Memory Access Optimization Through Combined Code Scheduling, Memory Allocation, and Array Binding in Embedded System Design

Jungeun Kim
System R&D Laboratories,
Samsung Electronics Co., Ltd.
Seoul, Korea
jein.kim@samsung.com

Taewhan Kim
School of Electrical Engineering,
Seoul National University,
Seoul, Korea
tkim@ssl.snu.ac.kr

ABSTRACT

In many of embedded systems, particularly for those with high data computations, the delay of memory access is one of the major bottlenecks in the system's performance. It has been known that there are high variations in memory access delays depending on the ways of designing memory configurations and assigning arrays to memories. Furthermore, embedded DRAM technology that provides efficient access modes is actively developed, possibly becoming a mainstream in future embedded system design. In that context, in this paper we propose an effective solution to the problem of (embedded DRAM) memory allocation and mapping in memory access code generation with the objective of minimizing the total memory access time. Specifically, the proposed approach, called MACCESS-opt, solves the three problems simultaneously: (i) determination of memories, (ii) mapping of arrays to memories, and (iii) scheduling of memory access operations, so that the use of DRAM access modes is maximized while satisfying the storage size constraint of embedded system. Experimental data on a set of benchmark designs are provided to show the effectiveness of the proposed integrated approach. In short, MACCESS-opt reduces the total memory access latency by over 18%, from which we found that our memory mapping and scheduling techniques in MACCESS-opt contribute about 12% and 6% reductions of total memory access latency, respectively.

Categories and Subject Descriptors:
C.3 [Special-purpose and application-based systems]: Real-time and embedded systems

General Terms: Algorithms, Design

Keywords: memory access, scheduling, binding

1. INTRODUCTION

An effective utilization of chip area is an important issue in the design of embedded systems. In particular, with the increasing design complexity, embedded memory has become a critical component whose integration needs to be addressed during the process of system design. To improve the access bandwidth, modern memories provide efficient access modes such as page mode and read-modify-write [1, 2]. Since the new generation of memories (e.g., extended data out DRAM's, synchronous DRAM's, etc.) incorporates some of those efficient access modes, it is very necessary to exploit the memory access modes in the memory allocation problem in embedded system design. The major tasks which affect the memory accesses are scheduling of memory accesses in code, allocation of memory modules, and binding variables to memories for storage. Since the results of those tasks affect each other in significant ways, to fully exploit the access modes of memories to reduce the total memory access latency, it is required to have a global view on the interactions among the tasks.

In most memory access intensive embedded applications, array accesses are dominant in the total memory accesses. On the other hand, it is known that page mode access is one of the most efficiently used DRAM accesses. Consequently, the ways of determining the number and size of memories (i.e., allocation), the ways of assigning arrays to memories (i.e., binding), and the ways of determining the memory access time (i.e., scheduling) would significantly effect the amount of the use of page mode accesses.

Several memory-related hardware and software optimization issues, such as memory configuration to minimize the area, memory selection, and variable binding, have been addressed in the literature. Schmit and Thomas [3] addressed the problem of allocating memories to minmize the cost of index calculation cost delay in behavioral synthesis. However, they did not consider the exploitation of DRAM's efficient access modes. Panda [4] proposed a memory bank exploration algorithm which makes use of a sequence of memory accesses in behavioral HDL code to minimize the number of page misses. Panda, Dutt, and Nicolau [5] modeled a number of realistic page access modes in DRAMs and proposed an algorithm for arranging scalar variables to memory and organizing array variables by applying loop transformation techniques in behavioral synthesis with the objective of maximizing the number of page mode accesses. Shiue and Chakrabarti [6] proposed an optimal ILP model and a heuristic based algorithm for solving the problem of determining memory configuration with minimum area satisfying power constraint, or with minimum power consumption satisfying area constraint. Balasa, *et al.* [7] used a dataflow analysis technique to configure a memory architecture consisting of one or more memories satisfying a given timing constraint. Note that the forementioned approaches did not take into account the utilization of scheduling effect, which is a critical factor that can influence the quality of memory configuration. In [8, 9], the problem of memory allocation is addressed. Given a library of memory modules, the authors tried to find

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

a memory organization that results in a maximum performance or a minimum energy consumption under performance constraint. (A full survey on memory and array optimization in high-level synthesis can be found in [10].) Kadayif *et al.* [11] proposed a locality-conscious memory access scheduling strategy, where they evaluate the potential data reuse between processes, and determine the memory access schedule based on the evalutaion. However, they focused on the useability of cache, and did not consider various types of memory modules or access modes. Choi and Kim [12] showed that the problem of (non-array) variable assignment to memory to maximize the page mode accesses is NP-complete, and proposed a greedy heuristic to solve the problem.

