Memory Management for Dual-addressing Memory Architecture

Ting-Wei Hung s1016004@mail.yzu.edu.tw

Yen-Hao Chen s971433@mail.yzu.edu.tw

Yi-Yu Liu yyliu@saturn.yzu.edu.tw

Department of Computer Science and Engineering, Yuan Ze University
Chungli, Taiwan 320, R.O.C.

Abstract— Dual-addressing memory architecture is designed for two-dimensional memory access with both row-major and column-major localities. In this paper, we highlight two memory management issues in dual-addressing memory. First, to avoid the external fragmentation, we propose a virtual dual-addressing memory design to enable memory management via operating system. After that, to deal with the size mismatch between user-defined data and dual-addressing memory, we discuss data arrangement policies at different data granularity. With the proposed memory management techniques, we are capable of maximizing the memory utilization of dual-addressing memory.

I. Introduction

With the increasing latency gap between dynamic random access memory (DRAM) and logic, DRAM has become one of the most critical performance bottlenecks in a computing system [1]. DRAM is conceptually considered as a one-dimensional array with one serial address for each memory cell. To maintain a feasible aspect ratio for memory chip fabrication, the DRAM organization is composed of a regular two-dimensional memory cell array and two orthogonal address decoding circuits. Consequently, the serial addresses obtained from address decoding circuits define neighborhood structure of a DRAM. Once the neighborhood structure of DRAM is fixed, the access latency of DRAM cells are determined according to the spatial locality of the pre-defined neighborhood structure.

Dual-addressing memory organization is proposed to support two-dimensional memory access patterns [2]. Specialized memory organizations for specific memory access patterns are imperative for high performance computing. In this work, translations from virtual dual-addressing memory to physical dual-addressing memory is proposed. The translation mechanism enables memory management via operating system. Furthermore, to make use of the dual-addressing memory in a generic computing system, the size mismatch issue between user-defined data and dual-addressing memory is discussed. With the proposed memory management techniques, we are capable of maximizing the memory utilization of dual-addressing memory.

The rest of this paper is organized as follows. The preliminary of commodity memory organization and dual-addressing memory organization are discussed in Section II. In Section III, we propose virtual dual-addressing memory to tackle the external fragmentation. Data arrangement at different data granularity is discussed in Section IV. Section V concludes this paper and points out important issues for future research.

II. Preliminary

In this section, we give preliminary background of commodity memory organization and dual-addressing memory organization.

A. Conventional Memory Organization

Commodity DRAM utilizes one transistor and one capacitor for one data bit storage. To maintain the DRAM chip form factor as well as decoding circuit efficiency, memory address is partitioned into row address and column address. Conventionally, the higher address bits are denoted as row address and the lower address bits are denoted as column address. Each memory access requires both row and column access phases. Figure 1 draws the organization of a conventional memory. Once the memory address is ready, the row decoder decodes the row address and asserts a corresponding word line. The memory data sets on the word line, refer to a page, are then ready to be processed in the column access phase. According to the column address, sense amplifier and column decoder are activated to detect and select the corresponding memory data set of the page, respectively [3]. In the memory organization drawn in Figure 1, data sets within the same page can be directly accessed by column address decodings only with short access time while data sets from different pages require the assertions of different word lines followed by column address decodings with long access time. Therefore, the sequence of row access followed by column access explicitly defines the horizontal neighborhood structure of a DRAM. The horizontal neighborhood structure favors one-dimensional data array and multi-dimensional data array with row-major memory access patterns.

