Array Allocation Taking into Account SDRAM Characteristics †

Hong-Kai Chang Youn-Long Lin
Department of Computer Science
National Tsing Hua University
Hsinchu, 300, Taiwan, R.O.C

Abstract- Multimedia, image processing and other signal processing applications often involve data stored in large arrays. Due to chip area limitation, arrays are typically assigned to off-chip memories, such as DRAM. This being the case, we try to optimize off-chip memory accesses to improve performance. We take the characteristics of the current mainstream SDRAM memory into account. We propose an algorithm to allocate arrays to different banks to increase the probability of utilizing SDRAM's multi-bank characteristic. Experimental results show significant improvement over traditional approaches.

I. INTRODUCTION

Memory system has become the bottleneck of system performance nowadays. In order to close the performance gap between memory and processor, cache memories are often introduced. They are suitable for general-purpose computation, but not for multimedia DSP applications. The memory subsystem often needs to be customized in order to fit the application specific requirement.

The trend of embedding DRAM into the chip relieves the bottleneck between main memory and processor. This being the case, the existence of on-chip cache should be reconsidered because it often takes up a great portion of the chip area. If the chip is dedicated to a special application, we can try to remove cache memories, thus reduce the area cost and control complexity.

In this paper, we focus on system chips without cache - the performance of DRAM plays an important role. The development of DRAMs, from the old FPM (Fast Page Mode) DRAM [10], to its successor EDO (Extended Data Out) DRAM [12], and the current mainstream SDRAM (Synchronous DRAM) [3], even Rambus DRAM [13] in the future, has made significant improvements in speed. If we can fully utilize their special features, the overall system performance can be greatly improved. We will discuss how to allocate arrays to make a better scheduling by taking the SDRAM's characteristics into account.

II. RELATED WORK

There are many topics on alleviating the bottleneck between memory and processor but only a few researchers consider the utilization of off-chip memory characteristics. Panda, Dutt, and Nicolau proposed incorporating EDO DRAM access model into high-level synthesis [1]. Their main object is to utilize the page mode access. They modeled EDO DRAM's memory access modes for high level synthesis and proposed algorithms to incorporate them. Several techniques are used to transform input behavior for further optimizations. The memory controller must supply individual DRAM commands, such as row decode, column decode, and precharge, so that the user program can control the opening or closing of DRAM pages.

On-chip memories, like cache and scratch pad memory, are used to improve memory system performance. Exploration and optimization of local memory in an embedded system is discussed in [5]. An analytical estimation to tailor on-chip memory configuration is proposed. The impact of on-chip memory size, partition of cache and scratch pad memory, and cache line size, are discussed.

Prefeching techniques are introduced to hide memory latencies. The general idea is to prefetch data as soon as possible if there are no data dependencies. The authors of [6] proposed a technique for cache coherent processors by using low-overhead cache miss traps.

Another topic that affects the system performance and cost is array mapping to physical memories. It involves the memory configuration used and the grouping and binding of arrays to memory components. For example, mapping arrays to fast but expensive memory improves the performance with higher cost. A more detail discussion on this subject can be found in [7]. In our work, we map arrays to different memory banks.

III. MOTIVATION

A. DRAM operations

The address of a DRAM word is divided into two parts: row address and column address. They must be provided sequentially to the memory. Fig.1 shows the addressing of a traditional DRAM and a 2-bank SDRAM. Data with different row addresses belong to different memory *pages*. For SDRAM, pages with different bank addresses (part of row address) belong to different memory *banks*.

There are three phases when accessing a DRAM: row decode, column decode, and precharge. The row decode phase provides row address, and the column decode phase provides column address with the write enable signal indicating whether the access is a read or a write. The required operation is then performed. The precharge command is performed if the succeeding access is to a different page. We call the situation when the succeeding access refers to the currently open page as *page hit* and otherwise *page miss*. A page miss causes the precharge and row decode commands and is much slower than a page hit. Note that for an SDRAM, each bank has its own open page and precharging one bank will not affect open pages of other banks.