In this paper, we propose an effective method of maximizing page mode accesses of code, that is, minimizing DRAM access latency of code, by solving the two important tasks in embedded system design: (1) *memory access scheduling* in code, which is a software optimization task and (2) *memory allocation and mapping*[1] for arrays in code under memory size constraint, which is a hardware-software optimization task.

2. PRELIMINARIES AND MOTIVATING EXAMPLE

Memory module	size ($bits * words$)	area (mm^2)
M1	16 x 1024	5.4154
M2	32 x 1024	10.8309
M3	16 x 2048	7.6586
M4	32 x 2048	15.3171
M5	16 x 4096	10.8308
M6	32 x 4096	21.6617
M7	16 x 8192	15.3171

Table 1: A memory module library

Before we illustrate how the memory allocation for arrays and access scheduling affect the execution of page mode memory accesses, we explain briefly what the normal and page modes are in DRAMs [12]. The execution of *normal mode* access starts from a row decoding stage where the entire row that contains a set of m words[2] is copied into a row buffer. In the following column decoding stage, the element (i.e., word) in the buffer corresponding to the column address is selected, and is read or written according to the status of r/w signal. Finally, a precharging stage is performed to prepare the execution of row decoding for the next memory access operation. On the other hand, in *page mode* an initial access which starts from row decoding followed by column decoding is first performed. Then, if the word to be accessed in the next memory operation has already been in the same page that was retrieved just before, the execution of row decoding is not needed. That is, only an additional column decoding using the corresponding column address is required to access the word from the buffer. Such a subsequent access is called *page access*, and the initial access followed by page accesses is called *page mode* access(s). In terms of access delay, the page access delays are usually much shorter than the initial access delay. Consequently, applying as many page accesses as possible to a sequence of memory accesses is a key to reduce the overall memory access latency.

Fig. 1(a) shows a segment of high-level source code in DFG (dataflow graph) form, in which there are 9 instructions manipulating four arrays A, B, C and D with size of $16bits * 1024words$ each. Fig. 1(b) then shows a possible execution schedule of the instructions. Note that a schedule of instructions determines the order of memory access operations in the instructions. For example, in the schedule in Fig. 1(b), *read_A[i]* and *read_B[i]* in $op1$ should be executed before the execution of *read_B[i]* and *read_C[i]* in $op2$ because $op2$ is scheduled to be executed after the execution of $op1$. However, the schedule alone cannot tell exactly which memory access operations are page accesses and which operations are non-page accesses until the outcomes of memory allocation and binding are known. Fig. 1(c) shows a set of possible memory allocations and bindings for arrays A, B, C and D when we use the DRAM module in the library of Table 1. We can easily see that the total memory cost will be the largest when each array is bound to a distinct memory module, but the page accesses will be at a maximum. Conversely, the total memory cost becomes the smallest when all arrays are bound to a single memory module, but the page accesses will be at a minimum.

Example 1. (*memory access sequence with no allocation/binding and schedule optimization*) Fig. 1(d) shows memory access sequences resulting from the execution schedule of instructions in Fig. 1(b) and a random allocation/binding indicated at the top of Fig. 1(d) (i.e., two memories, one containing A and B, the other containing C and D). As a result, the total memory cost and memory access latency corresponding to the schedule in Fig. 1(b) and the allocation/binding in Fig. 1(d) are 21.66 units of area and 59 (= 4·5 + 4·8 + 2·2 + 1·3) clock cycles, which is the sum of the access delays of 4 NRs (normal reads), 4 NWs (normal writes), 2 PRs (page reads) and 1 PW (page write), respectively, and an NR, an NW, a PR, and a PW take 5, 8, 2, and 3 clock cycles, respectively.

Example 2. (*memory access sequence with allocation/binding optimization only*) On the other hand, the top of Fig. 1(e) shows another memory access sequences generated when we use the allocation/binding at the top of Fig. 1(e) and the schedule in (b), which results in total 18.48 units of memory cost and 41 cycles of access latency, which is about 31% reduction over that in (d).

Example 3. (*memory access sequence with both allocation/binding and schedule optimization*) Finally, Fig. 1(f) shows another memory access sequences when the memory allocation/binding and schedule are changed according to the top of Fig. 1(f). Consequently, the corresponding total latency is reduced to 38 cycles, which is about 36% and 7% reductions over those in Fig. 1(d) and Fig. 1(e), respectively. This example strongly implies that both of the tasks of memory allocation/binding and instruction scheduling can affect the amount of the use of page accesses significantly, thus, the total memory access latency, and the two tasks should be taken into account in an integrated fashion to fully exploit the use of page mode accesses.

