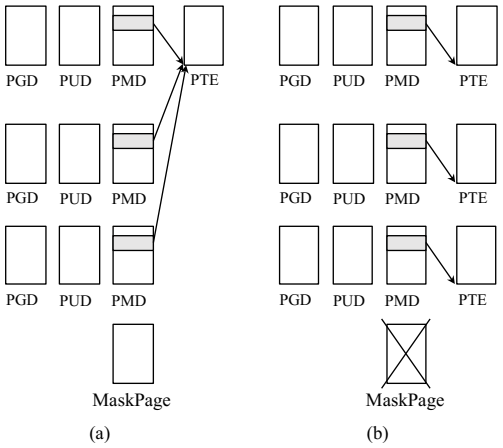


Fig. 12: Operation of the MaskPage.

The OS populates the MaskPage with the PC bitmask and the pid_list information for all the pages mapped by the PMD table set. Figure 13 shows its contents. It contains up to 512 PC bitmasks for the 512 *pmd_t* entries in the PMD table set. Further, it contains a single pid_list, which has the ordered pids of the processes in the CCID group that have performed a CoW on any of the pages mapped by the PMD table set. The pid_list has at most 32 pids. Hence, there can be at most 32 distinct processes performing CoW on these pages.

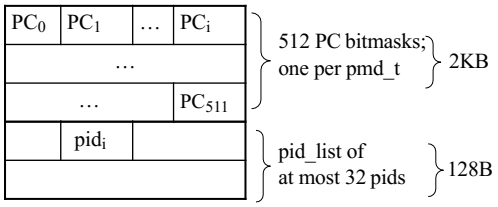


Fig. 13: MaskPage with 512 PC bitmasks and one pid_list.

As a process performs a CoW on a page in this PMD table set for the first time, the OS puts its pid in the next position in the ordered pid_list. If this is position *i*, it means that the process claims bit *i* in all of the 512 PC bitmasks. Of course, bit *i* is set in only the PC bitmasks of the *pmd_t* entries that reach pages that the process has performed a CoW on.

With this design, on a TLB miss, as a *pmd_t* entry is accessed, the hardware checks the OR_{PC} bit (Section III-A). If it is set, then the hardware accesses the MaskPage in parallel

with the request for the *pte_t* entry. The hardware then reads the corresponding PC bitmask and loads it into the L1 TLB.

If more than 32 processes in a CCID group perform CoWs on pages mapped by a PMD table set, this design runs out of space, and all the processes in the group need to revert to non-shared translations — even if many processes in the group share many {VPN, PPN} mappings. Specifically, when a 33rd process performs a CoW, the OS allocates a page of *pte_t* translations for each of the processes in the group that were using shared translations in the PMD page set. In these new translations, the OS sets the Ownership (O) bit. The only physical data page that is allocated is the one updated by the writing process. The result is the organization shown in Figure 12(b), for a single PTE table being shared.

Consolidating CoW information from all the pages mapped by a PMD table set in a single MaskPage may sometimes be inefficient. However, we make this choice because selecting a finer granularity will increase space overhead. Furthermore, recall that writable pages (e.g., dataset) and read-only pages (e.g., code) can have an unlimited number of sharers. CoW pages can also be read-shared by an unlimited number of sharers; they just cannot have more than 32 writing processes. It can be shown that, with an extra indirection, one could support more writing processes.

REFERENCES

[1] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 457–468.

[2] Amazon, "AWS Lambda," https://aws.amazon.com/lambda.

[3] Amazon Web Services, "Amazon Elastic Container Service," https://aws.amazon.com/ecs.

[4] N. Amit, "Optimizing the TLB shootdown algorithm with page access tracking," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[5] Apache Software Foundation, "Apache HTTP Server Project," https://httpd.apache.org.

[6] ArangoDB, "Transparent Huge Pages Warning," https://github.com/arangodb/arangodb/blob/2b84348b7789893878ebd0c8b552dc20416c98f0/lib/ApplicationFeatures/EnvironmentFeature.cpp.

[7] ArangoDB Inc., "ArangoDB," https://www.arangodb.com.

[8] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[9] A. Awad, S. D. Hammond, G. R. Voskuilen, and R. J. Hoekstra, "Samba: A Detailed Memory Management Unit (MMU) for the SST Simulation Framework," Sandia National Laboratories, Tech. Rep. SAND2017-0002, January 2017.

[10] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 14:1–14:25, Jun. 2017.

[11] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 62–63.

[12] B. Burns and D. Oppenheimer, "Design Patterns for Container-based Distributed Systems," in *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*, Denver, CO, USA, Jun. 2016.

[13] Canonical, "Ubuntu Server," https://www.ubuntu.com/server.

[14] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi, "Simulation and Analysis Engine for Scale-Out Workloads," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 22:1–22:13.

[15] J. Choi, J. Kim, and H. Han, "Efficient Memory Mapped File I/O for In-Memory File Systems," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[17] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 435–448.

[18] Docker, "Docker Community Edition," https://github.com/docker/docker-ce.

[19] ——, "Docker Hub," https://hub.docker.com.

[20] ——, "Swarm Mode Overview," https://docs.docker.com/engine/swarm.

