Memory Layout Techniques for Variables Utilizing Efficient DRAM Access Modes in Embedded System Design

Yoonseo Choi and Taewhan Kim
Department of Electrical Engineering & Computer Science
and Advanced Information Technology Research Center(AITrc)
Korea Advanced Institute of Science and Technology, KOREA
yschoi@jupiter.kaist.ac.kr tkim@cs.kaist.ac.kr

Abstract – The delay of memory access is one of the major bottlenecks in embedded systems' performance. In software compilation, it is known that there is high variations in memory access delay depending on the ways of storing/retrieving the variables in code to/from the memories. In this paper, we propose an effective storage assignment technique for variables to maximize the use of memory bandwidth. Specifically, we study the problem of DRAM memory layout for storing the non-array variables in code to achieve a maximum utilization of page and/or burst modes for the memory accesses. The contributions of our work are, for each of page and burst modes: (1) We prove that *the problem is NP-hard*; (2) We propose an exact formulation of the problem and *efficient memory layout algorithms*, called Solve-MLP for the page mode and Solve-MLB for the burst mode; From experiments with a set of benchmark programs, we confirm that our proposed techniques use on average 20.0% and 9.9% more page accesses and 54.0% and 86.6% more burst accesses than those by OFU (the order of first use) and the technique in [1, 2], respectively.

Categories and Subject Descriptors

B.3 [**Memory structures**]: [Design styles]

General Terms

Algorithms, Performance, Embedded System

Keywords

Memory Layout, Page/Burst Modes, Storage Assignment

1. INTRODUCTION

In embedded systems, memory is one of the major sources of performance bottleneck and power consumption [3]. A large number of techniques to reduce memory access latency and increase memory bandwidth in the execution of code have been proposed. In compiler and computer architecture literature, for example, cache organization, cache block alignment, prefetching of predicted data from memory and software pipelining are considered [4, 5, 6]. In high-level/system synthesis community, alleviating performance bottleneck due to memory access is one of the major research topics; Wuytack *et al.* [7] addressed the problem of flow graph balancing to minimize the required memory bandwidth. They added ordering constraints to the flow graph to minimize the number of memory ports in system-level design. Catthoor *et al.* [8] addressed the customization of memory architecture, including memory allocation, data packing into memories and memory port sharing in embedded multimedia system design.

It should be noted that most of modern DRAMs used in embedded system design support efficient memory access modes. In particular, the page and burst access modes are extensively supported in DRAMs [9, 10]. In general, the latency of random access is much higher than that of page/burst access. Further, memory operates in lower power per bit in page/burst mode than in random access mode. For example, *IBM's Cu-11 Embedded DRAM* [9] macro supports random access mode of 10ns and page mode access of 5ns at worst. It has active current of 60*mA*/Mb roughly in random cycle and 13*mA*/Mb in page cycle. Consequently, a full exploitation of these memory access modes is very necessary to alleviate the performance bottleneck caused by the delay of memory accesses. Panda *et al.* [1] modeled a number of realistic page access modes in DRAMs and proposed an algorithm for arranging non-array variables to memory and organizing array variables via loop transformation techniques in high level synthesis to utilize the access modes. The formulation of the problem of arranging non-array variables in a memory is analogous to that in [2], which was originally used to reduce the number of cache misses. Though the formulation looks reasonable, it is, in a strict sense, not an exact formulation. Khare *et al.* [11] extended the work in [1] to support the burst mode in a dual-memory architecture, in which they focused mainly on an efficient interleaving of memory accesses. Grun, Dutt and Nicolau [12] proposed an approach that allows compiler to exploit detailed timing information of the memories. They showed a further optimization (using page and burst modes) is possible when an accurate timing of memory accesses to multiple memories is extracted. They also presented an approach [13] of extracting, analyzing and clustering the most active memory access patterns in an application and customizing memory architecture. Ayukawa *et al.* [14] proposed an access sequence control scheme for relieving the page-miss penalty in random access mode. They introduced an embedded DRAM macro attached with a special hardware logic for access sequence control. In this paper, to complement the prior work [1, 2, 11, 12, 13, 14], we propose a new approach to the problem of arranging non-array variables in code to DRAM so that the efficient page and/or burst modes are maximally utilized.