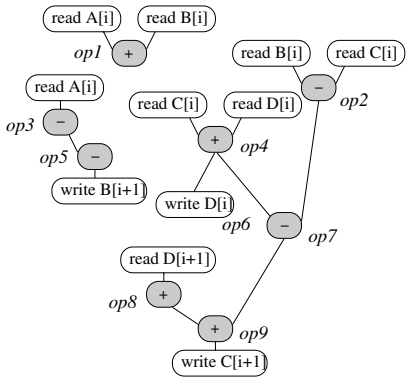
3. THE PROPOSED ALGORITHM

The problem we want to solve is: *Given a DFG of high-level source code with array accesses and DRAM module library and memory cost constraint, allocate memories, bind arrays to memories, and schedule instructions in DFG, so that the total memory access latency is mimimized while satisfying the memory cost constraint.*

For an initial schedule of instructions in DFG, our proposed approach, called MACCESS-opt (memory access code optimization), solves the problem by performing the two steps iteratively: (Step 1) *memory reallocation/rebinding* and (Step 2) *rescheduling memory accesses*. In Step 1, we reconsider the task of memory allocation and array binding to reflect the changes of instruction schedule in the previous iteration. This step tries to find a memory configuration that is well suited to the new schedule so that the latency of memory accesses is further reduced. In Step 2, we attempt to reschedule the instructions (incrementally) in a way that the changed schedule com-

[1] In some literature, *memory allocation and mapping* are collectively referred to as memory configuration.

[2] The value of m is memory-dependent. Such a row of m words in memory is collectively called a *page* of size m.



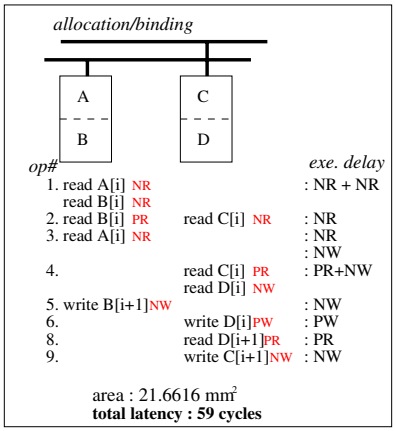
(a) DFG representation of a segment of high–level source code

op1:	e = A[i] + B[i]	; read A[i], read B[i]
op2:	f = B[i] − C[i]	; read B[i], read C[i]
op3:	g = A[i] − 3	; read A[i]
op4:	h = C[i] + D[i]	; read C[i], read D[i]
op5:	B[i+1] = e − g	; write B[i+1]
op6:	D[i] = h	; write D[i]
op7:	i = f − h	
op8:	t = D[i+1] + 4	; read D[i+1]
op9:	C[i+1] = i + t	; write C[i+1]

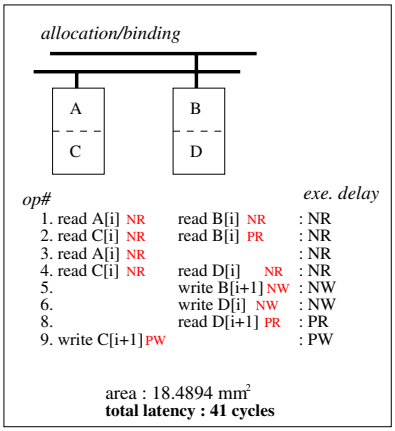
(b) An execution sequence (i.e. schedule) of the DFG in (a)

Memory Configuration (array group : memory)	Area (mm^2)
A: M1, B: M3, C: M1, D:M3	26.1480
AC: M3, B: M3, D: M3	22.9758
A: M1, BC: M5, D: M3	23.9048
A: M1, B: M3, CD: M5	23.9048
AC: M3, BD: M5	18.4894
...	
ABCD: M7	15.3171

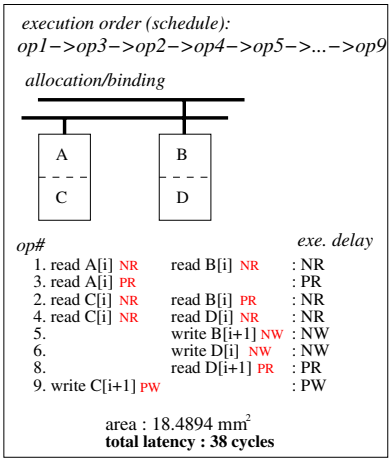
(c) A set of possible memory allocations and bindings for the arrays A, B, C, D used in (a)



(d) Memory access sequences resulting from the operation schedule in (b) and a random memory allocation and binding shown above.



(e) Memory access sequences resulting from the operation schedule in (b) and memory reallocation and rebinding from (c)



(f) Memory access sequences resulting from a careful rescheduling of operations and memory allocation/binding shown above.

