PAPER

Virtual Address Remapping with Configurable Tiles in Image Processing Applications

Jae Young HUR^{†a)}, *Member*

SUMMARY The conventional linear or tiled address maps can degrade performance and memory utilization when traffic patterns are not matched with an underlying address map. The address map is usually fixed at design time. Accordingly, it is difficult to adapt to given applications. Modern embedded system usually accommodates memory management units (MMUs). As a result, depending on virtual address patterns, the system can suffer from performance overheads due to page table walks. To alleviate this performance overhead, we propose to cluster and rearrange tiles to construct an MMU-aware configurable address map. To construct the clustered tiled map, the generic tile number remapping algorithm is presented. In the presented scheme, an address map is configured based on the adaptive dimensioning algorithm. Considering image processing applications, a design, an analysis, an implementation, and simulations are conducted. The results indicate the proposed method can improve the performance and the memory utilization with moderate hardware costs.

key words: address mapping, memory management units, image processing, performance, embedded system architectures

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. Despite this advantage, MMU has certain performance overheads due to page table walks. To conduct an address translation, MMU accesses main memory to acquire page table entries. This is called page table walk. Frequent page table walks can significantly degrade the performance because an application can be stalled. Therefore, it is desired to reduce page table walks and their overheads [1].

System performance is significantly affected by memory address patterns. In an image processing application, a memory access pattern is typically regular and has certain strides. A stride is defined as an address difference between adjacent memory transactions. As an example, in a raster-scan display and image blending applications, an access pattern is horizontally linear. In these applications, the stride is a transaction size, for example 64 B. When this linear access pattern is considered, a linear address map is favorable for MMU because temporal and spatial localities are high. In certain cases, however, the stride of an application is large. As an example, in an image rotation, an image is accessed

in the vertical direction. In this case, the stride is an image horizontal size. This pattern is not favorable for MMU especially when an image size increases because the locality becomes low. Accordingly, page table walks can frequently occur. To reduce the page table walk overheads, the tiled address map can be considered. When an image processing application operates in tile level, the tiled map can be better than the linear map. However, the tiled map still can be insufficient when an address stride is large. In this work, to reduce page table walks, we propose to rearrange and cluster tiles in a configurable way. In the presented scheme, the address map is configured based on the access pattern of a given application. The presented approach combines the advantages of the efficiency of the legacy address maps and the configurability of our design. Considering image processing workloads in input/output (I/O) devices, we present a generic algorithm, an analysis, a design, and experiments. The main contributions of this paper are:

- We present the novel tiled address map that can be clustered in a configurable way. To construct the configurable address map, the tile number remapping algorithm is proposed. The presented address map does not require the additional memory space when an image size is un-aligned with a tile size.
- The adaptive dimensioning algorithm to configure the address map is presented.
- The performance and the memory utilization of the presented address map are evaluated.
- A hardware design is presented and the overheads are evaluated.

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

2. Related Work

(1) *MMU architectures to reduce page table walks:* In [1]–[4], to reduce page table walks, the MMU architectures that exploit a contiguous allocation in a physical memory are presented. In [1], a page table contains the information on contiguous allocations. In [2] and [3], a single TLB entry can be mapped to multiple pages. The multipage mapping approaches such as Clustered TLBs [2] and CoLT [3] pack multiple page table entries into a single TLB entry. In [4],

Manuscript received July 10, 2019.

Manuscript revised September 10, 2019.

Manuscript publicized October 17, 2019.

[†]The author is with the Vietnamese-German University, Binh Duong, Vietnam.

a) E-mail: JaeYoung.Hur@gmail.com

DOI: 10.1587/transinf.2019EDP7194

an address range translation is presented. The range translation maps a contiguous virtual address range to contiguous physical pages. Our work differs from [1]–[4] in that we present a method to reduce page table walks by rearranging an address map for a given MMU architecture.

(2) *Data tiling*: A number of data tiling techniques for efficient memory accesses are reported. In [5], the tiling operation of 2D data for an embedded hardware accelerator is presented. When an application code has nested loops, the memory transfers of 2D (rectangular) data can be reduced using the loop-tiled operation and its scheduling. In [6], the software technique called data layout transformation for a given address map is presented. Considering a DRAM mapping, an 1D array is transformed into a structure for efficient processing in GPU. In [7], for a given address layout, an input/output MMU (IOMMU) design that operates partitioned data tiles is presented. In [7], page table walk overheads are reduced by exploiting the shared pages among the accelerators. Our work differs from [5]–[7] in that we present an address layout transformation taking a virtual memory mapping into account.

(3) *Address tiling*: A number of tiled address maps for efficient memory accesses are reported. In [8], the tile sizing algorithm using the constraint programming is presented. In [9], an MMU performance for various tile sizes is analyzed. In [9], an address generation algorithm especially for unaligned image sizes is not presented. In [10], the 4D tile format is presented where a dimension is fixed. In [11], a cache memory mapping with both unit tile and unit line accessibility based on the 4-level Z-order tiling layout is presented. Our work is similar to [9]–[11] in that a clustered tile address layout is presented. Our work differs from [9]–[11] in that the tiled map is configurable.

In [12], the tiled address map is rearranged to enhance memory bank interleaving. In our work, tiles can be clustered taking a virtual memory scheme into account. In [13], the tiled address map to reduce page table walk overheads is presented. Our work is close to [13] in that an address map is rearranged by clustering tiles. Our work differs from [12] and [13] in the followings ways. First, the address map is configurable. We present the clustering algorithm that operates in a systematic way. The presented design unifies the tiled map and the conventional linear map. The presented design does not require the additional memory space when an image size is un-aligned with a tile size. Second, we present the novel dimensioning algorithm to determine the configuration parameters. Third, performance and hardware overhead evaluations are presented.

