LETTER

Block Level TLB Coalescing for Buddy Memory Allocator

Jae Young HUR[†a)], ***Member***

SUMMARY Conventional TLB (Translation Lookaside Buffer) coalescing schemes do not fully exploit the contiguity that a memory allocator provides. The conventional schemes accordingly have certain performance overheads due to page table walks. To address this issue, we propose an efficient scheme, called block contiguity translation (BCT), that accommodates the block size information in a page table considering the Buddy algorithm. By fully exploiting the block-level contiguity, we can reduce the page table walks as certain physical memory is allocated in the contiguous way. Additionally, we present unified per-level page sizes to simplify the design and better utilize the contiguity information. Considering the state-of-the-art schemes as references, the comparative analysis and the performance simulations are conducted. Experiments indicate that the proposed scheme can improve the memory system performance with moderate hardware overheads.

key words: *architectures, memory allocation, translation lookaside buffer, page table, performance*

1. Introduction

A modern system typically accommodates memory management units (MMUs) to enhance memory utilization. MMU enables us to isolate the virtual address space from the resource constrained physical address space. However, MMU has certain performance overheads due to page table walks. To conduct an address translation, MMU accesses the main memory to acquire page table entries. This is called page table walk. The acquired page table entries are stored in a TLB for further use. The frequent page table walks can significantly degrade performance because an application can be stalled. Therefore, it is desired to reduce the page table walks and their overheads.

Modern OS (Operating System) allocates a physical memory typically in block level. As an example in Linux, the Buddy algorithm [1] is widely used as a default memory allocator that efficiently allocates free memory blocks for an application. A block contains the contiguous power-of-2 pages (or bytes). Traditionally, to exploit the contiguity and reduce page table walks, an architecture defines multiple page sizes. However, the traditional design does not fully exploit the capability of the contiguous allocator. Suppose OS allocates an 128kB sized physical block. If 128kB page size is not defined in the architecture, the system treats the block as 32 separate (4kB sized) pages. In this case, MMU conducts page table walks to separately acquire the 4kB pages. As a result, the traditional design can suffer from performance overheads due to page table walks. To reduce the overheads, we propose a novel technique that coalesces contiguous ranges in block level. Our approach combines the advantages of the block-level coalescing and the efficiency of a legacy memory management scheme. The main contributions of this paper are:

- To reduce page table walks, we propose the block-level TLB coalescing scheme considering the Buddy algorithm. In our scheme, a page table walk acquires the entire block mapping information. The acquired information is coalesced in a single TLB entry.
- To simplify the design and better utilize the contiguity, we present unified page sizes in a level of the hierarchical page table structure.
- The memory system performance and TLB hardware overheads are evaluated.

The paper is organized as follows. In Sect. 2, related work is described. In Sect. 3, we present the conventional designs. In Sect. 4, we present our design. In Sect. 5, the experimental results are described. Finally, the conclusions are drawn in Sect. 6.

2. Related Work

A number of TLB coalescing schemes mainly considering the Buddy algorithm are reported. In [2], the contiguity of *equal* sized blocks is represented in a page table. The TLB in [2] has *hybrid* type entries. To determine the block size, the special OS process is required. Our work is similar to [2] in that a certain concept of *block-level* contiguity is utilized in a page table. Our work differs from [2] in the following ways. First, in our design, the actual contiguity of possibly *variable* sized blocks in the physical address space is represented. Second, the TLB in our design has *homogeneous* type entries in that the contiguity is represented in any entry. Third, our design does not require the special OS process.

In our prior work [3], the *page-level* contiguity information is represented in a page table. In [3], the contiguity is represented for 4kB page size. Our work is close to [3] in that the contiguity information is accommodated in a page table. Our work differs from [3] in the following ways. First, the *block-level* contiguity is represented in a page table that can outperform [3]. Second, we present *unified per-level page sizes* that can represent the larger contiguity than [3].