Figure 1: Examples showing how two tasks, memory allocation/binding and operation scheduling, affect the results of total latency of memory accesses. We assumed that a page read (PR), a normal read (NR), a page write (PW) and a normal write (NW) take 2, 5, 3, and 8 clock cycles, respectively.

bined with the memory allocation/binding obtained in Step 1 leads to a reduced total latency of memory accesses. (The rescheduling step is further divided into two substeps called *macro-rescheduling* and *micro-rescheduling*, which will be described in detail in the next section.) We repeatedly performed the two step until there is no reduction on the total latency of memory accesses. We now provide the details of the procedure of the two steps. Note that the initial schedule of instruction in DFG is the one that is written initially in the code.

• Step 1: Reallocation/Rebinding

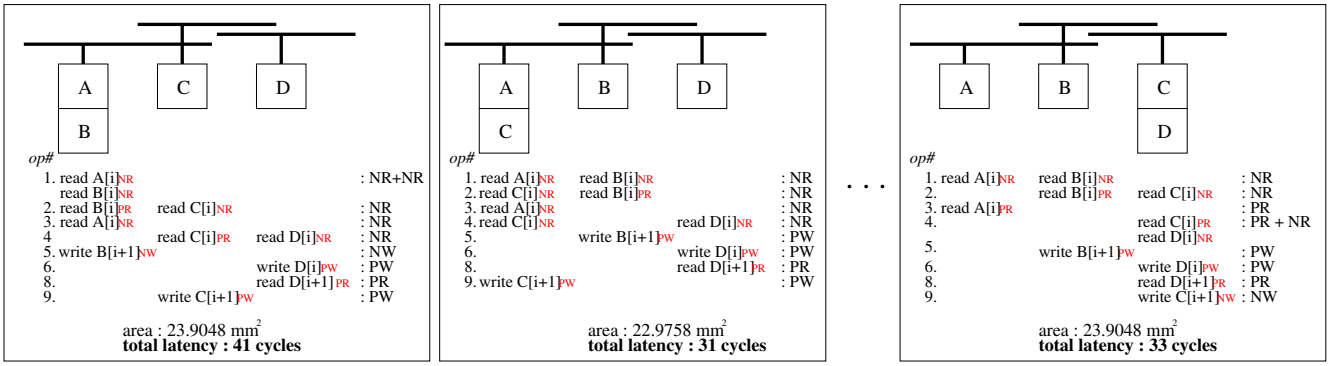
In this step, we attempt to find a solution of allocation and binding that is best suitable to the result of scheduling memory access operations obtained in step 2 of the previous iteration. The objective is to find a solution of allocation and binding that leads to a minimum latency of memory accesses under the total memory cost constraint.

We use a bottom-up approach to solve the problem. Initially, we assume that each array is bound to a distinct memory instance, thus using a total of N memory instances for N arrays. Consequently, the total memory cost of the initial allocation/binding will be the largest, but the latency is the least. Then, we apply a pair-wise merge among the memories allocated; For each pair of allocated memories m_i and m_j, we find a memory, M_{ij}, with the lowest cost from the memory library that can occupy all the arrays bound to m_i and m_j, and compute the resultant latency. We then, among the pairs of memory instances, select the pair with the largest value of the quantity

$$\Delta C_1 = \frac{M_0 - M}{L - L_0} \quad (1)$$

where L_0 and M_0 are the latency and total memory cost of the current schedule and allocation/binding, respectively and L and M are the latency and total memory cost for the merging of the two memory instances. ΔC_1 indicates a measure of the effectiveness of the memory merge on the memory cost reduction.

For example, Fig. 2(a) shows a number of results for the merging of pair of memory instances, from the leftmost to the right, merging memory instance containing A and instance containing B, merging instance with A and instance with C, and merging instance with C and instance with D. The table in Fig. 2(b) summarizes the total memory cost (i.e., area), latency and the value of ΔC_1 for each merge, from which we found that merging two memory instances having arrays A and D produces the largest value of ΔC_1. Consequently, we merge the two memory instances into one to contain the arrays in the instances. The merge process then repeats until there remains only one memory instance. Fig. 2(c) shows the merging steps from bottom to top in which the right side indicates the total memory cost and latency obtained at each iteration of the merging process.