DAC 2003, June 2–6, Anaheim, California, USA.

2. PRELIMINARIES AND MOTIVATING EXAMPLE

DEFINITION 2.1. *For a sequence of variable references $a_1, a_2, \cdots, a_{i-1}, a_i \cdots$, and an assignment of the variables to the memory, let us denote $v(a_i)$ the variable referenced by a_i. Further, let $f_{page}(v(a_i))$ and $f_{addr}(v(a_i))$ be the page and relative location of the memory at which variable $v(a_i)$ is located, respectively.*
• *When normal mode and page mode are available, the access of a_i is called* **a page access** *if and only if*
$$f_{page}(v(a_i)) = f_{page}(v(a_{i-1})).$$
• *When normal mode and burst mode are available, the access of a_i is called* **a burst access**[1] *if and only if*
$$f_{page}(v(a_i)) = f_{page}(v(a_{i-1})) \text{ and}$$
$$f_{addr}(v(a_i)) = f_{addr}(v(a_{i-1})) + 1.$$

For a sequence of k memory references, if it is implemented with normal mode only, k row decoding, k column decoding and k precharge stages are needed.[2] However, if it is implemented with both normal and page modes, the best solution is to use one row decoding, k column decoding (one for initial access, $k-1$ for page access), and one precharge stages. In terms of access delay, the page and burst 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. Further, it is also true that an aggressive use of burst accesses rather than normal mode accesses definitely reduces the memory access delay.

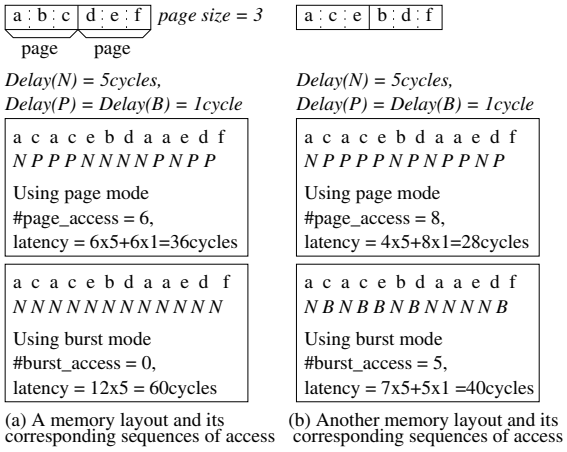

Figure 1: An example illustrating the effects of memory layout on the number of page/burst accesses.

One of the most effective ways to reduce the memory access latency using page or burst mode is to find the (relative) placement of variables in memory (i.e., memory layout for variables) which offers the maximum number of page/burst accesses. Figure 1 illustrates how different memory layouts affect the memory access latency. The top of Figure 1(a) shows a memory layout for a sequence of variable accesses in a program segment where variables a, b and c are in the first page, and d, e and f are in the second page. The sequence of normal and page accesses corresponding to the layout is shown in the upper box of Figure 1(a). It consists of 6 page accesses (i.e., 6 column-decodes) and 6 normal mode accesses (i.e., 6 row-decodes+column-decodes+precharge). On the other hand, the upper box of Figure 1(b) shows the sequence of normal and page accesses corresponding to another layout shown in the top of Figure 1(b). Note that the number of page accesses used is two more than the case of Figure 1(a), resulting in 22% reduction (i.e., from 36 cycles to 28 cycles) in memory access latency.

The lower boxes of Figures 1(a) and (b) show the sequences of memory access modes used for the corresponding memory layouts when the burst and normal modes are available, respectively. This also clearly reveals that the arrangements of variables in memory drastically affect the length of memory access latency. Thus, the optimization problem we want to solve is to find an efficient arrangement of variables to memory for a sequence of variable accesses so that the page/burst accesses are extensively utilized.

[1]There are a number of techniques for implementing burst modes. Our definition is an abstraction of them.

[2]The buffer used in row decoding stage will contain a set of m words where the value of m is memory-dependent. The m words in each row of memory are collectively called a *page* of size m.