3. Conventional Designs

An MMU operation and the conventional address maps are described as a background. Then their issues are presented.

Fig. 1 MMU operation.

3.1 MMU Operation

When an application is invoked, an operating system (OS) allocates the virtual memory space. The address space is divided in pages. When a page size is 4 kB, the offset in a page is 12 bits. OS finds free space in a physical memory. OS maps virtual pages into physical pages. As depicted in Fig. 1 (a), OS constructs a page table for the mapping. Then OS stores the page table in main memory. A page table entry (PTE) contains a physical page number (PPN) for a virtual page number (VPN). In Fig. 1 (b), the page table contains four PTEs. When an application runs, MMU translates a virtual address (VA) into a physical address (PA) by converting VPN into PPN.

In Fig. 1 (b), MMU operates as follows. A master generates transactions with virtual addresses. A transaction has a request channel (to memory) and a response channel (from memory). First, in the request channel, MMU receives a request transaction from the master. TLB lookup is conducted. If TLB is hit, MMU translates the address and sends the transaction to the memory. If TLB is miss, MMU conducts a page table walk (PTW). Second, in the response channel, MMU receives a response from the memory. If the response is a page table walk, TLB is updated and the address is translated.

The capacity of a page table walk is defined by the following:

$$PtwCapacity = (\text{Number of acquired PTEs}) \times (\text{Page size}), \quad (1)$$

where $PtwCapacity$ indicates the address space size (in bytes) that a page table walk can cover. This is determined by the number of acquired page table entries and a page size. There are two types of page table walks.

(1) *Single fetch*: A page table walk acquires a demanding

Table 1 Parameter definitions in the conventional linear and tiled maps.

Parameters	Description	Unit
$BaseAddr$	Base address	-
(A, B)	Pixel coordinate (Horizontal, Vertical)	Pixel
$(ImgH, ImgV)$	Image size (Horizontal, Vertical)	Pixel
$(TileH, TileV)$	Tile size (Horizontal, Vertical)	Pixel
$BytePixel$	Pixel size	Byte
$AposInTile$	Horizontal position of coordinate A in a tile $= (A + ImgH \times \lfloor \frac{B}{TileV} \rfloor) \% TileH$	Pixel
$BposInTile$	Vertical position of coordinate B in a tile $= B \% TileV$	Pixel
$TileSize$	Tile size $= TileH \times TileV \times BytePixel$	Byte

single page table entry. This is called *single fetch* (SF). In Fig. 1 (b), suppose MMU receives a request transaction with the VPN 2. If TLB is miss, MMU conducts a page table walk and acquires the PPN 4. Then MMU converts the VPN 2 into the PPN 4. When a page size is 4 kB, $PtwCapacity$ is 4 kB ($= 1 \times 4kB$).

(2) *Block fetch*: A page table walk can acquire multiple consecutive PTEs [14]. This is called *block fetch* (BF). The number of entries to acquire is determined by data width, burst length of a transaction, and a PTE size. Suppose data width is 16 B, burst length is 4, and a PTE size is 4 B. Then the block fetch can acquire up to 16 ($= \frac{16 \times 4}{4}$) PTEs. Compared to the single fetch, the block fetch can reduce page table walks. When a page size is 4 kB and the block fetch acquires 16 PTEs, $PtwCapacity$ is 64 kB ($= 16 \times 4kB$).

3.2 Address Maps

An address map associates an image pixel coordinate with the virtual address of a transaction. There are three general rules to map an image pixel into an address. First, a unique image pixel is mapped to a unique address. Second, a virtual address should be within the allocated memory space. Third, when multiple masters (or processes) share the address space, the masters should use the same address map to maintain the consistent data communication. In Table 1, the parameters are shown. There are two types of address maps widely used in practice.

(1) *Linear map*: In Fig. 2 (a), the linear map is depicted. Addresses sequentially increase in the horizontal direction. An arrow indicates the order of addresses. The address of a coordinate (A, B) is:

$$Address = BaseAddr + BytePixel \times (A + B \times ImgH), \quad (2)$$

where $BaseAddr$ denotes the base address of an allocated memory. $BaseAddr$ is the address of the coordinate $(0, 0)$.

(2) *Tiled map*: In Fig. 2 (b), the tiled map [12] is depicted. A number in a circle is a tile number and indicates the order of tiles. Tile numbers are linearly arranged. Additionally, the pixel addresses inside a tile are linearly arranged. As

Fig. 2 Conventional address maps.

a result, an image constitutes a *band of tiles* as depicted in Fig. 2 (c). The main advantage is that an image size is not necessary to be aligned with a tile size. The address of a coordinate (A, B) is [12]:

$$Address = BaseAddr + (TileNum \times TileSize) + BytePixel \times (BposInTile \times TileH + AposInTile), \quad (3)$$

where $TileNum$ denotes a tile number. This can be calculated by the following [12]:

$$TileNum \text{ of } (A, B) = \lfloor \frac{A + (ImgH \times \lfloor \frac{B}{TileV} \rfloor)}{TileH} \rfloor. \quad (4)$$

Suppose a tile size is 16×16 pixels. $BaseAddr$ is 1000. $ImgH$ is 128 pixels. Then $TileNum$ for the coordinate $(16, 1)$ is 1 ($= \lfloor \frac{16 + (128 \times \lfloor \frac{1}{16} \rfloor)}{16} \rfloor$). $TileSize$ is 1024 B ($= 16 \times 16 \times 4B$). $BposInTile$ is 1 ($= 1 \% 16$). $TileH$ is 16 pixels. $AposInTile$ is 0 ($= (16 + 128 \times \lfloor \frac{1}{16} \rfloor) \% 16$). In this case, the address of the coordinate $(16, 1)$ is 2088 ($= 1000 + (1 \times 1024) + 4 \times (1 \times 16 + 0)$).