† Supported in part by the National Science Council, R.O.C, under contract no. NSC 89-2215-E-007-005

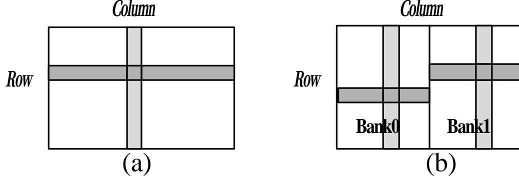

Fig. 1. Addressing for (a) a traditional DRAM (b) a 2-bank SDRAM

B. SDRAM' s multi-bank architecture

An SDRAM has multiple pages, often two or four. Each bank has its own open page independent of the other(s). For example, the 2-bank SDRAM shown in Fig.1 (b) can have two open pages. The old EDO DRAM can have only one page open. Thus, using SDRAM, the probability of page hit is higher and it increases with the number of banks.

C. Motivational example

Here we use an example to illustrate the advantages of multiple-bank over single-bank architecture. Assume that we have three memory controllers. Controller I can utilize SDRAM' s multiple-bank feature but Controller II can not. Controller III is also aware of multiple-bank but always performs precharge after a read/write access. Controller I and III show the behaviors of SDRAM controllers and Controller II shows the behaviors of EDO DRAM controller.

Assume that Data1 and Data3 are in the same page of a bank, and Data2 and Data4 are in another page of another bank. The access sequence is Data1, Data2, Data3, and then Data4. Fig.2, Fig.3 and Fig.4 illustrate the behavior of the three controllers. Controller I needs 10 cycles in this case while controller II needs 27 cycles. Controller III is slower than Controller I because it always performs precharge after read/write access, but faster than Controller II because it takes advantage of bank interleaving access.

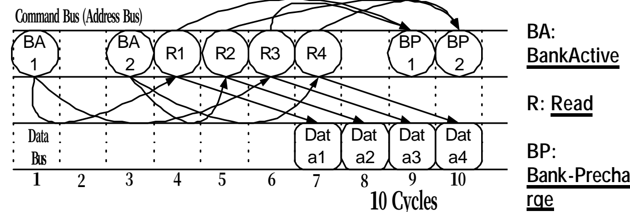

Fig. 2. Interleavingly reading four data from two pages (Controller I)

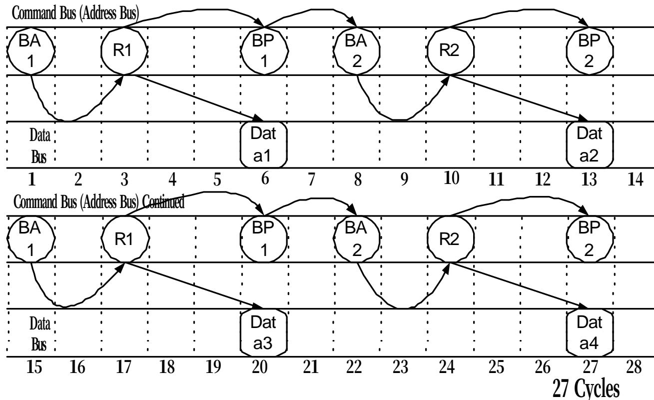

Fig. 3. Interleavingly reading four data from two pages (Controller II)

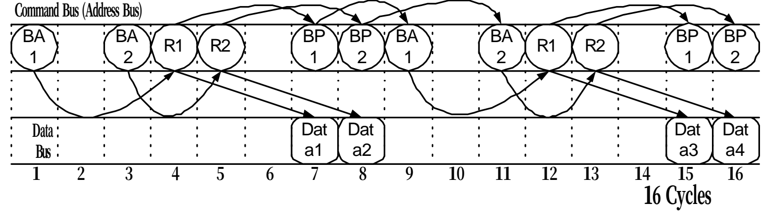

Fig. 4. Interleavingly reading four data from two pages (Controller III)

In the three figures, "BA" represents a bank active command of SDRAM, which sends row address. It equals to EDO DRAM' s row decode command. "R" represents a read command, which sends column address and read/write signal. It equals to EDO DRAM' s column decode command. "BP" represents a bank precharge command, which precharges a bank, and equals to EDO DRAM' s precharge command.