3. MEMORY LAYOUT UTILIZING PAGE MODE

3.1 The Problem Formulation

The optimization problem is, given a sequence of variable accesses, to find a memory layout with maximum number of page accesses (e.g., *ibm Cu-11 Embedded DRAM* [9]). We call the problem *MLP* (*Memory Layout with Page mode*). An instance of *MLP* is characterized by (S, V, m) where S is a variable sequence to be accessed, V is the set of variables in S, and m is the page size, partitioning the variables in V into disjoint groups, each of which has m or less than m variables.

PROBLEM 3.1. *decision-MLP: For an MLP instance with (S,V,m) and k, where $m \geq 3$, is there a memory layout, L, in which the number of page accesses for S is greater than or equal to k? (k is a natural number, indicating the number of page accesses in this case.)*

DEFINITION 3.1. *The* access graph *of S is a multigraph $G(V,E)$ where node set V is the set of variables in S and there are n edges between two nodes v_i and v_j if and only if v_i and v_j are adjacent to each other in S exactly n times.*

Figure 2(a) shows the access graph as in Definition 3.1 for a sequence of variable accesses. For example, the access graph has three edges between nodes a and c since variables a and c are adjacent to each other 3 times in S. Figure 2(c) shows an alternative representation of the access graph in a simple graph form with edge weights.

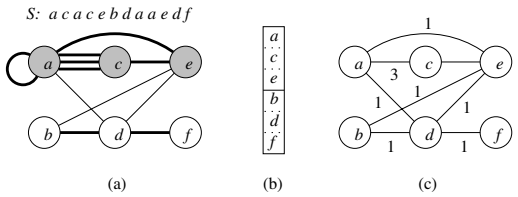

Figure 2: A sequence of variable accesses and (a),(c) its access graph representations; (b) (a)'s implied memory layout.

A graph partitioning, $\Pi^m_{G(V,E)}$, on multigraph $G(V,E)$, partitions V into t disjoint subsets $V_1, V_2, \cdots V_t$, each containing at most m vertices. The gain of $\Pi^m_{G(V,E)}$ is defined:

$$g(\Pi^m_{G(V,E)}) = \sum_{(v_i,v_j)\in E,\ v_i,v_j\in V_l, l=1,\cdots,t} w(v_i,v_j) \quad (1)$$

where $w(v_i, v_j)$ represents the number of edges between v_i and v_j in G.

An optimal graph partitioning of $G(V,E)$ is defined to find $\Pi^m_{G(V,E)}$ that maximizes the quantity of $g(\cdot)$. We claim that the partitioning problem of an access graph is NP-complete.

PROBLEM 3.2. *decision-GP: For a multigraph $G(V,E)$ and k, is there a $\Pi^m_{G(V,E)}$ $(m \geq 3)$ such that $g(\Pi^m_{G(V,E)}) \geq k$?*

PROBLEM 3.3. *decision-AccGP: For an access graph $G(V,E)$ and k , is there a $\Pi^m_{G(V,E)}$ $(m \geq 3)$ such that $g(\Pi^m_{G(V,E)}) \geq k$?*

Note that *decision-GP* is NP-complete since the partitioning problem for graphs without multiple edges is known to be NP-complete [15].

THEOREM 3.1. *decision-AccGP is NP-complete. (We showed that decision-GP is reducible to decision-AccGP in polynomial time.)*

THEOREM 3.2. *decision-MLP is NP-complete. (We showed that decision-AccGP is polynomial time reducible to decision-MLP.)*

THEOREM 3.3. *For an instance (S,V,m) of MLP problem, every memory layout L implied by an optimal Π^m_G of the corresponding instance $G(V,E)$ of problem AccGP is optimal. (Note that the construction procedure of the edges in E is the one we used in proof of the previous theorem. AccGP is the optimization version.)*

For example, Figure 2(a) shows an optimal graph partitioning and Figure 2 (b) shows its implied memory layout, leading to 8 page accesses that is the same as the value of the gain of the partitioning. The ultimate goal we want to minimize is the access latency[3] of S in (*S*,*V*,*m*) of *MLP*. The following theorem indicates that finding a maximum gain in *AccGP*, which is equivalent to finding a maximum number of pages accesses in *MLP*, also leads to finding a minimum value of access latency in *MLP*.