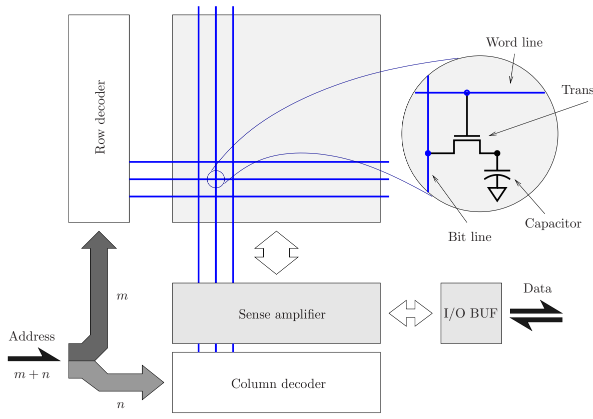


Fig. 1. Conventional memory organization.

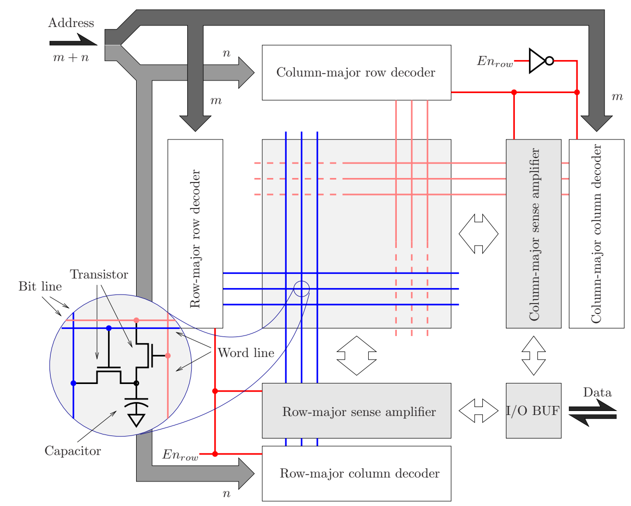


Fig. 2. Dual-addressing memory organization.

B. Dual-addressing Memory Organization

Figure 2 depicts the organization of dual-addressing memory. There are two transistors and one capacitor for one data bit storage. Unlike the dual-ported memory which supports two consecutive memory accesses from the same address decoding architecture [4], the dual-addressing memory utilizes two address decoding architectures. In Figure 2, two word lines and two bit lines are orthogonal to each other, respectively. Hence, there are two pairs of decoding and sensing circuits in dual-addressing memory. The address decoding scheme, which is the same to conventional memory drawn in Figure 1, is denoted as row-major decoding. The address decoding scheme, which is orthogonal to conventional row-major decoding, is denoted as column-major decoding. As drawn in Figure 2, there are $m+n$ address bits with enable signal En_{row} to select one decoding scheme. The row-major decoding decodes the upper m bits and asserts a corresponding horizontal word line before multiplexing vertical bit lines by the lower n bits. The column-major decoding decodes the lower n bits and asserts a corresponding vertical word line before multiplexing horizontal bit lines by the upper m bits. Notice that the upper and lower address bits can be the same ($m = n$) in practice such that the decoder and sense amplifier can be shared by both addressing schemes. Without loss of generality, we assume $m \neq n$ in this paper.

To utilize the dual-addressing memory, programmers are required to specifically declare a dual-addressable (two-dimensional and multi-dimensional) data structure. The data structure will then be bound to the dual-addressing memory after compilation. There are four types of new instructions required to support dual-addressing memory access. These instruction types are row-major read, row-major write, column-major read, and column-major write. The row-major read and write instructions behave as read and write instructions for conventional memory, respectively. The column-major read and write instructions swap the upper and lower address bits before performing column-major word line decoding and bit line multiplexing.

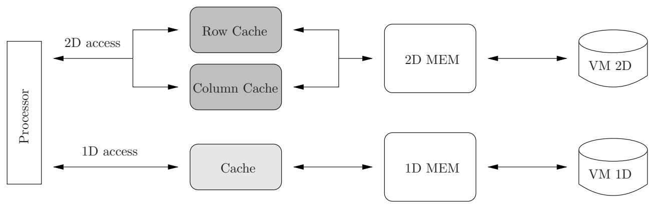


Fig. 3. Proposed memory system architecture.