3.3 Issues

Two issues of the conventional address maps are as follows. A main issue is the mismatch between an access pattern and an underlying address map. The mismatch can incur frequent page table walks. When a page table walk occurs, subsequent transactions can be stalled to maintain the data order especially in a streaming multimedia application. In Fig. 3 (a), a linear address mapping example is depicted in the transaction granularity. An address is represented in a hexa-decimal number. In this example, a transaction size is 64 B. When $BytePixel$ is 4 B, a transaction accesses 16 ($= \frac{64}{4}$) pixels. Suppose a page size is 4 kB. When an access pattern is linear, adjacent transactions in an image row access the same page. In this case, a page table walk is not required. Subsequently, the linear map is inherently suitable for a linear access pattern. However, when an image is accessed in the vertical direction, an address stride is 4 kB. This means the adjacent transactions access different pages,

Fig. 3 Address mapping examples.

incur TLB misses, and incur page table walks. To alleviate this problem, a tiled map can be used. In Fig. 3 (b), the tiled map [12] is depicted. A tile size is 1 kB ($= 16 \times 16 \times 4B$). Accordingly, the addresses in a tile are within the same 4-kB boundary. This means the addresses in a tile have the high spatial locality. This can improve a TLB hit rate compared to the linear map. However, the tiled map still can have certain page table walk overheads because of the mismatch issue. It is noted that different applications can have different access patterns. Based on these motivations, an adaptive approach is proposed in the next section as a solution to the mismatch issue.

Another issue is an allocated memory size. The address map in [13] inherently assumes that an image size is aligned with a tile size. This means an image size should be multiple of a tile size. As an example, in Figs. 3 (a) and 3 (b), the allocated memory size is 256 kB ($= 1024 \times 64 \times 4B$). Suppose an image size is 1016×56 . In Fig. 3 (a), the allocated memory size is 222 kB ($= 1016 \times 56 \times 4B$). In the approach of [13], the allocated memory size is still 256 kB to align an image size with a tile size. In this case, the memory space 34 ($= 256 - 222$) kB is additionally allocated and is not utilized.

4. Address Remapping with Configurable Tiles

The aim is to alleviate the above-mentioned mismatch issue, reduce page table walks, improve memory utilization, and improve performance. Additionally, we aim to avoid the wasted memory space when an image size is un-aligned with a tile size. We mainly consider an I/O device accelerator that operates dedicated image processing tasks in an

Fig. 4 The proposed design approach.

Table 2 Main parameters of the proposed map.

Parameters	Description	Unit
LEVEL	Number of levels for clustering	-
n_j	Clustered tile length in the level j . A clustered tile contains $n_j \times n_j$ tiles.	Tile
T	Number of tiles in a granular row	Tile
TileNum _{j}	Tile number for a pixel coordinate in the level j	-

embedded system. In this case, the address map hardware calculates an address for an image pixel coordinate. It is desired to design the hardware with an insignificant cost. In this section, the proposed address map is presented. Additionally, the adaptive dimensioning algorithm to configure the address map is presented.

4.1 Overview

The general approach is to devise an address map such that an access pattern and an underlying address map are as close as possible. To do this, an address layout is rearranged to adapt to an access pattern. The method to rearrange an address layout is to cluster tiles in a systematic way. To efficiently implement the adaptivity feature, a configurable map is devised. In Fig. 4, the design approach is depicted. In Table 2, the main parameters are shown. In Fig. 4 (a), the level-0 tiled map is depicted. This operates as a baseline. The linear map can be implemented by configuring $TileV$ as 1. In Figs. 4 (c) and 4 (d), the clustered map examples are depicted. These can be built by remapping tile numbers. For a given level j , the address map operates the following three steps:

1. For a coordinate (A, B) , $TileNum_0$ is calculated using Eq. (4).
2. $TileNum_0$ is remapped to $TileNum_j$.
3. For $TileNum_j$, an address is calculated using Eq. (3).

where $TileNum_j$ denotes a tile number in the level j .

Table 3 Variables in Fig. 5.

Parameters	Description	Unit
$GRowNum$	Granular row number	-
$CRowNum$	Circular $GRowNum$	-
$PairNum$	Pair number	-
$DistInRow$	Tile number distance between adjacent pairs inside a granular row	Tile
$DistOutRow$	Tile number distance between the pairs outside a granular row	Tile
$Distance$	Tile number distance between the original tile and the remapped tile	Tile

Tile number remapping algorithm

Given: n_j , $LEVEL$, $TileH$, $ImgH$

Input: $TileNum_0$ (Tile number in level 0)