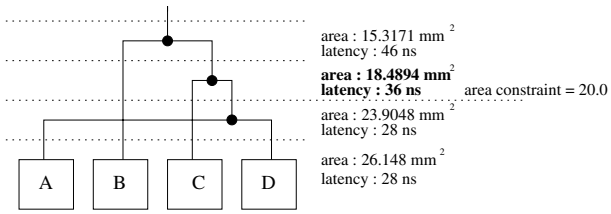


(a) Possible groupings of 4 memory instances and the corresponding memory costs (in terms of area) and latencies for the schedule in Fig.1(b)

M0 (initial memory cost) = 26.1480; L0 (initial latency) = 28

Memory grouping	M:memory cost	L:latency	Δ C1
{AB}, {C},{ D}	23.9048	41	0.172
{AC}, {B}, {D}	22.9758	31	1.05
{AD}, {B}, {C}	**23.9048**	**28**	**infinite**
{A}, {BC}, {D}	23.9048	40	0.186
{A}, {BD},{ C}	21.6617	38	0.44
{A}, {B}, {CD}	23.9048	33	0.44

(b) A summary of all the possible mergings of memory instances and their costs



(c) The results of botton−up clustering of memory instances for the schedule in Fig.1(b)

Figure 2: Example explaining the procedure of refining allocation/binding (in a bottom-up fashion) for the schedule of memory access operations obtained from Step 1 of the previous iteration.

When we assume to have a total memory cost constraint of 20 mm^2, the best allocation and binding for the schedule in Fig. 1(b) is the third layer from bottom in Fig. 2(c), where two memory instances are allocated, one bound to arrays A, C and D, and the other bound to array B.

Reallocation/rebinding($\mathcal{S}$, $\mathcal{L}$, M_limit)
/* $\mathcal{S}$: schedule, $\mathcal{L}$: memory library, */
/* M_limit: upper bound of total memory cost */
• For each array in $\mathcal{S}$, allocate and bind a distinct memory instance of the lowest-cost from $\mathcal{L}$;
• Set $\mathcal{Q}$ = {allocated and bound memory instances};
• Compute $M_0 = f(\mathcal{Q})$; /* $f(\cdot)$: total memory cost */
while ($M_0 > M\_limit$) {
  • Compute $L_0 = g(\mathcal{S}, \mathcal{Q})$; /* $g(\cdot)$: latency
  foreach (pair of $m_i, m_j \in \mathcal{Q}$) {
    • Get a memory instance, m_{ij}, from $\mathcal{L}$ of the lowest-cost, which can occupy all arrays in m_i and m_j;
    • Compute $M = f(\mathcal{Q}\text{-}\{m_i, m_j\}+\{m_{i,j}\})$;
    • Compute $L = g(\mathcal{S}, \mathcal{Q}\text{-}\{m_i, m_j\}+\{m_{i,j}\})$;
    • Compute $\Delta C_1(M, L, M_0, L_0)$; /* Eq.(1) */
  } **endfor**
  • Select $m_i, m_j \in \mathcal{Q}$ with the largest value of $\Delta C_1(\cdot)$;
  • Allocate $m_{i,j}$ and bind all arrays in m_i and m_j to $m_{i,j}$;
  • Set $\mathcal{Q} = \mathcal{Q}\text{-}\{m_i, m_j\}+\{m_{i,j}\})$;
  • Update $M_0 = f(\mathcal{Q})$;
} **endwhile**
• **return** $\mathcal{Q}$;

Figure 3: The proposed reallocation/rebinding for a schedule $\mathcal{S}$ in MACCESS-opt.

Fig. 3 summarizes the procedure of reallocation/rebinding. Notation Q is used to represent an instance of memory allocation and binding, and $f(\mathcal{Q})$ and $g(\mathcal{S}, \mathcal{Q})$ are used to represent the calculation of the total memory cost for $\mathcal{Q}$ and the calculation of total memory latency for schedule $\mathcal{S}$ and allocation/binding $\mathcal{Q}$, respectively.