Note that two consecutive BA commands for different banks must be separated by at least 2 cycles. After an R command is performed, data is ready 3 cycles later. After a BP command is performed, the corresponding bank does not accept any new command for 2 cycles. Accesses can be pipelined and we can send commands for new accesses to the SDRAM without waiting for the completion of the current access.

IV. ARCHITECTURE CONSIDERATIONS

A. System architecture

Both the address mapping from logical to physical memory and the processor instruction set influence the schedule and address assignments. Although page size is decided by DRAM' s column address, the mapping from host address (in processor) to memory address (in DRAM) may cause the size of a continuous host addressing space within page smaller than DRAM' s actual page size.

For example, we make a mapping for a 17-bit host address to a 9x8 (Row address 9bits x Column address 8bits) SDRAM memory address, shown in Table 1. The actual page size is 256 words (because the Column address has 8 bits) but in processor' s view the page size is 128 words because while a7 changes, the address maps to another row. In the table, "BA", "A7"~"A0" represent memory address pins. "a16"~"a0" represent host address pins. Row and Column Address share the memory address pins "BA"~"A0". The pin labeled with "BA" is the Bank Active pin, which selects the bank of SDRAM.

TABLE 1
A 9x8 SDRAM ADDRESS MAPPING TABLE

	BA	A7	A6	A5	A4	A3	A2	A1	A0
Row.Addr	a7	a16	a15	a14	A13	A12	a11	a10	a9
Col.Addr		a8	a6	a5	A4	A3	a2	a1	a0

For SDRAM, the mapping of Bank Select pin(s) decides the interleaving size of banks. For the example in Table 1, the bank interleaving size is 128 words. If we exchange the mapping of a7 and a0, the mapping becomes word interleave.

The programmer and compiler can take the advantages of different memories by assigning different variables into corresponding host addresses if they know the address mapping from host to physical memory. If the processor provides instructions to open/close single memory page, the programmer can have additional opportunities to optimize memory access.

B. Paging policies

Paging policies of the DRAM controller also influence the system performance. If the controller keeps the memory page open until a page miss happens, we can benefit from the faster page mode access (like Fig.2). If the controller always performs precharge after a read/write access, there is no chance to utilize page mode because it always has to active the bank/page again before any new access. But, some optimizations may be still possible for SDRAM, such as interleaving access among banks (like Fig.4).

V. SOLVING THE PROBLEM

A. System architecture

We assume the system is based on the Harvard Architecture, which has separated program and data memory. Thus we can focus on the access of data from the memory, without the interference of fetching instructions. We have tried experiments on a PC, which is a non-Harvard architecture, but the results are not always as expected because program fetching and system interrupts all impact the memory access sequence.

The memory controller behaviors like Intel's BX chipsets [2]. It is aware of the existence of banks, perform precharge command only at page miss, and can have as many pages open at once as the number of SDRAM banks.

B. Problem inputs/outputs

1. Inputs

i) A data flow graph (DFG), whose nodes represent operations, and edges represent data dependencies. It defines the program behavior. ii) Resource constraints, which specify the number of function units, the number of clock cycles needed to complete the operation, and the type of the function units. iii) Memory system configurations, such as timing constraints of the SDRAM, the number of banks, and the number of DRAM modules.

2. Processing steps

i) Read the input data and create internal data structures. ii) Call our bank allocation algorithm to allocate each array to a suitable bank. iii) Call a static list scheduling algorithm to schedule the operations, with consideration of SDRAM's timing constraints.