THEOREM 3.4. *The access latency in (S,V,m) of MLP is minimized by the memory layout implied by the optimal graph partitioning of the corresponding instance of AccGP.*

Note that there is a work [1, 2] which addressed the problem of graph partitioning to find a minimum access latency. However, its graph derivation is different from ours in that the weight assigned to edge (u,v) by [1, 2] represents, rather than the number of consecutive accesses of (u,v) or (v,u) in S, the total number of pairs of accesses of u and v in S in which the distance between them is within page size m. In their graph, if the access interval of two variables is close enough to be within the page size, those accesses are considered as having a high possibility of execution in page mode. However, this is not alway true because their representation has a tendency of overestimating the number of page accesses. (See Figure 3)

3.2 The Proposed Algorithm

Due to the NP-completeness of the memory layout problem, we develop an efficient (greedy) heuristic, called Solve-MLP, based on a node clustering procedure, which is performed on the edge weighted (simple) access graph for S (e.g., Figure 2(c)).[4]

[3]The access latency of S is the sum of latencies of all access stages (i.e., row-decode, column-decode, precharge) used in memory access for variables in S.

[4]Note that when the access graph in a multigraph form is transformed into a simple access graph, the self-loops are simply deleted since the weight of self loops is always included in the gain regardless of the resulting partition.

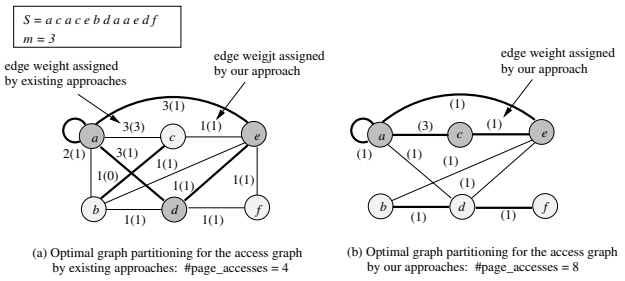

Figure 3: Examples of graph formulations and solutions according to the existing approaches [1,2] and our proposed approach.

The inputs to the proposed algorithm are the edge weighted access graph $G(V,E,W)$ and page size m. The algorithm is an iterative process. At each iteration, a group of nodes whose size is m is extracted from the access graph. This process iterates for the remaining nodes until all the nodes are extracted. Each group extracted represents a distinct page, and the variables corresponding to the nodes in the same group indicate the variables in the same page. Each iteration of the algorithm starts from a seed which is the edge with the largest edge weight. Let (u,v) be the edge selected, and set $P = \{u,v\}$. We iteratively expand set P by adding nodes, one at a time until $|P| = m$ or no nodes are left. The selection of node to be included at each time is determined based on the following two measurements: For each node x not in P,

$$attract\_in(P,x) = \sum_{e=(x,y)\in E \text{ and } y\in P} w(e) \quad (2)$$

$$attract\_out(P,x) = \sum_{e=(x,y)\in E \text{ and } y\notin P} w(e) \quad (3)$$

We select the node with the largest value of $attract\_in(P,\cdot)$. If there are ties and $|P| = m-1$, we choose the one with the smallest value of $attract\_out(P,\cdot)$ because the value of $attract\_out(P,\cdot)$ directly contributes to the number of normal accesses which require a relatively long access delay. However, when $|P| < m-1$ we choose a node among the ties in a random manner because there is no clear clue at this point. Once a node is selected, it is added to P and the node is deleted from the access graph G. The grouping process then repeats for the nodes in G. Note that Solve-MLP does not result in increased page requirement since by Solve-MLP at most one page has less thatn m variables in it, and all other pages will have exactly m variables, where m is determined by the given memory architecture.

Figure 4 shows the steps of clustering nodes by Solve-MLP. First, we select the edge (a,c) because it has the largest edge weight and set $P = \{a,c\}$ as shown in Figure 4(a). Then, we expand P by adding node e because its $attract\_in(P,\cdot)$ is one of the largest, as indicated in Figure 4(b). Now, we complete the grouping of size 4 by including one more node. Since $|P| = 3$ (m = 3) and there are two nodes d and g whose $attract\_in(P,\cdot)$ values are the largest. we choose g because its $attract\_out(P,\cdot)$ value is smaller than that of d, as shown in Figure 4(c). Consequently, the variables to be in the same page are a,c,e and g. We repeat the grouping process for the updated access graph in Figure 4(d).