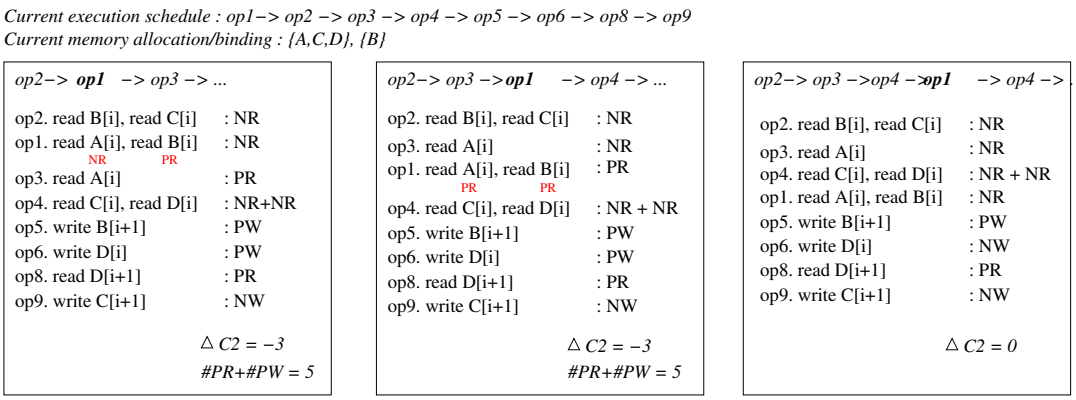
• Step 2: Rescheduling

The objective of this step is to find a best schedule of intructions and memory access operations for the result of the updated memory allocation and bining produced in Step 1. The rescheduling step consists of two parts: *macro-rescheduling*, which tries to change the schedule of instructions and *micro-rescheduling*, which tries to change the schedule of memory access operations in the execution of the instructions. The *macro-rescheduling* should preserve the data dependencies in the code while the *micro-rescheduling* should take into account the limited use of the registers in the processor. We perform the *macro-rescheduling*, followed by the *micro-rescheduling*. During the *macro-rescheduling*, it is assumed that the executions of the memory read and write accesses of the instruction occur immediately before and after the clock step of the execution of the instruction. (The *micro-rescheduling* will exploit the possiblity of pre-read and post-write under the register constraint to reduce the latency further.)

We use an iterative approach to solve the *macro-rescheduling*, and consists of two loops, one nesting the other. The *inner-loop* performs the following: It first generates all possible candidates of reschedules of instructions that can be obtained by rescheduling each instruction for execution without violating the data dependency in the code, and calculates the quantity.

$$\Delta C_2 = L - L_0 \tag{2}$$

where L and L_0 is the latencies before and after the corresponding reschedule of instruction, respectively. Then, among the instances of schedules, it chooses the one with the least value of ΔC_2, and applies the corresponding schedule. The instruction that was selected and rescheduled is then locked. The process repeats from the updated schedule and the iteration stops when all the instructions are locked or no more rescheduling can be applied without violating the instruction dependencies in the code. The best schedule of instructions is



(a) Results for all possible macro−reschedulings of op1

	reschedule	L	△C2	#PR+#PW
op1	op1−>op2−>...	36	0	
	op2−>op1−>op3−>...	33	**−3**	**5**
	op2−>op3−>op1−>op4−>...	33	**−3**	**5**
	...−>op3−>op4−>op1−>op5−>...	36	0	
op2	op2−>op3−>op1−>...	33	0	
	op3−>op2−>op1−>...	36	+3	
	op3−>op1−>op2−>op4−>...	36	+3	
	...−>op1−>op4−>op2−>op5−>...	36	+3	
op3	op2−>op3−>op1−>...	33	0	5
	op2−>op1−>op3−>op4−>...	33	0	5
	...−>op4−>op3−>op5−>...	36	+3	

	reschedule	L	△C2	#PR+#PW
op4	...−>op1−>op4−>op5−>...	33	0	5
	...−>op1−>op5−>op4−>op6−>...	33	0	5
	op2−>op3−>op4−>op1−>...	43	+10	
	op2−>op4−>op3−>...	40	+7	
	op4−>op2−>...	40	+7	
op5	...−>op4−>op5−>op6−>...	33	0	5
	...−>op5−>op4−>op6−>...	33	0	5
	...−>op4−>op6−>op5−>op8−>...	33	0	5
	...−>op6−>op8−>op5−>op9	33	0	5
	...−>op8−>op9−>op5	33	0	5
op6	...−>op5−>op6−>op8−>...	33	0	5
	...−>op5−>op8−>op6−>op9	33	0	5
	...−>op8−>op9−>op6	36	+3	

(b) The cost computations for all possible macro−reschedules

Figure 4: Example explaining the procedure of *macro-rescheduling* for the result of reallocation/rebinding obtained from Step 1.

the one with the least ΔC_2 value among the schedules obtained during the iteration process. The *outer-loop* then unlock all the instructions, and again, set the best as an initial schedule to *inner-loop*. The *outer-loop* stops when there is no more reduction in total latency.

For example, consider a reschedule of operation $op1$ in Fig. 1(b), given the memory allocation and binding (i.e., $\{A, C, D\}$, $\{B\}$) in Fig. 2(c), to a clock step after the execution of operation $op2$ as shown in the left side of Fig. 4(a). The schedule leads to a latency reduction of 3, as indicated by ΔC_2 at the bottom of Fig. 4(a). Fig. 4(a) shows all the possible reschedules of $op1$ with the corresponding ΔC_2 values. For every operation, we attempt to reschedule the operation and compute the quantity of ΔC_2 in *Eq.(2)*. For example, the table in Fig. 4(b) shows all candidates of *macro-reschedules* (i.e., instruction-level) with ΔC_2 values. Among the reschedules of all instructions, we select the reschedule of instruction with the smallest ΔC_2 value. If there are ties, we select the one with less number of page read and write accesses in the schedule. For example, in the table in Fig. 4(b), the second and third schedules are selected because their ΔC_2 is the smallest. We then check the numbers of page read and write accesses of the two reschedules (denoted by *#PR+#PW* in the table). In this example, the numbers are the same. In that case, we select one randomly from the two reschedules. The next step is to apply the *micro-rescheduling* to the final schedule obtained from the *macro-rescheduling*. We omit the details due to the space limitation.