Table I lists row-major and column-major decoding results with $m = 2$ and $n = 3$. The first line in each entry represents the decoded address of row-major decoding (colored in red). The second line in each entry represents the decoded address of column-major decoding (colored in blue). From Table I, it is clear that each memory data set has row-major and column-major addresses for row-major and column-major memory accesses, respectively. Consequently, row-major memory access favors horizontal neighborhood structure while column-major memory access favors vertical neighborhood structure. Therefore, the dual-addressing memory is capable of maintaining spatial locality in two-dimensional memory access patterns.

Since the memory density of dual-addressing memory is lower than that of conventional DRAM, it is inefficient to replace the whole DRAM by using dual-addressing. Hence, we integrate a small dual-addressing memory into a computing system to support two-dimensional memory access behaviors. Figure 3 illustrates the proposed memory system architecture.

TABLE I
Row-major and column-major decodings

	000	001	010	011	100	101	110	111
00	0 (00000) 0 (00000)	1 (00001) 4 (00100)	2 (00010) 8 (01000)	3 (00011) 12 (01100)	4 (00100) 16 (10000)	5 (00101) 20 (10100)	6 (00110) 24 (11000)	7 (00111) 28 (11100)
01	8 (01000) 1 (00001)	9 (01001) 5 (00101)	10 (01010) 9 (01001)	11 (01011) 13 (01101)	12 (01100) 17 (10001)	13 (01101) 21 (10101)	14 (01110) 25 (11001)	15 (01111) 29 (11101)
10	16 (10000) 2 (00010)	17 (10001) 6 (00110)	18 (10010) 10 (01010)	19 (10011) 14 (01110)	20 (10100) 18 (10010)	21 (10101) 22 (10110)	22 (10110) 26 (11010)	23 (10111) 30 (11110)
11	24 (11000) 3 (00011)	25 (11001) 7 (00111)	26 (11010) 11 (01011)	27 (11011) 15 (01111)	28 (11100) 19 (10011)	29 (11101) 23 (10111)	30 (11110) 27 (11011)	31 (11111) 31 (11111)

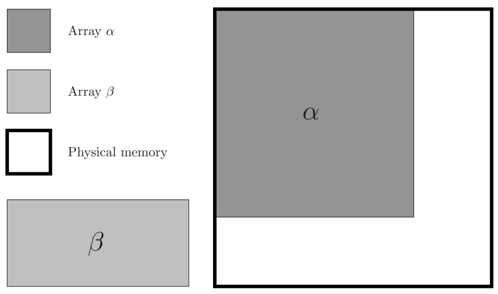
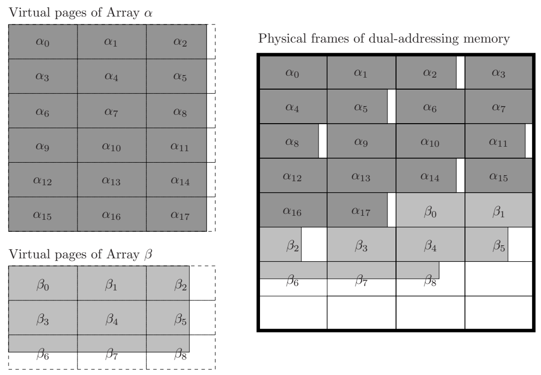


Fig. 4. External fragmentation in dual-addressing memory.


Virtual pages of Array α
Virtual pages of Array β
Physical frames of dual-addressing memory
α_0 … α_{17}
β_0 … β_8
Fig. 5. Paging for dual-addressing memory.

III. Virtual Memory Design for Multiple Dual-addressing Memory Arrays

In a typical computation scenario, there are multiple two-dimensional memory arrays declared. Consequently, all arrays with column-major memory access behaviors are required to be allocated in dual-addressing memory in order to achieve less memory access latency. As a result, multiple dual-addressing memory arrays share a single physical dual-addressing memory. Figure 4 shows an example. There are two dual-addressing memory arrays, α and β. Array α is allocated at the top-left corner in dual-addressing memory. Owing to the limitation of geometrical shape, array β is unable to be allocated in the dual-addressing memory even if the size of unused dual-addressing memory is enough for array β.