Output: $TileNum_j$ (Tile number in level j)

```

1.  $T = n_j \times \text{floor}(ImgH / (TileH \times n_j))$ 
2.  $GRowNum = \text{floor}(TileNum_0 / T)$ 
3. // Level-1 remapping
4.  $DistInRow_1 = n_j \times (n_j - 1)$ 
5.  $DistOutRow_1 = T - n_j$ 
6.  $PairNum_1 = \text{floor}((TileNum_0 \% T) / n_j)$ 
7.  $CRowNum_1 = GRowNum \% n_j$ 
8.  $Distance_1 = (PairNum_1 \times DistInRow_1) - (CRowNum_1 \times DistOutRow_1)$ 
9.  $TileNum_1 = TileNum_0 + Distance_1$ 
10. // Hierarchical remapping
11. for ( $j = 2 ; j = LEVEL ; j++$ ) do
12.    $n_j = n_{j-1} / 2$ 
13.    $DistInRow_j = n_j \times (n_j - 1)$ 
14.    $DistOutRow_j = n_{j-1} - n_j$ 
15.    $PairNum_j = \text{floor}((TileNum_0 \% n_{j-1}) / n_j)$ 
16.    $CRowNum_j = GRowNum \% n_j$ 
17.    $Distance_j = (PairNum_j \times DistInRow_j) - (CRowNum_j \times DistOutRow_j)$ 
18.    $TileNum_j = TileNum_{j-1} + Distance_j$ 
19. end for

```

Fig. 5 Tile number remapping algorithm.

The level-1 clustered tile contains $n_1 \times n_1$ tiles. In Fig. 4 (c), n_1 is 4 tiles. The level-2 clustered tile contains $n_2 \times n_2$ tiles within $n_1 \times n_1$ tiles. In Fig. 4 (d), n_2 is 2 tiles. To simplify hardware implementations, n_{j+1} is defined by $\frac{n_j}{2}$. Additionally, n_j , $TileH$, and $TileV$ are power-of-2 numbers. Five parameters to determine an address layout are an image size ($ImgH$, $ImgV$), a tile size ($TileH$, $TileV$), a clustered-tile length (n_1), a pixel size ($BytePixel$), and a level ($LEVEL$). When these parameter values are available, a unique address map can be obtained. A tiled map is denoted by $TileL_x n_y$. In this notation, x is a level and y is n_1 . As an example, Fig. 4 (c) is denoted by $TileL_1 n_4$.

4.2 Tile Number Remapping

In this section, the algorithm to remap a tile number is presented. In Table 3, the variables are shown. In Fig. 5, the algorithm is shown. For an input ($TileNum_0$) and given values, $Distance_j$ (in the level j) is calculated. $Distance_j$ indicates the difference of the tile numbers before and after the remapping. Then $TileNum_j$ is obtained by $TileNum_{j-1} +$

$Distance_j$. The algorithm operates in the following way:

- When an application is invoked, an image size ($ImgH$) and the dimension information ($TileH$, $TileV$, n_1 , $LEVEL$) are determined. Then T , n_j , $DistInRow_j$, and $DistOutRow_j$ are derived.
- When an application runs, for a pixel coordinate, $TileNum_0$, $GRowNum_j$, $PairNum_j$, $CRowNum_j$, and $Distance_j$ are calculated. Finally, $TileNum_j$ is obtained.

The design can operate in multiple levels. The level-1 clustering algorithm is described in lines 4~9 in Fig. 5. In the level 1, T is not necessarily a power-of-2 number. The algorithm further allows multi-level clustering up to the user-defined $LEVEL$. This is described in lines 11~19 in Fig. 5. It operates similar to the level 1. The difference is that the clustering (in the level j) is conducted within $n_{j-1} \times n_{j-1}$ tiles.

In Fig. 6, an example is depicted. Tiles in Fig. 6 (a) are clustered as depicted in Figs. 6 (b) and 6 (c). To obtain Fig. 6 (b), the parameter values are calculated as shown in Fig. 6 (d). To obtain Fig. 6 (c), the parameter values are calculated as shown in Fig. 6 (e). In Fig. 6 (f), when a coordinate is (16, 16), $TileNum_0$ is 9. $Distance_1$ is -4. Then $TileNum_1$ is 5 ($= TileNum_0 + Distance_1 = 9 - 4$). $Distance_2$ is -2. Then $TileNum_2$ is 3 ($= TileNum_1 + Distance_2 = 5 - 2$).

The algorithm in Fig. 5 suggests that $Distance_j$ can be directly calculated from an input ($TileNum_0$) and given values. Therefore, the remapped tile number can be calculated by the following:

$$TileNum_j = TileNum_0 + \sum_{i=1}^j Distance_i \quad (5)$$

where j is a natural integer and indicates a level.

4.3 Handling Un-Aligned Image Size

The advantage of the presented design is that an image size is not necessary to be aligned with a tile size. In the presented approach, the allocated memory size (in bytes) is $ImgH \times ImgV \times BytePixel$. In Fig. 7 (a), the level-0 map is depicted. The image contains 4 rows. A row contains 4.5 tiles. The algorithm operates in the granularity of T tiles. Accordingly, the algorithm treats Fig. 7 (a) as 7 (b). We call a row in Fig. 7 (b) a *granular row* to differentiate with an original row in Fig. 7 (a). The image contains 5 granular rows. A granular row contains 4 tiles. T ($= 4$ tiles) is aligned with n_1 ($= 2$ tiles) in that T is multiple of n_1 . To construct a clustered map, tile numbers are remapped from Fig. 7 (b) to Fig. 7 (c). In Fig. 7 (c), a clustered tile contains 2×2 tiles (or 32×32 pixels). In Fig. 7 (d), the final address layout is depicted. In this example, the remapping is applied to $GRowNum$ 0~3 (or tile numbers 0~15). In this way, the presented tiled map can be generated for an un-aligned image size.

In the approach of [13], when an image size is un-aligned with a clustered-tile size, it is required to allocate

Fig. 6 Tile number remapping example.

