BabelFish: Fusing Address Translations for Containers

Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas, *University of Illinois at Urbana-Champaign, Champaign, IL, 61820, USA*

Cloud computing has begun a transformation from using virtual machines to using containers. Containers are attractive because of their "build once, run anywhere" computing model and their minimal performance overhead. Cloud providers leverage the lean nature of containers to run hundreds of them or more on a few cores. Furthermore, containers enable the serverless paradigm, which involves 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 article proposes BabelFish*, a novel architecture to share page translations across containers in the TLB and in page tables.* BabelFish *reduces the mean and tail latency of containerized workloads, cold-start effects of function execution, and container bring-up time. This work also advocates for the need to provide more hardware support for containerized and serverless environments.*

Cloud computing has been undergoing a radical transformation with the emergence of *Containers*.[1] Like a virtual machine (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, Amazon's ECS, IBM's Cloud, and Microsoft's Azure now provide container-based solutions. The most prominent container solution is Docker containers. In addition, there are management frameworks, such as Google's Kubernetes[2] and Facebook's Twine,[3] which automate the deployment, scaling, and maintenance of containerized applications.

Container environments are typically oversubscribed, with many more containers running than cores. Moreover, container technology has laid the foundation for *Serverless* computing,[4] a new cloud computing paradigm provided by services like Amazon's Lambda, Microsoft's Azure Functions, Google's Cloud Functions, and IBM's Cloud Functions. 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.[5,6]

Our detailed analysis of containerized environments reveals that, very often, the same virtual page number (VPN) to physical page number (PPN) translation, with the same permission bit values, is 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.[7,8] In such environments, applications scale with additional containers, which run the same application on different sections of a common dataset. While each container serves different requests and accesses different data, a large number of the pages accessed is the same across containers.

0272-1732 © 2021 IEEE
Digital Object Identifier 10.1109/MM.2021.3073194
Date of publication 19 April 2021; date of current version 25 May 2021.

Another reason for the replication is that containers are created with forks, which replicate translations. Further, since containers are stateless, data are usually accessed through the mounting of directories and the memory mapping of files, which further creates translation sharing. 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.[9] All of this leads to numerous replicated page translations.

> *WE PROPOSE BABELFISH, A NOVEL ARCHITECTURE TO SHARE TRANSLATIONS ACROSS CONTAINERS IN THE TLB AND IN PAGE TABLES, WITHOUT SACRIFICING THE ISOLATION PROVIDED BY THE VIRTUAL MEMORY ABSTRACTION.*

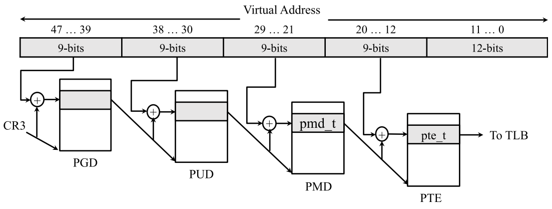

FIGURE 1. Page table walk.

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.

HANDLING TLB MISSES IN X86 LINUX

When a processor access misses in both L1 and L2 TLBs, a page table walk begins. This is a multistep process performed in hardware. Figure 1 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 contain 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 are 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 physical page number (PPN) and additional flags that the hardware uploads into the L1 TLB to proceed with the translation of the virtual address.

In theory, a page walk involves four cache hierarchy accesses. In practice, a 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.

OUR PROPOSAL: BABELFISH

To remedy this problem, we propose *BabelFish*, a novel architecture to share translations across containers in the TLB and in page tables, without sacrificing the isolation provided by the virtual memory abstraction. BabelFish eliminates the replication of {VPN, PPN} 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 the higher performance of containerized applications and functions, and faster container bring-up.

BabelFish has two parts. One enables TLB entry sharing, and the other enables page table entry sharing.

ENABLING TLB ENTRY SHARING

In containerized environments, TLBs may contain multiple entries with the same {VPN, PPN} pair, the same permission bits, and different PCID tags. Such replication 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 in the same CCID group will want to share many TLB 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 2 shows an example for a two-way set-associative TLB. This support allows all the processes in the same CCID group to share entries.

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.

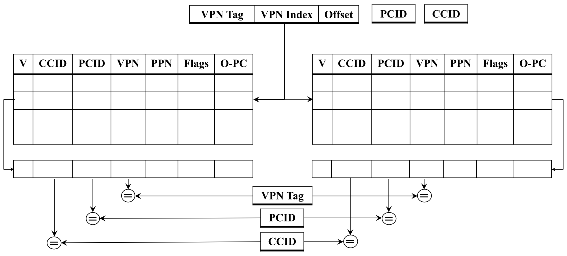

FIGURE 2. Two-way set-associative BabelFish TLB.