As illustrated in Figure 4, the dual-addressing memory management for multiple arrays is reducible to a conventional fixed-outline floorplanning problem in electronic design automation [5]. To tackle this hard problem, virtual dual-addressing memory is proposed in this section to facilitate memory management in operating system level.

We define the dual-addressing page as a minimum memory block managed by operating systems. For simplicity, the page dimensions are power of 2 in both row and column directions. Furthermore, the physical dual-addressing memory is partitioned into multiple frames according to the page dimensions. All data stored in a page/frame is dual-addressable. Figure 5 illustrates the idea of dual-addressing paging. In Figure 5, arrays α and β are subdivided into small virtual pages and allocated in physical dual-addressing memory frames.

Unlike conventional virtual memory system which requires only one address translation from virtual space to physical space [6], virtual dual-addressing memory system requires two address translations to ensure data localities in both row-major and column-major memory accesses: (1) user-defined array space to virtual space and (2) virtual space to physical space.

Given a user-defined memory array μ with width U_W and height U_H, the user-defined array address of indices U_x and U_y can be derived by Equation 1. The U_{addr} cannot be used for dual-addressing memory access since U_W and U_H may not be power of 2.

$$U_{addr} = U_y \times U_W + U_x \qquad (1)$$

Therefore, the user-defined array is partitioned in two-dimensional manner according to the predefined geometrical page shape. In our formulation, the page width and page height are denoted as P_W and P_H, respectively. To obtain the virtual page number of the data $\mu[U_y][U_x]$, we use Equations 2 and 3 to calculate the location of partitioned page in X and Y coordinates, denoted as X_{page} and Y_{page}, respectively. Similar to Equation 1, the exact virtual page number can be derived by Equation 4.

$$X_{page} = \lfloor \frac{U_x}{P_W} \rfloor \qquad (2)$$

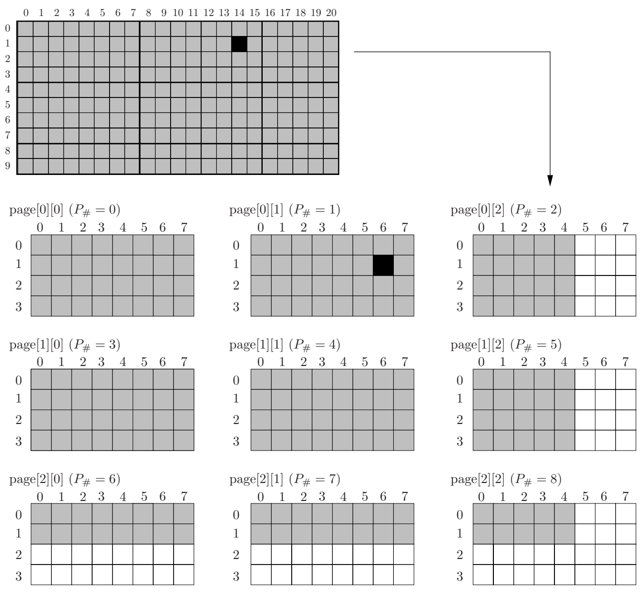


Fig. 6. Translation of user-defined array address to virtual address.

$$Y_{page} = \lfloor \frac{U_y}{P_H} \rfloor \quad (3)$$

$$P_{\#} = Y_{page} \times \lceil \frac{U_W}{P_W} \rceil + X_{page} \quad (4)$$

In page $P_{\#}$, the column offset and row offset can be derived by Equations 5 and 6, respectively. Finally, the virtual address of $\mu[U_y][U_x]$ is obtained by concatenating the page number, page row offset, and page column offset in Equation 7.