Fig. 7 Tile number remapping for an un-aligned image size.

the additional memory space. The allocated memory size (in bytes) is the following:

$$\text{Memory size} = \left\lceil \frac{\text{ImgH}}{(n_1 \times \text{TileH})} \right\rceil \times (n_1 \times \text{TileH}) \times \left\lceil \frac{\text{ImgV}}{(n_1 \times \text{TileV})} \right\rceil \times (n_1 \times \text{TileV}) \times \text{BytePixel}. \quad (6)$$

In Fig. 7, our design requires 18 kB (= 72 × 64 × 4B). For comparison, the design in [13] requires 24 kB (= 96 × 64 × 4B) to make an image size aligned with 32 pixels. This means the design in [13] requires 33% more memory space than our design. In Table 4, the allocated memory size over-

Table 4 Allocated memory space overhead of [13] in percentage.

Image sizes	n_1				
	2	4	8	16	32
720 × 480	2.2	12.1	12.1	12.1	34.1
1280 × 720	2.2	6.3	6.3	6.3	41.4
1440 × 1080	0	2.2	6.3	6.3	21.9
1920 × 1080	0.7	0.7	6.3	20.9	34.1
4096 × 2160	0.7	0.7	0.7	6.3	15.6

head in [13] is shown. As an example, when an image size is 720 × 480 and n_1 is 4, the design in [13] requires 12.1% more memory space than our design.

4.4 Configurable Hardware Design

The presented address map is configurable in that various address layouts are implemented using a single hardware design. In Fig. 8 (a), the address map logic is depicted. This logic calculates an address for a coordinate. Three components are as follows.

(1) *Level-0 tile number calculation*: For a coordinate (A, B), $TileNum_0$ is calculated using Eq. (4).

(2) *Tile number remapping*: In Fig. 8 (b), the logic is depicted. This logic implements the tile number remapping algorithm in Fig. 5. In this logic, maximally three levels are allowed. $NewTileNum$ is calculated using Eq. (5).

(3) *Address calculation*: For the calculated tile number, an address is obtained using Eq. (3).

Fig. 8 The address map logic.

4.5 Adaptive Dimensioning

In Fig. 8, a unique address map for a given image can be determined by four configuration parameters ($TileH$, $TileV$, n_1 , and $LEVEL$). In this section, the method to determine these parameters is presented. Image processing applications (related to image display) are mainly considered. The presented approach allows OS to determine an address map using access patterns and the type of page table walks. The design space to determine the parameters is significant. Therefore, an algorithm to reduce complexity is presented. The parameter values can be determined as follows.

(1) *Tile size* ($TileH$, $TileV$): A tile contains $TileH \times TileV$ pixels. These can be determined by a transaction size. Then adjacent pixel data can be packed in a single transaction. As an example, when *BytePixel* is 4 B and a transaction size is 64 B, $TileH$ and $TileV$ would be chosen by 16 ($= \frac{64}{4}$) pixels.

(2) *Clustered tile length* (n_1): The level-1 clustered tile contains $n_1 \times n_1$ tiles. To narrow down the design exploration space, n_1 is constrained by the following:

$$n_1 \leq \sqrt{P}, \text{ where } P = \frac{PtwCapacity}{TileSize}, \quad (7)$$

where $PtwCapacity$ is the capacity of a page table walk.

Dimensioning algorithm

Given: Access pattern

Input: $TileV$, $TileSize$, $PtwCapacity$

Output: Clustered tile length n_1 , $LEVEL$

```

1. // Find  $n_1$  and  $LEVEL$  to minimize cost function
2.  $P = PtwCapacity / TileSize$ 
3.  $MaxLevel = \text{floor}(\log_2(\sqrt{P}))$ 
4. for ( $j=0$ ;  $j \leq MaxLevel$ ;  $j++$ ) do
5.    $N = 2^j$ 
6.   // Evaluate cost function
7.    $CostVer = 1 / (TileV \times N)$ 
8.    $CostHor = N / P$ 
9.   if access pattern is vertical do
10.     $Cost = CostVer$ 
11.   else if access pattern is vertical and horizontal do
12.     $Cost = CostVer + CostHor$ 
13.   end if
14.   if  $Cost$  is minimum do
15.     $n_1 = N$ 
16.     $LEVEL = j$ 
17.   end if
18. end for

```

Fig. 9 Dimensioning algorithm to obtain n_1 and $LEVEL$.

$TileSize$ is a tile size in bytes. We give this constraint to enable a page table walk to cover the address space for a clustered tile. In Fig. 9, the algorithm to determine n_1 is shown. In lines 7~8 of Fig. 9, cost functions are defined. The cost function represents the approximate and the relative amount of TLB misses assuming a TLB is empty as follows.

- $CostHor$ is a cost when an access pattern is horizontal. $CostHor$ is defined by $\frac{n_1}{P}$. This is because a TLB miss can occur once every $\frac{P}{n_1}$ transactions. As an example, in Fig. 3 (b), n_1 is 1. When $PtwCapacity$ is 4 kB and $TileSize$ is 1 kB, P is 4 ($= \frac{4kB}{1kB}$). Then a TLB miss can occur once every 4 ($= \frac{4}{1}$) transactions. Accordingly, $CostHor$ is $\frac{1}{4}$.
- $CostVer$ is a cost when an access pattern is vertical. $CostVer$ is defined by $\frac{1}{TileV \times n_1}$. This is because a TLB miss can occur once every $TileV \times n_1$ transactions. As an example, in Fig. 3 (b), $TileV$ is 16 and n_1 is 1. Then a TLB miss can occur once every 16 ($= 16 \times 1$) transactions. Accordingly, $CostVer$ is $\frac{1}{16}$.