Manuscript received May 7, 2019.
Manuscript publicized July 17, 2019.
[†]The author is with Vietnamese-German University, Binh Duong, Vietnam.
a) E-mail: JaeYoung.Hur@gmail.com
DOI: 10.1587/transinf.2019EDL8089

3. Conventional Designs

We review the conventional MMU operation, the prior TLB coalescing schemes, and their issues.

3.1 Traditional MMU Operation

When an application is invoked, OS allocates a memory space. An address space is divided in pages. In Fig. 1 (a), the application requires six virtual pages with VPNs (virtual page numbers) 0~5. OS finds free spaces in a physical memory. In Fig. 1 (a3), two blocks are available in the physical memory. As an example, in $Block_1$, *StartPPN* (Start physical page number) is 4 and *EndPPN* (End physical page number) is 7. OS maps VPNs into PPNs. The Buddy allocator conducts the mapping in block level with the granularity of power-of-2 pages. Then OS constructs a page table for the mapping and stores the page table in main memory. In Fig. 1 (b), the page table contains six PTEs (page table entries). A PTE contains a single page mapping information between VPN and PPN. Figure 1 (a2) depicts the PTE format in ARM v7 architecture [4].

When the application runs, a master accesses memory with virtual address (VA). Then MMU conducts a TLB lookup to check whether the PPN for the VPN is stored in TLB. If the PPN is not stored in TLB (or TLB is miss), MMU conducts a page table walk, acquires the PTE, and stores the PTE in TLB. Finally, MMU translates the virtual address into the physical address by converting VPN into PPN. In Fig. 1 (b), TLB contains six entries for VPNs 0~5. To do this, six page table walks can be required.

3.2 TLB Coalescing

To improve TLB utilization and reduce page table walks, multiple PTEs can be coalesced in a single TLB entry. To do this, the contiguity information can be represented in a page table. The system behavior is significantly affected by the representation methodology. Figure 2 (a) depicts the page table format of the *anchor translation* (AT) scheme [2]. The contiguity indicates how many pages are contiguous in the *ascending* direction starting from an *anchor* entry. In [2], the contiguity information is represented in every other *equal sized* virtual block (called *Distance*). The special OS process is required to determine the *Distance* (block size) value in the heuristic way. The TLB has two types (*anchor* and *regular*) of entries. The contiguity is represented only in the *anchor* entry. Two issues of [2] are the followings. First, only when the page table walk for an *anchor* entry is finished, TLB acquires the contiguity information. As an example, in Fig. 2 (a), the page table walk for VPN 1 acquires no contiguity information until the page table walk for VPN 0 is finished. Accordingly, page table walks can undesirably occur. Second, the contiguity value larger than the *Distance* can not be represented. Consequently, it is difficult to capture the *actual* contiguity information and fully exploit the capability of a memory allocator.

Figure 2 (b) depicts the *page-level* contiguity representation in [3]. The contiguity indicates how many *next* pages are contiguous in the *ascending* direction starting from an *accessed* entry. As an example, the page table walk for VPN 1 acquires the information that the contiguity value is 2. This means the next two PPNs 6~7 are contiguous to PPN 5. A main issue is that Fig. 2 (b) does not represent the contiguity in the *descending* direction. The page table walk for VPN 1 does not acquire the information that PPN 5 is contiguous to PPN 4. Accordingly, to access VPN 0, an additional page table walk is required. If the address pattern of an application is descending or random, certain page table walk overheads still can remain.

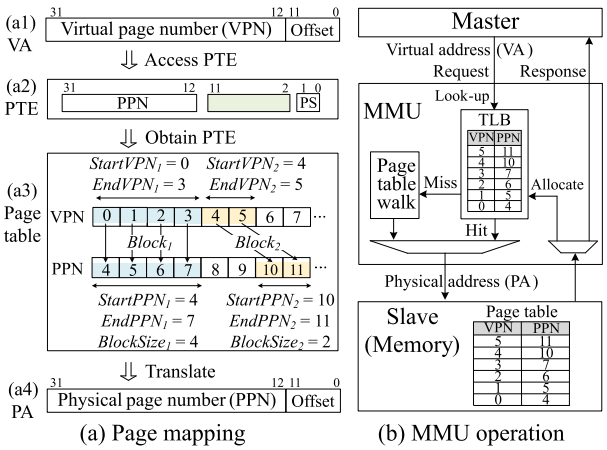


Fig. 1 Traditional MMU operation.