$$P_x = U_x \bmod P_W \quad (5)$$

$$P_y = U_y \bmod P_H \quad (6)$$

$$V_{addr} = P_{\#} \| P_y \| P_x \quad (7)$$

Figure 6 shows a translation example of the aforementioned array β. The dimension of array β is 10×21 and the dimension of each page is 4×8. There are $\lceil \frac{21}{8} \rceil = 3$ pages in X coordinates and $\lceil \frac{10}{4} \rceil = 3$ pages in Y coordinates. The array data $\beta[1][14]$ is partitioned in page[0][1] since $X_{page} = \lfloor \frac{14}{8} \rfloor = 1$ and $Y_{page} = \lfloor \frac{1}{4} \rfloor = 0$. Therefore, the page number is $P_{\#} = 0\times3+1 = 1$. In page[0][1], the page column and row offsets of $\beta[1][14]$ are $P_x = 14 \bmod 8 = 6$ and $P_y = 1 \bmod 4 = 1$, respectively. Finally, the virtual address of $\beta[1][14]$ is $V_{addr} = 1\|1\|6 = (000101110)_2$.

Once the virtual address is obtained, we can use the page table to map a virtual page to its corresponding physical frame. In order to maintain row-major and column-major localities, the location of each physical frame is characterized by X and Y coordinates, denoted as X_{frame} and Y_{frame}. Finally, the row-major and column-major physical addresses are summarized in Equations 8 and 9, respectively.

$$P^{row}_{addr} = Y_{frame} \| P_y \| X_{frame} \| P_x \quad (8)$$

$$P^{column}_{addr} = X_{frame} \| P_x \| Y_{frame} \| P_y \quad (9)$$

Figure 7 shows a translation example from virtual address to physical address according to the result of Figure 6. Since the page number is $(0001)_2$, the corresponding physical frame X and Y coordinates are $X_{frame} = (10)_2$ and $Y_{frame} = (011)_2$, respectively. The row-major and column-major physical addresses are obtained by shuffling the four fields, Y_{frame}, X_{frame}, P_y, and P_x.

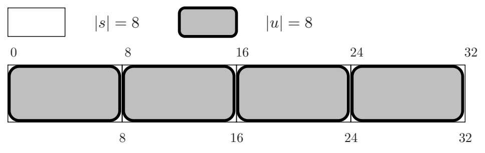


Fig. 8. Example of $|u| = |s|$.

IV. Data Arrangement at Different Granularity

According to Table I, there are two addresses (row-major and column-major) associate with each memory data set. Therefore, the size of dual-addressing memory data set is fixed. Since the size of user-defined data element, denoted as $|u|$, may not be same to that of dual-addressing memory data set, denoted as $|s|$, we need to select suitable data arrangement policy to access user-defined data efficiently. Figure 8 illustrates a regular memory-access behavior when $|u| = |s|$.

If there is a size mismatch between user-defined data and dual-addressing memory data set (i.e., $|u| \neq |s|$), we can use the minimum number of dual-addressing memory data sets as a stride (i.e., $\lceil \frac{|u|}{|s|} \rceil$ memory data sets) to accommodate the user-defined data. The advantage of regular stride simplifies memory access at the expense of padding unused memory. Figure 9 shows two padding examples. As shown in Figure 9, regular stride may result in considerable memory wastage.

To avoid memory wastage, we can concatenate all user-defined data elements as a whole. However, the user-defined data offsets within a dual-addressing memory data set vary on data indices. As a result, the irregular offset complicates dual-addressing memory access. Taking Figure 10(a) as an example, the first and the second data elements require only one memory access while the third data element requires two memory accesses. Similarly, the first data element require two memory accesses while the second data element require three memory accesses in Figure 10(b).