3. Outputs

i) The scheduling results, including function units used and nodes scheduled at every time step. Every loop is scheduled for one iteration here. ii) Total cycle counts, taking into account loop iterations. iii) Bank allocation table, which specifies the mapping of arrays to SDRAM banks.

C. Timing verification

In order to verify that our scheduling results meet SDRAM timing constraints, the scheduler generates a Verilog file that contains SDRAM commands for each clock cycle in the schedule. Simulated with an SDRAM simulation model compatible with Intel's PC SDRAM spec, our scheduling results indeed meet the SDRAM timing constraints.

D. DRAM refresh

The need of refreshing periodically is one important property of all DRAMs. But from Panda's work, we know that the effect of refresh to the performance is quite modest. For simplicity's sake, our work does not take this property into account and modification to the scheduling need to be done in the succeeding step.

VI. BANK ALLOCATION ALGORITHM

A. Definition

Our bank allocation algorithm is based on an *array distance table.* The table contains the distance from one array-accessing node to another in the DFG. An array pair (*X*,*Y*) with short distance means that the accesses to array *X* and array *Y* are strongly related. They may be two input sources of an operator, such as an "+" or "*". The operator "+' or "*" can be scheduled only after its two operands are fetched from the memory. This being the case, we can allocate them to different banks, thus shortens the time needed to fetch them both. On the other hand, an array pair (*Z*,*W*) with long distance means that accesses to *Z* and *W* are separated by several operations, and it's not urgent to access them concurrently. So, they can be allocated to the same bank, without increasing the schedule length.

B. Calculating array distance

We use the *Successive Over-Relaxation algorithm* (SOR.C) shown in Fig.5 to illustrate how to calculate array distance. There are 7 two-dimensional arrays in the example. We assume that a memory page is large enough to contain any row of any array. For each of array *a-f,* only one row is accessed per iteration. For array *u,* three rows (*i-1, i, i+1*) are accessed per iteration. Therefore, for each loop iteration, accessing array *u* involves three DRAM pages, while accessing array *a-f* involves one DRAM page each.

```
main()
{
 float a[N][N], b[N][N], C[N][N], d[N][N], e[N][N], f[N][N];
 float omega, resid, u[N][N];
 int j,l;

for (j=2; j<N; j++)
  for (l=1;l<N;l+=2) {
        resid=a[j][l]*u[j+1][l]+
                      b[j][l]*u[j-1][l]+
                      c[j][l]*u[j][l+1]+
                      d[j][l]*u[j][l-1]+
                      e[j][l]*u[j][l] –
                      f[j][l];
      u[j][l]  -= omega*resid/e[j][l];
    }
}
```

Fig.5. SOR.C

Before calculating the array distance, we must know the ***node distance***. For each node, the distance from it to its nearest array-accessing node is called node distance. Fig.6 shows the DFG of SOR.C. The node distances are shown in the parentheses of each node in the following order: *a[i]* to *f[i]*, *u[i]*, *u[i+1]*, *u[i-1]*. A '-' means that the distance to the corresponding array has not been determined yet.