4. Proposed Design

In this section, we describe the property of the Buddy allocator and present our design to exploit the property.

4.1 Property of Buddy Allocator

We observe the significant property that *StartPPN* is *aligned* with a *block size*. In other words, *StartPPN* is multiple of a block size. Suppose the block size is 2^n pages where n is an unsigned integer. We formulate *StartPPN* and *StartVPN* using a modulo operation as follows.

$$StartPPN = \text{PPN} - (\text{PPN} \ \% \ 2^n) \tag{1}$$

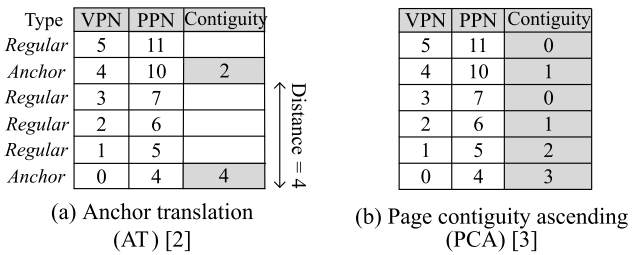


Fig. 2 Page table formats for TLB coalescing.

$$StartVPN = \text{VPN} - (\text{VPN} \ \% \ 2^n) \quad (2)$$

where PPN is the mapped physical page number for the accessed VPN. Considering the block size is 2^n pages, *EndPPN* and *EndVPN* are:

$$EndPPN = StartPPN + 2^n - 1 \quad (3)$$

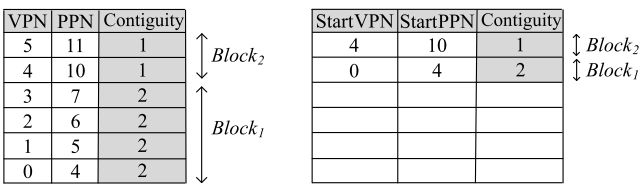
$$EndVPN = StartVPN + 2^n - 1 \quad (4)$$

Equations (1)~(4) suggest that, if value n is known for the single page mapping information (VPN, PPN), the entire block mapping information can be obtained.

4.2 Block-Level Coalescing

We represent the block size information (value n) in a page table entry. Considering the block size is the power-of-2 number, the value n instead of the actual block size is represented to reduce the number of bits. In our method, when any VPN is accessed, the information in Eqs. (1)~(4) is obtained. Figure 3 (a) depicts an example. When the page table walk for VPN 1 is conducted, the entire mapping information of $Block_1$ is obtained. Then the information is coalesced in a single TLB entry as depicted in Fig. 3 (b). When VPNs 0~3 are accessed later in any order, page table walks can be avoided because TLB is hit. To do this, Eqs. (1)~(4) are implemented in TLB hardware. Note the modulo operation (with the power-of-2 number) is implemented using simple bit operations. The main advantage of our design is that page table walks can be reduced. Table 1 shows the number of page table walks for different address patterns. In our design, regardless of address patterns, two page table walks are required. This is because a single page table walk acquires the entire block mapping information. For comparison, in [2], [3], depending on address patterns, more page table walks than our design can be required.

To support our design, OS (the Buddy allocator) should be modified. OS is required to set the block size information in a page table entry. Note that when a page table is constructed, OS *inherently* obtains the block size information (value n). In Fig. 3 (a), when $Block_1$ is allocated, the block size (4 pages) is an available information. Then OS simply writes the value 2 (= $\log_2 4$) in each entry. In practice, this can be implemented using several instructions. We measured the allocation time overhead in the Linux-based development platform. As a result, the overhead is less than 1% and can be insignificant.



(a) Page table (b) Coalesced TLB allocation

Fig. 3 Block contiguity translation (BCT). (a) The contiguity value (= $\log_2 BlockSize$) is represented in a page table. (b) A page table walk acquires the entire block mapping information. The acquired information is coalesced in a single TLB entry.