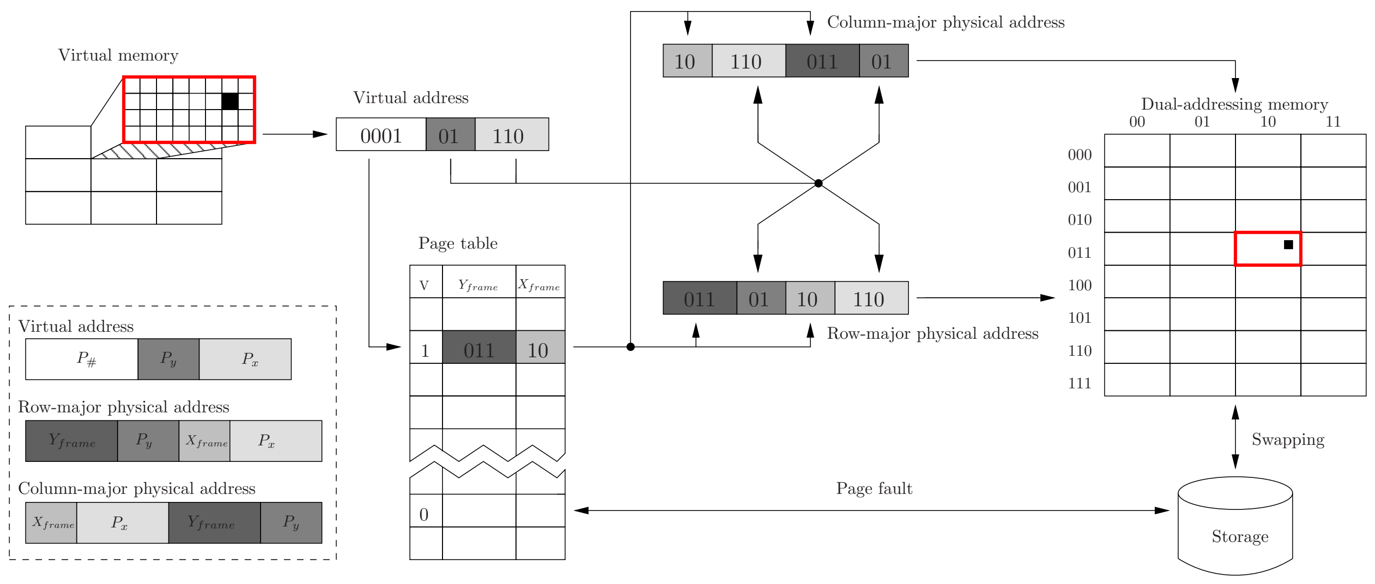


Fig. 7. Translations of virtual address to physical address.

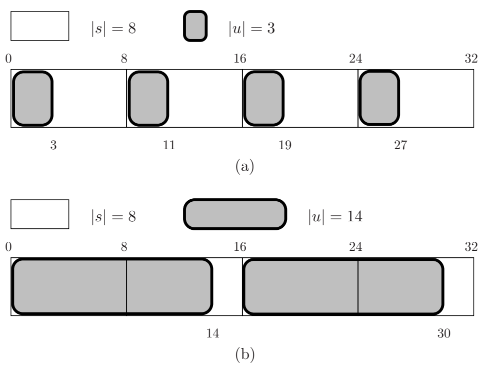


Fig. 9. Example of padding policy.

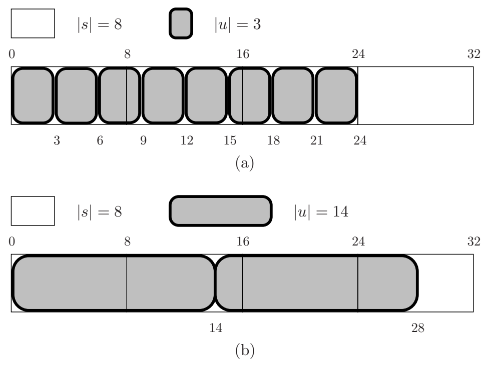


Fig. 10. Example of concatenating policy.