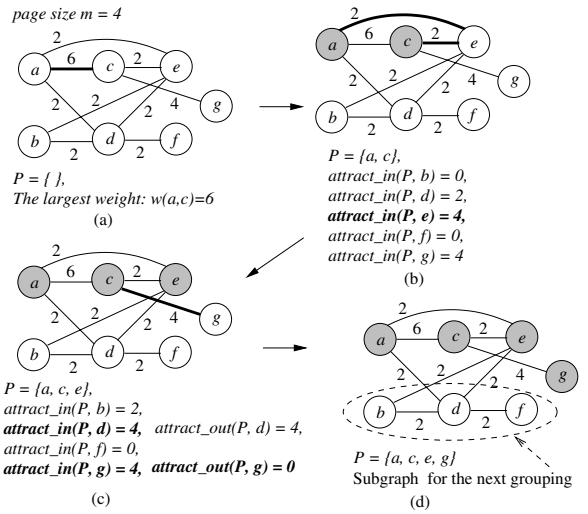

Figure 4: An example illustrating the steps of grouping variables by Solve-MLP**.**

4. MEMORY LAYOUT UTILIZING BURST MODE

4.1 The Problem Formulation

In this section we consider DRAMs which allow burst mode as well as normal mode (e.g., [10]). The memory layout problem is then to maximize the number of burst accesses. We call the problem *MLB* (*Memory Layout with Burst mode*). We first claim that *decision-MLB* is NP-complete.

PROBLEM 4.1. *decision-MLB: For an MLB instance with (S,V,m) and k, is there a memory layout, L, in which the number of burst accesses for S is greater than or equal to k?*

DEFINITION 4.1. *The directed access graph of S is a directed graph $G(V,A)$ where V is the set of variables in S and there are n arcs from v_i to v_j if and only if there are exactly n consecutive references of variables v_i, v_j in S, denoting $w\langle v_i, v_j\rangle$=n.*

Let $S = a_1, a_2, \cdots a_{i-1}, a_i, \cdots a_N$ be a sequence of variable references. Recall that a_i in a memory layout L is a burst access if $f_{page}(v(a_i)) = f_{page}(v(a_{i-1}))$ and $f_{addr}(v(a_i)) = f_{addr}(v(a_{i-1})) + 1$. Thus, if there are n consecutive accesses of variables $v(a_{i-1}), v(a_i)$ (i.e., $v(a_{i-1})$ is accessed immediatedly before $v(a_i)$ n times) , and $v(a_{i-1})$ and $v(a_i)$ are located consecutively in the same page, the n accesses of the $v(a_i)$ will be the burst accesses. (See Figure 5(a) for an example of access sequence and its directed access graph.)

DEFINITION 4.2. *A path cover of $G(V,A)$ is a set of node-disjoint directed paths which collectively cover all the nodes in V. A path cover of size m is a path cover $C^m_{G(V,A)}$ in which every (node-disjoint) path in $C^m_{G(V,A)}$ covers at most m nodes.*

We define a gain of path cover $C^m_{G(V,A)}$:

$$g(C^m_{G(V,A)}) = \sum_{\langle v_i,v_j\rangle \in A,\ v_i \in p,\ v_j \in p,\ p \in C^m_{G(V,A)}} w\langle v_i,\ v_j\rangle \qquad (4)$$

DEFINITION 4.3. *A maximum weighted path cover (MWPC) of size m in $G(V,A)$ is a path cover $C^m_{G(V,A)}$ that maximizes the quantity in Eq.(4).*

The memory layout L implied by a path cover C^m_G is the one that satisfies: Each path in C^m_G is a distinct page in L and the sequence of variables on the path is the relative placement order of the variables in the page. For example, Figure 5(b) is the memory layout implied by the path cover in Figure 5(a) when $m = 3$ where the heavy arrows indicate the path cover.

PROBLEM 4.2. *decision-MWPC: For a directed access graph $G(V,A)$ and k, is there a path cover $C^m_{G(V,A)}$, in which its $g(\cdot)$ is greater than or equal to k?*

THEOREM 4.1. *decision-MWPC is NP-complete. (We showed that* MWPC *is polynomial time reducible from the Hamiltonian path problem.)*

THEOREM 4.2. *decision-MLB is NP-complete. (We showed that decision-MWPC is polynomial time reducible to decision-MLB.)*

From the proof of the previous theorem, we derive the following.