Table 1 The number of page table walks in different address pattern examples for the mappings of Figs. 2 and 3.

Address patterns	VPN access order	Traditional	AT [2]	PCA [3]	BCT
Ascending	0, 1, 2, 3, 4, 5	6	2	2	2
Descending	5, 4, 3, 2, 1, 0	6	4	6	2
Mixed	4, 5, 1, 2, 3, 0	6	4	3	2

4.3 Unified Per-Level Page Size

A page table is organized typically in multiple levels to avoid the large sized contiguous page table. Traditionally, a system only can represent the fixed page sizes defined in an architecture. In ARM v7 architecture, the defined page sizes are 1MB, 16MB (in level 1) and 4kB, 64kB (in level 2). We call 1MB and 4kB as *base* page sizes in each level. When a 64kB block is allocated, 64kB page size is used to represent the mapping. The system treats the block as a single 64kB page. The advantage is that only one coalesced TLB entry is used. However, to use 64kB page size in the traditional way, there is an architectural limitation that *StartVPN* should be multiple of 16 (pages). In other words, VA[15:12] value should be zero [4]. This limitation makes OS often difficult to use such a page size. Moreover, TLB has certain hardware overheads to implement multiple page sizes.

In our scheme, page sizes are unified in a level. In ARM v7 architecture, two *base* page sizes (1MB and 4kB) are only implemented. Figure 4 depicts an example. Two applications request 2MB and 16kB memory respectively. Suppose a 2MB physical block is available for the first application. In this case, two contiguous 1MB pages are allocated. Then OS represents the contiguity value 1 (= $\log_2 2$) in the level-1 PTE. Similarly, suppose a 16kB physical block is available for the second application. Then OS represents the contiguity value 2 (= $\log_2 4$) in the level-2 PTE. When MMU operates, only one coalesced TLB entry for each application is required. Additionally, there is no limitation of a virtual address. Moreover, the TLB hardware can be simplified because 64kB and 16MB page sizes are eliminated. In our experiment, by simplifying the page sizes, the hardware area is reduced by 15%. The maximum contiguity

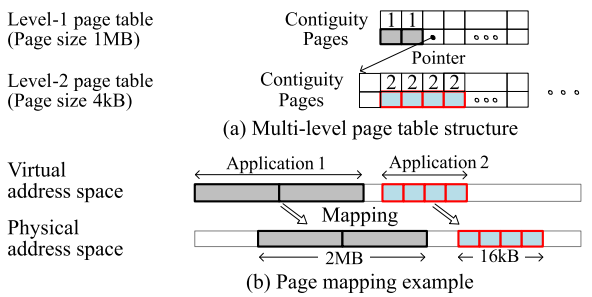
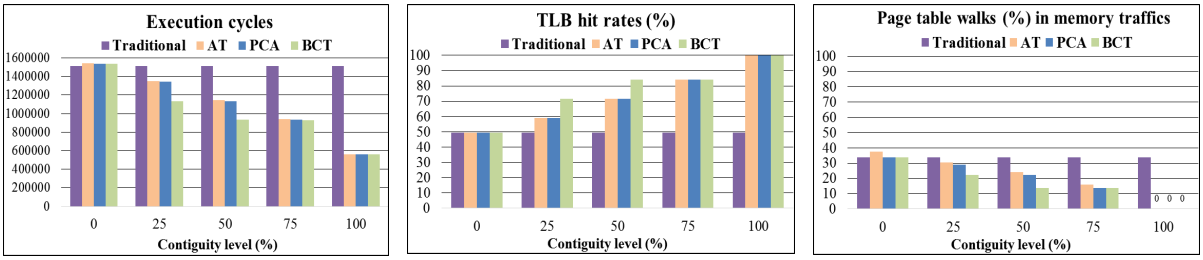


Fig. 4 Unified per-level page sizes in the hierarchical page table. Page sizes other than 1MB (for level 1) and 4kB (for level 2) are not required.