4. EXPERIMENTAL RESULTS

We implemented our proposed technique MACCESS-opt in C, ran on a Linux PC equipped with 2.4GHz Pentium4 processor, and tested it on a set of benchmark programs in numerical recipes [15] to check how much the proposed technique is effective. The programs of numerical recipes includes many function modules for solving mathematical problems and most of them belong to the memory access intensive applications with arrays. FOURFS is the function for the interpolation and extrapolation, SPLINE is a cubic spline algorithm used to performs interpolation of the coordinates during the raw position computation. STOERM is a routine implemented according to stoermer's rule which has been used as a typical method for the system of second-order conservative equations, and PZEXTR is the polynomial extrapolation routine. Finally, RATINT is a rational interpolation and extrapolation function. We evaluate our technique in two-fold: (1) checking the effectiveness of MACCESS-opt in maximizing page mode accesses and (2) checking the effectiveness of MACCESS-opt without Rescheduling and with Rescheduling in reducing total memory access latency.

• **Checking** MACCESS-opt **in maximizing page mode accesses**: Fig. 5 shows the amount of the increases of page mode accesses by the application of MACCESS-opt to the initial codes by varying the page size to 16, 64, 128, 256, and 512 for each design. The curves in Fig. 5 clearly shows that MACCESS-opt quite performs well in maximizing the use of page mode accesses by the exploration of memory allocation, binding and scheduling. In summary, the overall increase of page mode accesses by MACCESS-opt is in the range of 9% - 65%, and as the page size increases, the number of page accesses by MACCESS-opt also tends to slightly increase.

• **Checking** MACCESS-opt **without and with** Rescheduling **in reducing total memory access latency**: Table 2 summarizes the total memory latencies used by an initial schedule with a random binding, our MACCESS-opt without Rescheduling and with MACCESS-opt. The second column indicates the number of arrays of the corresponding code to test and the constraint of total memory cost. (We used the memory library in Table 1.) The remaining columns in-

design	array/ area	Total memory access latency (page_size = 16/64/128/256)		
		No_Realloc/Rebind/Resch	MACCESS-opt w/o Rescheduling	MACCESS-opt w/ Rescheduling
FOURFS	6/21.5	171/168/168/168	160/157/156/156	150/148/146/146
SPLINE	4/17.5	85939/82871/82360/82104	69451/65713/65090/64778	69451/65713/65090/64778
	4/16.5	85939/82871/82360/82104	76540/73616/73129/72885	72134/68683/68108/67820
STOERM	4/17.5	59077/58933/58909/58897	59077/58933/58909/58897	44642/44570/44558/44552
PZEXTR	6/28.5	189886/189762/185075/184731	143172/138760/138025/137657	143172/138760/138025/137657
RATINT	4/8.25	321483/303196/300148/298624	284310/269040/266495/265223	284308/269038/266493/265221
improvment			**12.2%/12.9%/12.5%/12.7%**	**18.0%/18.9%/17.2%/18.8%**

Table 2: Total memory access latencies by initial schedules with random binding, MACCESS-opt **without and with** Rescheduling**.**

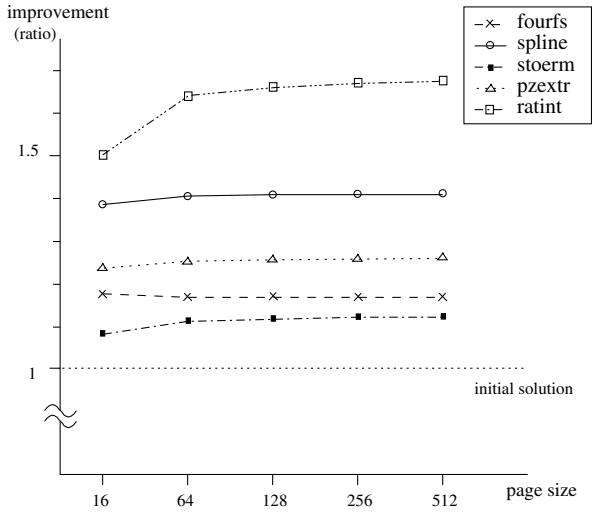


Figure 5: The increase of page mode accesses by the application of MACCESS-opt **to the initial codes, varying the page size to 16, 64, 128, 256, and 512 words.**