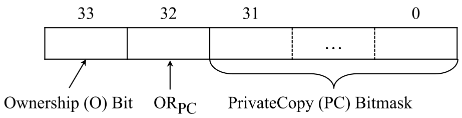

FIGURE 3. Ownership-PrivateCopy (O-PC) field. The PrivateCopy (PC) bitmask has a bit set for each process in the CCID group that has its own private copy of the page. The OR_{PC} bit is the logic OR of all the bits in the PC bitmask.

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 processes of the CCID group 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 of the CCID group have written to a copy-on-write (CoW) page and have made 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 2).

Ownership-PrivateCopy Field. The O-PC field is expanded in Figure 3. It contains the PrivateCopy (PC) bitmask, one bit that is the logic OR of all the bits in the PC bitmask (OR_{PC}), and the Ownership (*O*) bit. The PC bitmask has a bit set for each process of the CCID group that has its own private copy of this page. The rest of the processes of the CCID group, which can be an unlimited number, still share the clean shared page.

The BabelFish TLB is indexed as a regular TLB, using the VPN Tag. The hardware looks for a match in the VPN and 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.

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, which also includes an ordered list (*pid_list*) of processes that performed a CoW in 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 uses the second bit in the PC bitmask. More details are given in the conference paper.[10]

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 of 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*s requested from memory can thrash the cache hierarchy.[11] 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.

To solve this problem, BabelFish changes the page table structures so that processes of 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 4. The figure shows the translation of an address for two processes of 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). In the corresponding entries of their previous tables (PMD), both processes place 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*.

The default sharing level in BabelFish is a PTE table, which maps 512 4-KB 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 4-KB pages or 512 2-MB 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.

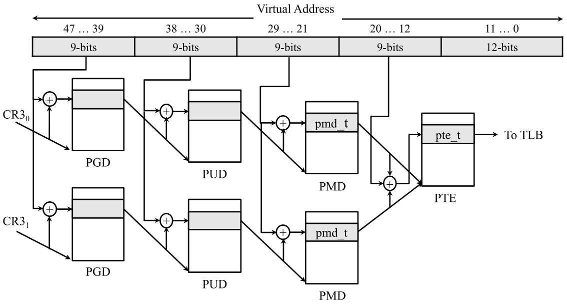

FIGURE 4. Page table sharing in BabelFish.

PUTTING IT ALL TOGETHER

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 5 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 5 show the conventional process. As container *A* accesses VPN_0, the access 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

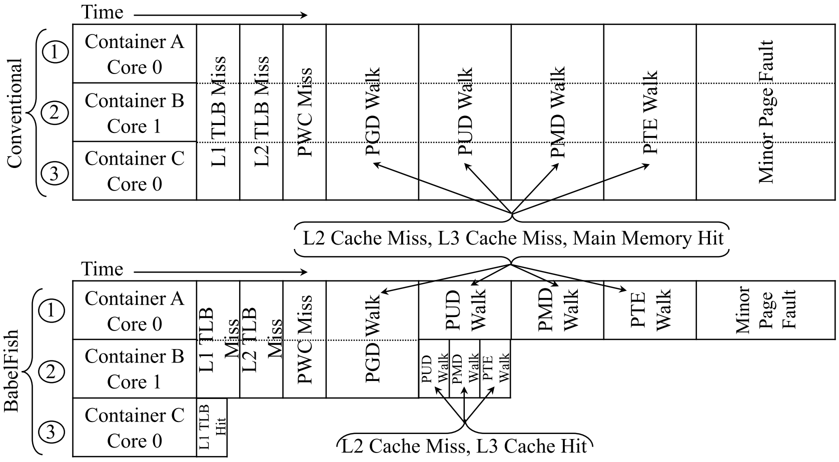

FIGURE 5. Timeline of the translation process in a conventional (top) and BabelFish (bottom) architecture. In the figure, container *A* runs on Core *0*, then container *B* on Core *1*, and then container *C* on Core *0*.

memory but not marked as present, *A* suffers a minor page fault as it completes the translation (see Figure 5). Finally, *A*'s page table is updated and a {VPN_0, PPN_0} translation for *A* 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 for *B* 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 a {VPN_0, PPN_0} translation for *C* is loaded into the TLB. The system does not take 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 5 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 coscheduled on the same physical core, either in SMT mode, or due to an oversubscribed system.

EVALUATION

We evaluate BabelFish with simulations of an 8-core processor running a set of Docker containers in an environment with conservative container colocation. We evaluate two types of containerized workloads (data serving and compute) and two types of FaaS workloads (dense and sparse). On average, under BabelFish, 53% of the translations in containerized workloads and 93% of the translations in FaaS workloads are shared.