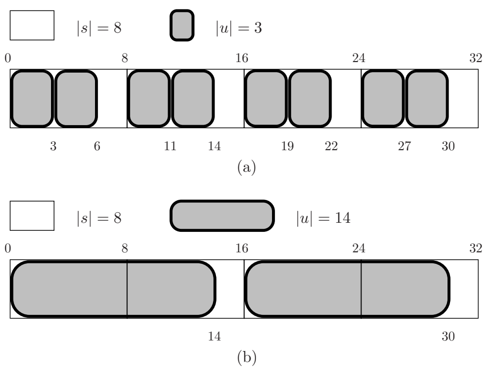


Fig. 11. Example of hyper-padding policy.

Accordingly, we suggest hyper-padding to resolve the size mismatch problem. For the case of $|u| < |s|$, we concatenating $\lfloor \frac{|s|}{|u|} \rfloor$ user-defined data elements within each dual-addressing memory data set to minimize memory wastage. For the case of $|u| > |s|$, we use $\lceil \frac{|u|}{|s|} \rceil \times |s|$ as a stride and pad the unused memory to avoid irregular memory-access situation. Figure 11 draws two hyper-padding examples. In Figure 11(a), concatenating policy is applied within each dual-addressing memory data set. In Figure 11(b), padding policy is applied to avoid irregular memory-access situation. Table II summarizes the padding, concatenating, and hyper-padding policies for different data granularity.

V. Conclusions and Future Work

We have proposed memory management techniques to support two-dimensional memory access behaviors for

TABLE II
Summary of address computations

Size	Padding		Concatenating		Hyper-padding	
	Wastage (%)	Regularity	Wastage (%)	Regularity	Wastage (%)	Regularity
$\|u\| = \|s\|$	0	Yes	0	Yes	0	Yes
$\|u\| < \|s\|$	$\frac{\|u\|-\|s\|}{\|s\|}$	Yes	0	No	$\frac{\|s\|-\lfloor\frac{\|s\|}{\|u\|}\rfloor\times\|u\|}{\|s\|}$	Yes
$\|u\| > \|s\|$	$\frac{\lceil\frac{\|u\|}{\|s\|}\rceil\times\|s\|-\|u\|}{\lceil\frac{\|u\|}{\|s\|}\rceil\times\|s\|}$	Yes	0	No	$\frac{\lceil\frac{\|u\|}{\|s\|}\rceil\times\|s\|-\|u\|}{\lceil\frac{\|u\|}{\|s\|}\rceil\times\|s\|}$	Yes

dual-addressing memory architecture. In operating system level, virtual dual-addressing memory is designed to facilitate memory management and to avoid the external fragmentation. After that, to deal with the size mismatch between user-defined data and dual-addressing memory, we suggest hyper-padding data arrangement policy for different granularity. With the proposed memory management techniques, we are capable of maximizing the memory utilization of dual-addressing memory.

References

[1] D. A. Patterson, J. L. Hennessy, "Computer organization and design: the hardware/software interface", Morgan Kaufmann, 2011.

[2] Y. H. Chen, Y. Y. Liu, "Dual-addressing memory architecture for two-dimensional memory access patterns", *in Proceedings of Design Automation and Test in Europe*, pp.71-76, 2013.

[3] B. Jacob, S. Ng, D. Wang, "Memory systems: cache, dram, disk", Morgan Kaufmann, 2007.

[4] N. Weste, D. Harris, "CMOS VLSI design: a circuits and systems perspective", Addison Wesley, 2010.

[5] L. T. Wang, Y. W. Chang, K. T. Cheng, "Electronic design automation: synthesis, verification, and test", Morgan Kaufmann, 2009.

[6] A. Silberschatz, P. B. Galvin ,and G. Gagne, "Operating system concepts", John Wiley & Sons, 2008.