(a) Execution cycles (b) TLB hit rates (c) Page table walks in memory traffics

Fig. 5 Experimental results for *rotated camera preview* workload. An image size is 1280 × 720.

that can be represented is 2^n MB. For comparison, in [3], the maximum contiguity is 512kB.

5. Experimental Results

To evaluate the performance of our BCT design, a cycle-based transaction-level performance model is implemented in C++. The model is integrated in the simulation environment of [3]. We consider *rotated camera preview* workload widely used in a mobile device, where a camera captures an image and the rotated image is displayed. In this workload, descending and ascending address patterns are mixed. We conduct three performance simulations considering AT [2] and PCA [3] schemes as references.

First, to evaluate memory system performance, we measure execution cycles to finish a single frame. Figure 5 (a) depicts the results. When the physical memory is fragmented, the physical pages are randomly generated. We explore different *contiguity levels*. In case *contiguity level* is 0%, the physical memory is fully fragmented. In case *contiguity level* is 100%, a large chunk (128 pages) of free blocks are available. It is desired that a large chunk of free memory is allocated to an application. To evaluate our performance in the conservative way, similar to [3], the contiguous blocks in the physical memory is assumed to be uniformly distributed. The main results of our experiments are:

- The performance tends to improve as *contiguity level* increases. This is because of the decreased page table walks and the increased TLB hit rates.
- Our design performs better up to 18% than [2] and up to 17% than [3]. This is because of the improved TLB hit rates.

Second, to evaluate MMU performance, we measure TLB hit rates. Figure 5 (b) depicts the results. When *contiguity level* is low, TLB hit rate is significantly low. When *contiguity level* is 0%, TLB hit rate is 50%. This is undesirable and is because of the virtual address pattern. As *contiguity level* increases, TLB hit rate increases. Our design improves TLB hit rates up to 13% compared to [2], [3].

Third, to evaluate memory utilization, we measure the percentage of page table walks in the total memory traffics. Figure 5 (c) depicts the results. When *contiguity level* is low, page table walks significantly and undesirably occupy the memory bandwidth. In Fig. 5 (c), when *contiguity level* is 0%, page table walks occupy 34% of the memory traffics. As *contiguity level* increases, page table walks tend to decrease. Our design reduces page table walks up to 10% compared to [2] and up to 9% compared to [3].

Finally, we evaluate the hardware overheads of our design. We implemented set associative (16 sets, 8 ways) TLBs in Verilog and synthesized in Xilinx FPGA device. Table 2 shows the area in the number of LUTs (Look-up Tables). The targeted device contains 612000 LUTs in total. Our TLB requires 1.8× more area than the traditional design. This is because our TLB maintains a logic to handle the contiguity information. Our TLB requires 29% less area than [3]. This is because the allocation logic to merge the ranges in [3] is not required.

Table 2 TLB hardware area cost. The number of LUTs (Look-up Tables) in Xilinx Virtex-7 xc7vx980t.

Area	Traditional	PCA [3]	BCT
Number of LUTs	1811	4593	3268

6. Conclusions

We presented the block-level TLB coalescing scheme for the Buddy allocator. We described the modified Buddy allocator to support our scheme where the modification overhead is insignificant. We presented the unified per-level page sizes that can simplify the design and better represent the contiguity. By coalescing the contiguity information in block level, the memory system performance, TLB hit rates, and memory utilization can be improved with moderate hardware overheads.

References

[1] K.C. Knowlton, "A fast storage allocator," Commun. ACM, vol.8, no.10, pp.623–625, Oct. 1965.
[2] C.H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations," ACM/IEEE Int'l Symposium on Computer Architecture (ISCA17), pp.444–456, Toronto, Canada, June 2017.
[3] J.Y. Hur, "Contiguity representation in page table for memory management units," IEEE Trans. Very Large Scale Integr. (VLSI) Systems, vol.27, no.1, pp.147–158, Jan. 2019.
[4] ARM, Ltd., ARM Architecture Reference Manual, ARMv7-A Edition, available at http://www.arm.com