In this way, the design space is explored to find n_1 such that $Cost$ is minimized. As $1 + \log_2 \sqrt{P}$ iterations are required, the algorithm requires $O(\log_2 \sqrt{P})$ complexity.

(3) $LEVEL$: When n_1 is obtained, $LEVEL$ is determined by the following:

$$LEVEL = \log_2 n_1. \quad (8)$$

A dimensioning example is as follows. When $TileSize$ is 1 kB and $TileV$ is 16 pixels, the results are depicted in Fig. 10. The $Cost$ values vary with n_1 values, an access pattern, and the type of page table walks. Suppose a page table walk is the single fetch. Then $PtwCapacity$ is 4 kB. In case

Fig. 10 Dimensioning example to determine n_1 . *Cost* in Fig. 9 is evaluated with different n_1 values. An access pattern can be vertical (*Ver*) or horizontal (*Hor*).

an access pattern is vertical, n_1 is determined by 2 as suggested in Fig. 10(a). This is because *Cost* is the minimum when n_1 is 2. *LEVEL* is determined by 1 ($= \log_2 2$). Accordingly, a clustered tile contains 2×2 tiles in the level 1. This layout is denoted by $\text{TileL}_1 n_2$.

Another dimensioning example is as follows. Suppose a page table walk is the block fetch. Then *PtwCapacity* is 64 kB. In case an access pattern is vertical, n_1 is determined by 8 as suggested in Fig. 10(b). *LEVEL* is determined by 3 ($= \log_2 8$). Accordingly, a clustered tile contains 8×8 tiles in the level 1, 4×4 tiles in the level 2, and 2×2 tiles in the level 3. This layout is denoted by $\text{TileL}_3 n_8$.

4.6 Configuration and Remapping

We describe when the address maps are configured and when the remapping occurs. The targeted address map is configured when an application is invoked. An access pattern can be known when an application is invoked. OS (or device driver) determines a targeted address map and configures the hardware. To do this, OS can use the dimensioning algorithm shown in Fig. 9. Alternatively, a system designer can analyze various access patterns in off line. The address pattern of an image processing application is typically regular. The off-line analysis result (for example, Fig. 10) can be stored in a table structure. Then OS can refer to the table when an application is invoked. As an example, in a mobile device, when an image is generated in the portrait mode and displayed in the landscape mode, the image should be rotated. Then the access pattern will be vertical. In case MMU conducts the single fetch, OS can refer to Fig. 10(a). The targeted address map can be accordingly configured.

The address map hardware conducts the remapping in run time to implement the targeted map. The proposed design technique (shown in Fig. 4) is to construct the targeted address map by remapping a tile number. It is noted that the remapping does not change the targeted address map. The targeted address layout does not change during run time.

4.7 Consideration on Data Channel Layouts

In the previous sections, the packed pixel data format with a single plane was considered. In this format, pixel data have an interleaved layout, for example, RGBRGB..., which is considered by *BytePixel*. All pixel data are stored in a single buffer. This format was considered for the sake of simplicity.

Fig. 11 Signal interface.

To design an address map, three rules described in Sect. 3.2 are obeyed. Additionally, it is assumed that image pixels in a single plane have the same size denoted by *BytePixel*. When there are multiple planes, a plane has a separate buffer allocated in a memory. The presented approach is to devise per-plane address layouts for a given access pattern and a channel layout. An address map can be separately determined for a plane. The operation scenario is the following:

- When an application is invoked, OS separately configures the targeted address map for a plane. In Fig. 11, the configuration parameters are shown in a shaded rectangle.
- When an application runs, the hardware controller in an image processing unit sends the signals to the address map for a plane. The signal interface is depicted in Fig. 11. Then the address map hardware calculates an address for given inputs.

The hardware controller should be appropriately designed to support various channel layouts. In this work, we focus on the address map hardware design and the configuration. The design of a controller for various channel layouts and extensive performance evaluations are left for future work.

5. Experimental Results

The experimented system is depicted in Fig. 12. The system is configured as shown in Table 5. A cycle-based transaction-level performance model is implemented in C++. The model is integrated in the simulation environment of [1]. Traffic generators are implemented to represent the memory access behavior of masters. An image size is 4096×2160 pixels. A pixel is 4-byte sized RGB format. A tile size is 16×16 pixels. A transaction contains 64 bytes of data. The interface of a component operates with AXI bus protocol [15]. A single datum is 128 bits wide. A physical address is 32 bits wide. When a virtual page is mapped to a physical page, the physical pages are randomly generated. A page size is 4 kB widely used in Linux and other operating systems.

In Table 6, the workloads for the simulation are shown. In *camera preview*, a camera captures an image and the raster-scan image is displayed. In *image scaling*, an image is resized. In *image blending*, two images are combined to generate a composite image. In *rotated preview*, a camera captures an image and a rotated image is displayed. To eval-

Fig. 12 System organization.

Table 5 System configuration.

Components	Item	Configuration
IOMMU	TLB	16 entries, Fully associative cache
	Replacement	Least Recently Used (LRU)
Interconnect	Arbitration	Round-robin
	Multiple outstanding	Max. 16
Memory controller	Address mapping	Row number: PA[31:14] Bank number: PA[13:11] Column number: PA[10:0]
	Request queue	16 entries
Memory (DRAM)	Model	DDR3-800
	Timing	$t_{CL}-t_{RCD}-t_{RP} = 5 - 5 - 5$
	Interface	One channel single chip
	Banks	8

uate performance and memory utilization, we conduct four experiments.