[21] ——, "What is Docker?" https://www.docker.com/what-docker.

[22] ——, "Best practices for writing Dockerfiles," https://docs.docker.com/develop/develop-images/dockerfile_best-practices, Tech. Rep., 2019.

[23] X. Dong, S. Dwarkadas, and A. L. Cox, "Shared address translation revisited," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 18:1–18:15.

[24] A. Ellis, "Open Functions-as-a-Service," https://github.com/openfaas/faas.

[25] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*, Cambridge, United Kingdom, Dec. 2014.

[26] Google, "Cloud Functions," https://cloud.google.com/functions/.

[27] ——, "Compute Engine," https://cloud.google.com/compute.

[28] ——, "Production Grade Container Orchestration," https://kubernetes.io.

[29] ——, "Kubernetes pods," https://kubernetes.io/docs/concepts/workloads/pods/pod, Tech. Rep., 2019.

[30] IBM, "Docker at insane scale on IBM Power Systems," https://www.ibm.com/blogs/bluemix/2015/11/docker-insane-scale-on-ibm-power-systems.

[31] ——, "IBM Cloud Computing," https://www.ibm.com/cloud-computing.

[32] IBM Cloud Functions, "Function-as-a-Service on IBM," https://www.ibm.com/cloud/functions.

[33] B. Ibryam, "Principles of Container-based Application Design," https://www.redhat.com/cms/managed-files/cl-cloud-native-container-design-whitepaper-f8808kc-201710-v3-en.pdf, Red Hat, Inc, Tech. Rep., 2017.

[34] Intel, "Developer's Manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[35] B. J. McKenzie, R. Harries, and T. Bell, "Selecting a hashing algorithm." *Software: Practice and Experience*, vol. 20, pp. 209 – 224, 02 1990.

[36] Jens Axboe, "Flexible I/O Tester," https://github.com/axboe/fio.

[37] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant Memory Mappings for Fast Access to Large Memories," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, Portland, Oregon, Jun. 2015.

[38] Y. A. Khalidi and M. Talluri, "Improving the address translation performance of widely shared pages," Sun Microsystems, Inc., Tech. Rep., 1995.

[39] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 427–444. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic

[40] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy Translation Coherence," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 651–664.

[41] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and Efficient Huge Page Management with Ingens," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, 2016.

[42] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 31–46.

[43] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data.

[44] Linux, "Linux Containers," https://en.wikipedia.org/wiki/LXC.

[45] Linux Kernel, "Page Table Types," https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/include/asm/pgtable_types.h?h=v4.19.1.

[46] ——, "Pagemap," https://www.kernel.org/doc/Documentation/vm/pagemap.txt.

[47] Linux Kernel Documentation, "Address Space Layout Randomization," https://www.kernel.org/doc/Documentation/sysctl/kernel.txt.

[48] M. Howard, "Address space layout randomization in Windows Vista," Microsoft Corporation, vol.26, 2006.

[49] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[50] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, "CSALT: Context Switch Aware Large TLB," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 449–462.

[51] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched Address Translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, 2019.

[52] Microsoft, "Azure," https://azure.microsoft.com.

[53] Microsoft, "Azure Functions," https://azure.microsoft.com/en-us/services/functions.

[54] MongoDB, "Transparent Huge Pages Warning," https://github.com/mongodb/mongo/blob/eeea7c2f80bdaf49a197a1c8149d7bc6cbc9395e/src/mongo/db/startup_warnings_mongod.cpp.

[55] MongoDB Inc., "MongoDB," https://www.mongodb.com.

[56] OpenShift Documentantion, "OpenShift Container Platform Cluster Limits," https://docs.openshift.com/container-platform/3.11/scaling_performance/cluster_limits.html, Red Hat, Inc, Tech. Rep., 2019.

[57] A. Panwar, A. Prasad, and K. Gopinath, "Making Huge Pages Actually Useful," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 679–692.

[58] PaX Team, "Address Space Layout Randomization," Phrack, 2003.

[59] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 558–567.

[60] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 258–269.

[61] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 743–758.

[62] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balls, and B. Jacob, "The Structural Simulation Toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 37–42, Mar. 2011.

[63] P. Rosenfeld, E. Cooper-Balls, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[64] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 469–480.

[65] N. Savage, "Going Serverless," *Commun. ACM*, 2018.

[66] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, 2019.

[67] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: ACM, 2016, pp. 1:1–1:13.

[68] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 121–135.

[69] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling Page Table Walks for Irregular GPU Applications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 180–192.

[70] D. Skarlatos, N. S. Kim, and J. Torrellas, "PageForge: A Near-Memory Content-Aware Page-Merging Architecture," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 17, 2017.

[71] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 20, 2020.

[72] VMware, "Security Considerations and Disallowing Inter-Virtual Machine Transparent Page Sharing," "https://kb.vmware.com/s/article/2080735", Tech. Rep., 2018.

[73] E. Witchel, J. Cates, and K. Asanović, "Mondrian Memory Protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 304–316.

[74] Z. Yan, J. Veselý, G. Cox, and A. Bhattacharjee, "Hardware translation coherence for virtualized systems," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 430–443.