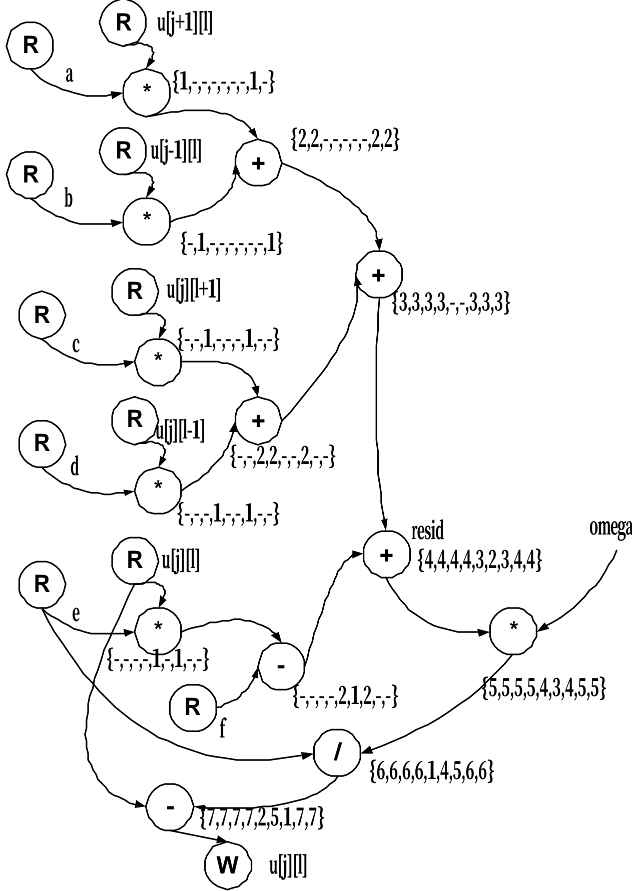

Fig. 6. DFG of the SOR example

Algorithm for calculating the node distances is proposed in Fig.7. After node distances are known, ***array distance*** (*x*, *y*) is calculated by:

```
Algorithm CalNodeDistance(node) {
//Input: The DFG, and the starting node
//Output: Distance to arrays for each succeeding node
//Initially, traverse from the top nodes of the DFG
 for (each succeeding node) do
   begin
     for (each distance to an array) do
       begin
          if (the distance == '-') //not determined yet
          //propagate the information to succeeding node
            update the distance as current node's +1;
          else
            //maintain the minimum value
            update the distance as min(original distance,
                                  current node's distance+1);
       end
     CalNodeDistance(succeeding node); //traverse down
   end }
```

Fig. 6. Algorithm for calculating node distance

1) Initially, all array distances are set to positive infinity.
2) For each node, add the sum of the node distances to *x* and *y* if both distances are determined.
3) If the result in 2) is smaller than the existing array distance (*x*, *y*), update the distance to the new value.

The resulting array distance table of the SOR example is shown in Table 2.

TABLE 2
ARRAY DISTANCE TABLE FOR SOR

	a[i]	b[i]	c[i]	d[i]	e[i]	f[i]	u[i]	u[i+1]	u[i-1]
a[i]	0	4	6	6	7	6	6	2	4
b[i]	4	0	6	6	7	6	6	4	2
c[i]	6	6	0	4	7	6	2	6	6
d[i]	6	6	4	0	7	6	2	6	6
e[i]	7	7	7	7	0	3	2	7	7
f[i]	6	6	6	6	3	0	3	6	6
u[i]	6	6	2	2	2	3	0	6	6
u[i+1]	2	4	6	6	7	6	6	0	4
u[i-1]	4	2	6	6	7	6	6	4	0

C. Allocating arrays to memory banks

After the array distance table is created, the bank allocation algorithm below is performed. It allocate arrays with strong relation (thus with short array distance) to different DRAM banks, to maximize the possibility of concurrent access.

1) Find the smallest distance in the table that has not been traversed and get two arrays x and y.
2) If both array x and y have not been assigned
   Choose a least used bank and assign array x to it.
3) If either x or y has not been assigned
   Except the bank that either x or y has been assigned to, choose a least used bank and assign it to the array that is not assigned.
4) Goto 1) until all arrays are assigned.

Note that once we allocate a row of an array to a bank, the other rows of the same array are allocated accordingly. For example, once row *u*[*i+1*] is assigned to bank 1 in a 2-bank configuration, row *u*[*i*] is accordingly assigned to bank 0, row *u*[*i-1*] to bank 1, and so on.