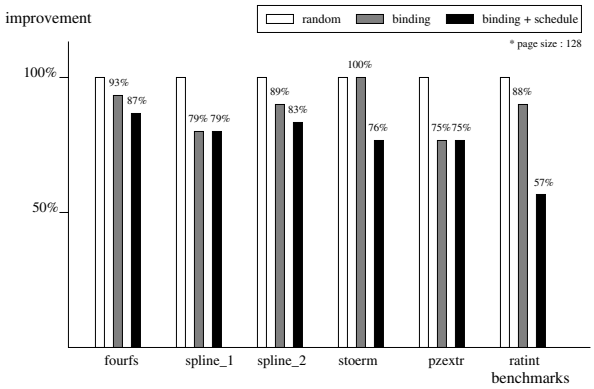


Figure 6: The reductions of total memory access latency by Reallocation/Rebinding **and** Rescheduling **for each testcase with page size = 128 words.**

dicate the total memory access latencies used by initial schedules with random memory allocation and binding by performing Reallocation/Rebinding only in MACCESS-opt and by MACCESS-opt, where we used the same delay numbers of memory accesses that are used in Fig. 1. The results show that the overall reduction of total memory access latency by our Reallocation/Rebinding is more than 12%, and about 6% additional reduction (i.e. total 18% reduction) is possible when our Rescheduling is combined with Reallocation/Rebinding. From the table, we can see that the reduction by MACCESS-opt is consistent independently of both the page size and the testcase, which is also verified by Fig. 6, in which the numbers indicate the percentage of reduction in total memory access latency used by Reallocation/Rebinding and MACCESS-opt for each testcase.

5. CONCLUSION

In this paper, we proposed an effective memory access code optimization algorithm, MACCESS-opt, which simultaneously solves the three important problems: (i) determination of memories, (ii) mapping of arrays to memories, and (iii) scheduling of memory access operations, with the objective of maximizing the use of DRAM access modes to reduce the total memory access latency, while satisfying the memory size constraint in embedded system design. The proposed algorithm can be usefully applied to the memory-resource constrained, but memory access intensive embedded system applications to improve the memory access latency by utilizing the efficient page mode accesses in (embedded) DRAMs. From our experimental data, we found that the three problems were almost equally effective in reducting the total memory access latency, contradicting each other in significant ways. In summary, the overall reduction of memory access latency by our MACCESS-opt was 18% on the average.

ACKNOWLEDGEMENT: This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

6. REFERENCES

[1] B. Prince, *High Performance Memories, New Architecture DRAMs and SRAMs Evolution and Function,* Wiley, West Sussex, 1996.

[2] S. Przybylski, "Sorting out the new DRAMs", *In Hot Chips Tutorial*, Stanford, CA, 1997.

[3] H. Schmit and D. E. Thomas, "Array Mapping Behavioral Synthesis", *ISSS*, 1995.

[4] P. R. Panda, "Memory Bank Customization and Assignment in Behavioral Synthesis", *ICCAD*, 1999.

[5] P. R. Panda *et al.*, "Incorporating DRAM Access Modes into High-level Synthesis", *IEEE TCAD*, Vol. 17, 1998.

[6] W. T. Shiue and C. Chakrabarti, "Memory Exploration for Low Power Embedded Systems", *DAC*, 2001.

[7] F. Balasa, *et al.*, "Dataflow-driven Memory Allocation for Multi-dimensional Signal Processing Systems", *ICCAD*, 1994.

[8] J. Seo, *et al.*, "An Integrated Algorithm for Memory Allocation and Assignment in High-level Synthesis", *DAC*, 2002.

[9] S. Bakshi, and D. Gajski, "A Memory Selection Algorithm for High-Performance Pipelines" *EDAC*, 1995.

[10] P. R. Panda, *et al.*, "Data and Memory Optimization Techniques for Embedded Systems", *ACM TODAES*, Vol. 6, 2002.

[11] I. Kadayif, *et al.*, "Locality-Conscious Process Scheduling in Embedded Systems", *Proc. Symposium on HW/SW Codesign*, 2002.

[12] Y. Choi, T. Kim, "Memory Layout Techniques for Variables Utilizing Efficient DRAM Access Modes", *DAC*, 2003.

[13] W-T. Shiue, *et al.*, "Low Power Multi-Module, Multi-Port Memory Design for Embedded Systems", *Workshop on Signal Processing*, 2000.

[14] P. Grun, *et al.*, "Memory Aware Compilation Through Accurate Timing Extraction", *DAC*, 2000.

[15] W. H. Press, *et al.*, "Numerical Recipes in C", *Cambridge university press*, 1992.