First, to evaluate the memory system performance, we measure execution cycles to finish a single frame. The execution cycles are obtained when the last transaction arrives at a line buffer in a master. It is noted that a scenario is associated with multiple processes where multiple masters concurrently operate. Accordingly, transactions from multiple masters are mixed in the shared main memory. We explore different types of address maps and page table walks. In Fig. 13, the results are shown. This is summarized in Table 6. The single fetch (SF) acquires a single page table entry. The block fetch (BF) acquires 16 page table entries. An access pattern is horizontal (Hor) or vertical (Ver). To evaluate the performance, the individual improvements (in percentages) of the selected map over the other maps are calculated. The baselines are the performances of the address maps other than the selected map. Then Δ is obtained by averaging the individual improvements. Our selected map is determined using the dimensioning algorithm in Fig. 9. $\text{TileL}_{x \times y}$ indicates a clustered tile, where $LEVEL$ is x and n_1 is y tiles. We experimented with other image sizes and obtained similar results. The main results are:

Table 6 Summarized performance results of Fig. 13.

Page table walk	Workload	Access patterns	Selected map	Δ (%)
(a) Single fetch	Camera preview	Hor	Linear	35.5
	Image scaling	Hor	Linear	49.9
	Image blending	Hor	Linear	48.0
	Rotated preview	Hor, Ver	TileL_0	13.3
	Rotated display	Ver	$\text{TileL}_{1 \times 2}$	14.5
(b) Block fetch	Camera preview	Hor	Linear	26.1
	Image scaling	Hor	Linear	38.2
	Image blending	Hor	Linear	34.0
	Rotated preview	Hor, Ver	$\text{TileL}_{1 \times 2}$	5.6
	Rotated display	Ver	$\text{TileL}_{3 \times 8}$	15.5

- Performance varies with access patterns and underlying address maps. When an access pattern is linear, the linear map is better than others. When an access pattern is not linear, our clustered tiled map is better than the linear map. This is because TLB hit rates are improved when the clustered tiled map is used.
- The block fetch tends to perform much better than the single fetch. However, performance still varies with access patterns and the address maps. Performance tends to improve when our selected map is used.
- In Fig. 13, our selected map performs 28% better than others on average.

Second, to evaluate MMU performance, we measure TLB hit rates. In Fig. 14, the results are shown. In *camera preview*, *image scaling*, and *image blending* workloads, the linear map is better than others. This is because TLB hit rate in the linear map is higher than 98% and is sufficiently high. However, in *rotated preview* and *rotated display* workloads, TLB hit rate in the linear map is significantly low. As an example, in Fig. 14 (a), TLB hit rates are 50% in *rotated preview* and 0% in *rotated display*. This is undesirable and is because of the vertical access pattern. TLB hit rates increase when our tiled maps are used. This means our approach can improve MMU performance. In Fig. 14 (b), the block fetch performs better than the single fetch. Even in this case, TLB hit rates tend to improve when our selected map is used.

Third, to evaluate the memory utilization, we measure bandwidth overheads of page table walks. The overhead is measured by $\frac{\text{Total number of page table walks}}{\text{Total number of transactions}}$ in main memory. In Fig. 15, the percentage of page table walks in the memory bandwidth is shown. In *camera preview*, *image scaling*, and *image blending* workloads, the linear map is better as the overhead is less than 2%. This overhead is insignificant and is because of high TLB hit rates. However, in Fig. 15 (a), the overheads are 34% in *rotated preview* and 50% in *rotated display* when the linear map is used. This means page table walks significantly occupy the memory bandwidth. This is undesirable and is because of low TLB hit rates. The overheads decrease when our tiled maps are used. This means our approach can improve memory utilization. In Fig. 15 (b), the block fetch utilizes the memory better than the single fetch. Even in this case, the memory utilization tends to improve when our selected map is used.

Fig. 13 Execution cycles to run a single image frame. Lower is better.

Fig. 14 TLB hit rates. Higher is better.

Fig. 15 Page table walk bandwidth overheads. Lower is better.

Fourth, we conduct a performance experiment for a 3-plane channel layout. In this layout, Y, U, and V data are stored in separate buffers. A pixel size (*BytePixel*) is 1 B. The targeted address map is determined using the dimensioning algorithm in Fig. 9. In this case, two configuration parameters are affected as follows.

- *PtwCapacity* for the block fetch should be determined by the number of planes. TLB accommodates 16 entries. In the previous experiments, the block fetch acquires 16 page table entries. This is because a single RGB plane was considered. However, when there are multiple planes, *PtwCapacity* for the block fetch should be reduced. In this experiment, as there are 3 planes and TLB has 16 entries, the block fetch acquires 4 page table entries.

- *TileH* should be determined by *BytePixel* and a transaction size. In this experiment, as *BytePixel* is 1 B and a transaction size is 64 B, *TileH* is determined by 64 pixels.

In Fig. 16, the results are shown. Similar to previous experiments, when an access pattern is linear, the linear map is better than others. When an access pattern is not linear, our clustered tiled map is better than the linear map. This is because TLB hit rates are improved when the clustered tiled map is used. The block fetch tends to perform much better than the single fetch. Performance tends to improve when our selected map is used. In Fig. 16, our selected map performs 34% better than others on average.

Finally, we evaluate the hardware cost of the address map component. To measure an area overhead, the logic

Fig. 16 Execution cycles when the number of planes is three. Lower is better. The block fetch acquires 4 page table entries.

Table 7 Hardware area cost of the address maps. The number of slice LUTs (look-up tables) in Xilinx Virtex-7 xc7vx980t.

Max. number of levels	0 [12]	1	2	3
Number of LUTs	914	3057	3234	3905

Table 8 Power consumption (Watts) of the address maps.