For our SOR example, we get from Table 2 that (*a, u[i+1]*), (*b, u[i-1]*), (*c, u[i]*), (*d, u[i]*), (*e, u[i]*) all have the smallest array distance of 2 (0s in the diagonal line, represents distance to the array itself, are of course ignored). It tells us that we should allocate row *u*[*i*] in a bank different from row [*i*] of array *c*, *d*, *e* and row *u*[*i+1*] from row *a*[*i*], row *u*[*i-1*] from row *b*[*i*], respectively. Assume that there are 2 banks, our choice is to assign row [*i*] of *c*, *d*, *e* to bank 0, and row [*i*] of array *u* to bank 1. Now, since row *u*[*i*] is assigned to bank 1, row *u*[*i-1*] and row *u*[*i+1*] are assigned to bank 0, accordingly. Thus, row [*i*] of array *a* and *b* are assigned to bank 1. Next, array (*f, u*) got the distance of 3, and since array u is assigned to bank 1, we assign array f to a different bank, that is bank 0. Therefore all arrays are allocated and our algorithm is terminated.

VII. EXPERIMENTAL RESULTS

We choose several benchmark examples that process large data stored in arrays to explore the effect of our work. The experiments were run on different memory configurations.

A. *Benchmark characteristics*

The benchmark "dhrc," "compress," "laplace," "sor," and "lowpass" are taken from the high-level synthesis design repository [4]. They are "differential heat release computation, " "image compress scheme," "Laplace algorithm," and "low pass filter for image," respectively. "Dequant" is the dequantization routine of the MPEG decoder taken from [8]. "Leafcomp" is also from the MPEG decoder application. "Mmult" is a matrix multiplication routine from Panda's paper [1]. "Fir" is taken from a text book [9]. "Wiener", "dct", and "sobel" are taken from another text book [11].

We partition the benchmarks into two groups. Benchmarks in the first group access multiple one-dimensional arrays. Those in the second group access single two-dimensional array. SOR is a special case that accesses multiple two-dimensional arrays. It is placed in the first group, however.

For the first group, our bank allocation algorithm is applied on each array. The address mapping is assumed to allow each array to be entirely contained in one bank. As for the second group, the algorithm is applied to each row of the array. For example, *A[i][j]* and *A[i+1][j]* are assigned to different pages. We assume here that the page is large enough to contain a row of the array.

We also experiment on the architecture with word-interleave address mapping. It means that if *A[j][i]* is in bank 0, *A[j][i+1]* will be placed in bank 1. The experiments are for the second group only, because they access array elements of different indexes during one loop iteration. Unlikely, benchmarks in the first group always access arrays with fixed indexes during one loop iteration. Since the first group won't benefit from this architecture, they are excluded here.

B. *Environment setup*

Table 3 lists the resource constraints used in our experiments. We use SDRAM, with timing rules following Intel's spec [3]. We make the options for CL=3, Trp=2, and Trcd=2, with burst length set to 1.

TABLE 3 RESOURCE CONSTRAINTS

Function Unit	ALU	Multiplier	Divider	SDRAM	SDRAM
Supported Op	+,-,>,S	*	/	BA,BP	R,W
Clocks	1	2	4	2	3
Quantity	1	1	1	2	2

TABLE 4
AVERAGE SCHEDULE LENGTH OF DIFFERENT CONFIGURATIONS

Configuration	1 Chip /2 Banks	1 Chip /4 Banks	2 Chips /1 Bank	4 Chips /1 Bank
W/O PageMode	70.20%	62.28%	64.93%	54.51%
W/ PageMode	53.38%	43.36%	52.52%	42.02%

C. *Results*

Results of the first group are shown in Fig.8, and the second group in Fig.9. In the second group, benchmarks with "2" in their name shows the results targeted toward the word-interleave architecture. We compare our results with Panda's work in Fig.10. Average schedule length of the first and second group are shown in Table 4.

Scheduling results that treat each memory access as a multi-cycle operation are shown in "Coarse". Reading data are assumed to be available after the precharge command is completed. Writing data should be ready before the operation is scheduled. Scheduling in other experiments treats each memory access as 3 operations: Decode, Read/Write, and Precharge. The data are available after the completion of read command, and should be ready before write command.