Figure 6 shows the latency or time reduction obtained by the extensions proposed by BabelFish. BabelFish reduces the mean and tail (95th percentile) latency of containerized data-serving workloads by 11% and 18%, respectively. It also lowers the execution time of containerized compute workloads by 11%.

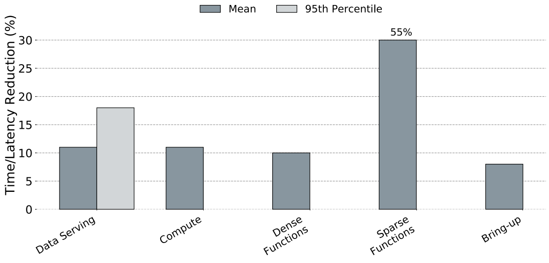

FIGURE 6. Latency or time reduction attained by BabelFish.

Finally, it reduces FaaS function execution time by 10%–55% and bring-up time by 8%.

FUTURE RESEARCH DIRECTIONS AND APPLICATIONS

The core concepts of BabelFish can be extended for other environments and resources. In this section, we present some possible directions.

Overcommitted Environments

While container environments place substantial pressure on the TLB and caches due to translations, as the working set of applications continues to increase, other workloads will also suffer from the same effects.

BabelFish novel design, which allows processes and containers within a group to share replicated translations in the TLB and in the cache hierarchy, is a general primitive that can be used in other environments to reduce the translation-induced pressure. Furthermore, BabelFish also supports the more advanced case where many of the processes of a group want to share the same translation, but a few other processes do not, and have made their own private copies. This overall design minimizes the context-switch overhead and effectively prefetches shared translation in the TLB and the caches.

BABELFISH NOVEL DESIGN, WHICH ALLOWS PROCESSES AND CONTAINERS WITHIN A GROUP TO SHARE REPLICATED TRANSLATIONS IN THE TLB AND IN THE CACHE HIERARCHY, IS A GENERAL PRIMITIVE THAT CAN BE USED IN OTHER ENVIRONMENTS TO REDUCE THE TRANSLATION-INDUCED PRESSURE.

Virtualized Environments

Conventional virtualized environments are slowed down by nested address translation. For example, a nested address translation may require up to 24 sequential memory accesses. In such deployments, content-aware page deduplication is a prevalent technique to reduce the memory pressure caused by memory overcommitment.

This page deduplication process creates page sharing in virtualized environments, generating more replicated translations. BabelFish can be extended to reduce this translation replication in virtualized environments.

Container Context Identifiers (CCIDs)

CCIDs are a new way to logically group processes and containers to enable resource sharing. CCIDs are a useful abstraction to allow the hardware and kernel to reason about the cooperation and isolation of execution contexts—enabling higher performance and security guarantees. BabelFish proposes the first-class support of CCIDs in both the kernel and in hardware, which represents a first step toward container-aware computing. Looking forward, it will be beneficial to provide more hardware and systems support for containerized and serverless environments.

REFERENCES

1. Docker, "What is a container?." [Online]. Available: https://www.docker.com/what-container
2. Google, "Production grade container orchestration." [Online]. Available: https://kubernetes.io
3. C. Tang *et al.*, "Twine: A unified cluster management system for shared infrastructure," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation,* USENIX Assoc., Nov. 2020.
4. N. Savage, "Going serverless," *Commun. ACM*, vol. 61, no. 2, pp. 15–16, Jan. 2018.
5. A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, Carlsbad, CA, USA, USENIX Assoc., 2018, pp. 427–444.
6. M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of Function-as-a-Service computing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 1063–1075.
7. B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *Proc. 8th USENIX Workshop Hot Top. Cloud Comput.*, Denver, CO, USA, Jun. 2016, pp. 108–113.
8. B. Ibryam, "Principles of container-based application design," Tech. Rep., Red Hat, Inc., 2017. [Online]. Available: https://www.redhat.com/cms/managed-files/cl-cloud-native-container-design-whitepaper-f8808kc-201710-v3-en.pdf f8808kc-201710-v3
9. IBM, "Docker at insane scale on IBM power systems." [Online]. Available: https://www.ibm.com/blogs/bluemix/2015/11/dockerinsane-scale-on-ibm-power-systems
10. D. Skarlatos, U. Darbaz, B. Gopireddy, N. S. Kim, and J. Torrellas, "Babelfish: Fusing address translations for containers," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 501–514.
11. Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, "CSALT: Context switch aware large TLB," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 449–462.