Max. number of levels	0 [12]	1	2	3
Static	1.006	1.006	1.006	1.006
Dynamic	0.046	0.103	0.112	0.137
Total	1.052	1.109	1.118	1.143

in Fig. 8 (b) is implemented in Verilog, synthesized, placed, and routed in Xilinx FPGA device. Similar to [12], *TileH* and *TileV* are set by 16. The tile remapping logic operates in 241 MHz clock frequency. In Table 7, the area in the number of look-up tables (LUTs) is shown. When the number of levels is three, our design requires 4.3× more area than [12]. The targeted device contains 612000 LUTs in total. Our design has certain area overheads compared to [12]. Additionally, we evaluate the power consumption overhead. To measure the power consumption (in Watts), the designs are placed and routed. Then Xilinx XPower tool is used. The supplied voltage of the device is 1 volt. In Table 8, the results are shown. The static power is due to the inherent FPGA fabrics. The dynamic power is related to design activities. When the number of levels is three, our design requires 3× more dynamic power than [12]. Our design has certain power overheads compared to [12]. These increased hardware costs would be traded for improved performance.

6. Conclusions

We presented the tile remapping algorithm to construct the configurable address map. An image size in the presented approach can be un-aligned with a tile size. The presented design unifies the tiled map and the conventional linear map. The design allows a system to configure an address map for the access pattern of an application. Three advantages (performance, memory utilization, allocated memory size) and two disadvantages (hardware area, power overheads) are presented. The block fetch can reduce the overheads of the single fetch in terms of TLB hit rate. However, per-

formance still can be degraded depending on address maps and access patterns. By configuring the address map using the adaptive dimensioning algorithm, the page table walk overheads can be reduced.

Acknowledgments

The author would like to thank Dr. Cuong Pham-Quoc (Ho Chi Minh City University of Technology, VNU-HCM) for supporting FPGA experiments.

References

- [1] J.Y. Hur, "Contiguity Representation in Page Table for Memory Management Units," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol.27, no.1, pp.147–158, Jan. 2019.
- [2] B. Pham, A. Bhattacharjee, Y. Eckert, and G.H. Loh, "Increasing TLB Reach by Exploiting Clustering in Page Translations," 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp.558–567, FL, USA, Feb. 2014.
- [3] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp.258–269, Vancouver, Canada, Dec. 2012.
- [4] 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," *Proc. 42nd Annual International Symposium on Computer Architecture - ISCA '15*, pp.66–78, OR, USA, June 2015.
- [5] M. Peemen, B. Mesman, and H. Corporaal, "Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators," *Design, Automation & Test in Europe Conference & Exhibition (DATE'15)*, pp.169–174, Grenoble, France, March 2015.
- [6] I.-J. Sung, N. Anssari, J.A. Stratton, and W.-M.W. Hwu, "Data Layout Transformation Exploiting Memory-Level Parallelism in Structured Grid Many-Core Applications," *Int. J. Parallel. Prog.*, vol.40, no.1, pp.4–24, Feb. 2012.
- [7] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting Address Translation for Accelerator-Centric Architectures," 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp.37–48, TX., USA, Feb. 2017.
- [8] V. Schwambach, S. Cleyet-Merle, A. Issard, and S. Mancini, "Image Tiling for Embedded Applications with Non-Linear Constraints," 2015 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp.1–8, Krakow, Poland, Sept. 2015.
- [9] N. Park, B. Hong, and V.K. Prasanna, "Tiling, Block Data Layout, and Memory Hierarchy Performance," *IEEE Trans. Parallel Distrib. Syst.*, vol.14, no.7, pp.640–654, 2003.

- [10] S. Hettiaratchi and P.Y.K. Cheung, "A novel implementation of tile-based address mapping," *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, pp.306–310, Paris, France, Feb. 2004.
- [11] B. Wang, Y. Fukazawa, T. Kondo, and T. Sasaki, "A Cache Memory with Unit Tile and Line Accessibility," *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pp.866–874, Innsbruck, Austria, July 2016.
- [12] J.Y. Hur, S.W. Rhim, and B.H. Lee, "Method and Apparatus for Performing Adaptive Memory Bank Addressing," US Patent 9,390,007, July 2016.
- [13] S.M. Hong, J.Y. Hur, and M.J. Jun, "Method of Operating Virtual Address Generator and Method of Operating System Including the Same," US Patent filed number 20170140501A1, May 2017.
- [14] S.M. Hong, S.J. Lee, J.Y. Hur, J.W. Kwon, I. Park, J.J. Lee, and J.Y. Jung, "System-on-chip and Address Translation Method Thereof Using a Translation Lookaside Buffer and a Prefetch Buffer," US Patent 9,645,934, May 2017.
- [15] ARM Architecture Reference Manual, ARMv7-A edition, available online: https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf (Accessed 20 May 2014).

Jae Young Hur received a B.S. degree in electronics engineering from Cheju National University, Cheju, South Korea, in 1995. He received M.S. degrees in electronics engineering from Sogang University, Seoul, South Korea, in 1998 and from Munich University of Technology, Munich, Germany, in 2002. He received a Ph.D. degree in computer engineering from Delft University of Technology, Delft, The Netherlands, in 2011. He was an engineer (from 1999 to 2000) and a senior engineer (from 2008

to 2016) in the semiconductor division, Samsung Electronics, Ltd., South Korea. Currently, he is a researcher in Vietnamese-German University, Binh Duong, Vietnam. His research interests include embedded system architectures, VLSI design, and reconfigurable computing.