Memory configurations are varied from 1 bank, 2 banks, to 4 banks, with one memory module used. We also try the condition "2 chips" and "4 chips", which means 2 or 4 independent DRAM modules with only one bank are used. These two configurations are proposed here to show the effects of relaxing command bus contention. For these two configurations, our algorithm assigns arrays to different chips instead of different banks. The resource constraints and SDRAM timing are the same as referred in Table 3. The cycle counts are normalized with coarse scheduling result equals to 100%.

Experiments labeled with "+P" utilize page mode accesses whenever possible. A benchmark could utilize the page mode if there exists at least one array whose allocated bank does not contain any other array during the same loop iteration. For example, if there are 3 arrays: a, b, and c, and there are 2 banks. If we allocate array a and c to bank 0, array b to bank 1, then array b enjoys page mode access in the loop body. Because the row addresses sent to bank 1 are always belonging to array b and therefore we have to send it just once.

Notice that there are no page mode results for the "word interleaving" architecture. Because the accessing page changes every loop iteration, it's impossible to utilize page mode access. It can still benefit from multiple banks, however.

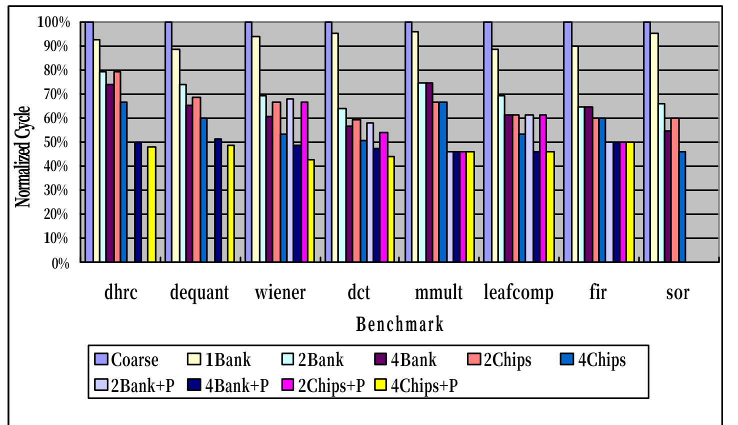

Fig.8. Results of the first group benchmarks

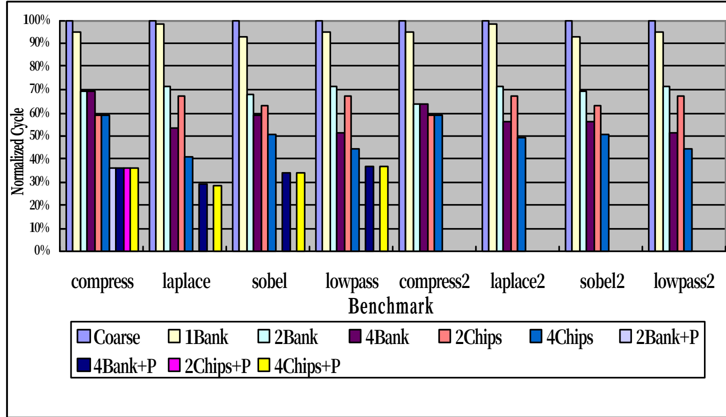

Fig.9. Results of the second group benchmarks

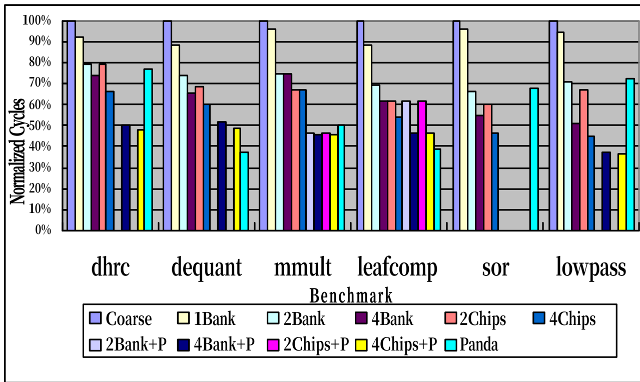

Fig. 10. Results compare to Panda's work

D. Discussions

From the results in Fig.8 and Fig.9, we can see that with our bank allocation algorithm, scheduling taking SDRAM's characteristics into account do improve the overall performance. For 2-bank cases, we get the schedule length reduced to 70% in average. For 4-bank cases, the average schedule length is reduced to 62%. If page mode is utilizable, the schedule length can be further reduced to 53% and 43%, respectively.

The use of multiple DRAM modules provides the ability of concurrent access to memory. From our results, using multiple modules with one bank helps reduce approximately 8% schedule length without the use of page mode, compares to only one module with 2 or 4 banks. If page mode is utilized, the improvements are only about 1%. This is due to the reduced number of command send to SDRAM. Without page mode, it needs 3 commands to complete each read/write operation, thus increase the probability of command bus contention.

From Fig.10, we can see that our work performs much better than Panda's work in "dhrc," "sor," and "lowpass". The results of "mmult" are nearly the same. There are still two cases that Panda's work performs better. We have to state that the comparisons are made just to show the advantages of using multi-bank SDRAM. Our work and Panda's are targeted to different architecture. In Panda's work, several behavioral transformation techniques were used. Our work does not make any transformation to the program. Our work shows the effect of bank assignment under common paging policies, and it can be integrated with other optimization techniques.

VIII. Conclusions & Future Work

In this paper, we have presented a bank allocation algorithm and a scheduler that takes SDRAM's characteristic into account. The scheduling meets timing constraints of Intel's PC SDRAM's spec and our experiments are based on common paging policies. Experimental results show a significant improvement by utilizing SDRAM's multi-bank characteristic. We propose our work to show the importance of utilizing memory's special characteristics. Other scheduling optimization techniques can be applied together to get better results.

We also experimented on several different memory configurations. System designer can decide on an appropriate configuration for his or her application by our simulation flow. After the configuration is made, results of our bank allocation algorithm can be integrated into compilers to allocate arrays to suitable addresses.

Our future work includes grouping and mapping arrays to incorporate burst transfer, extending to Rambus DRAM, and the integration of our algorithm with other scheduling techniques.

IX. Bibliography

[1] P. R. Panda, N. D. Dutt, and A. Nicolau, "Incorporating DRAM access modes into high-level synthesis", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, No.2, Feb. 1998, P.96-109.

[2] Intel 440BX AGPset: 82443BX host bridge/controller datasheet, from Intel's web site: http://www.intel.com/.

[3] PC SDRAM specification, version 1.51, from Intel corporation, Nov 1997.

[4] P. R. Panda and N. D. Dutt, "1995 high level synthesis design repository", in Proceedings of International Symposium on System Synthesis, 1995, P.170-174.

[5] P. R. Panda, N. D. Dutt, and A. Nicolau, "Local memory exploration and optimization in embedded systems", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No.1, Jan. 1999, P.3-13.

[6] J. Skeppstedt and M. Dubois, "Hybrid compiler/hardware prefetching for multiproceesors using low-overhead cache miss traps", in Proceedings of the International Conference on Parallel Processing, 1997, P.298-305.

[7] H. Schmit and D. E. Thomas, "Array mapping in behavioral synthesis", in Proceedings of the International Symposium on System Synthesis, 1995, P.90-95.

[8] P. R. Panda, N. D. Dutt, and A. Nicolau, "Memory issues in embedded systems-on-chip optimizations and exploration", Kluwer Academic Publishers, 1999.

[9] P. M. Embree and B. Kimble, "C language algorithms for digital signal processing", Prentice Hall, 1991.

[10] FPM dram datasheet., http://www.etron.com/614081.html

[11] I. Pitas, "Digital image processing algorithms", Prentice-Hall, 1993.

[12] EDORAM datasheet, http://www.etron.com/615162.html

[13] Rambus datasheet, http://www.rambus.com/developer/quickfind_documents.html