THEOREM 4.3. *The memory layout implied by an MWPC is optimal.*

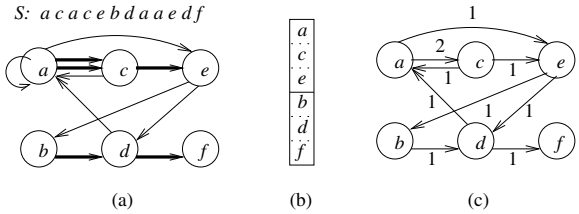

Figure 5: (a) A path cover; (b) The memory layout implied by the path cover in (a); (c) The access graph in a simplified weighted graph form for (a).

For example, Figure 5(a) shows an optimal path cover when m=3 and Figure 5(b) shows the memory layout implied by the path cover. The memory layout leads to 5 burst accesses, which is $w(a,c) + w(c,e) + w(b,d) + w(d,f) = 5$.

4.2 The Proposed Algorithm

Due to the NP-completeness of the problem, we propose an efficient heuristic algorithm, called Solve-MLB, that is similar to the Kruskal's maximum spanning tree algorithm. The inputs to the proposed algorithm are the edge weighted directed access graph $G(V,A,W)$ (e.g., Figure 5(c))[5] and page size m. The algorithm is greedy in that at each step it selects the arc with the largest weight, satisfying (a) it does not create a cycle, (b) it does not increase the in- or out- degree of node to more than one, and (c) it does not make any path length in terms of the number of nodes to be greater than m. If there are more than one arc that satisfy the conditions (a), (b) and (c), for each $\langle u,v\rangle$ of the arcs we compute

$$attract\_out\_burst(C, \langle u,v\rangle) = \sum_{\forall \langle x,y\rangle \ of\ types\ 1\ and\ 2} w\langle x,y\rangle \qquad (5)$$

where the arc types are illustrated in Figure 6. Note that node b in type 1 arc is not in the current path cover whereas node c in type 2 arc is in the current path cover. The arcs of types 1 and 2 with respect to candidate arc $\langle u,v\rangle$ are the ones that should be removed

[5]Note that when the directed access graph in multigraph form is transformed into a simple directed access graph, the self-loops are simply deleted because the removals have nothing to do with the path cover problem of maximizing $g(C^m_G)$ in *Eq.(4)*.

for further consideration if $\langle u,v\rangle$ were selected to be an element of path cover due to path and/or length violations. We select the arc among ties with the least value of $attract\_out\_burst(\cdot)$.

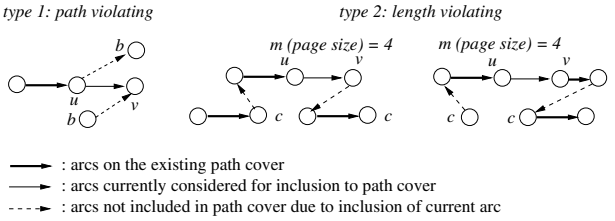

Figure 6: Arc types used to break the ties in Solve-MLB.

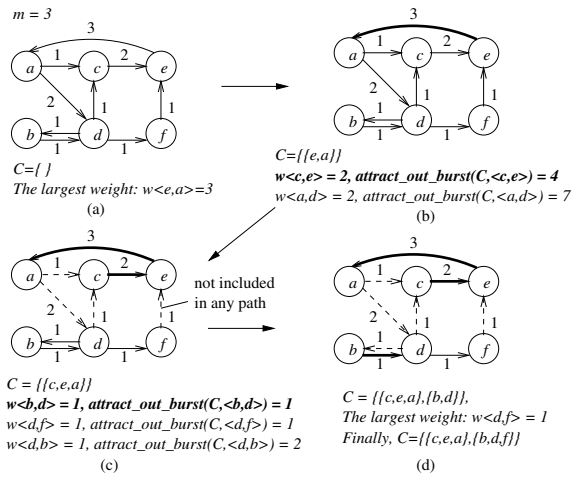

Figure 7: An example illustrating the steps of finding a path cover by Solve-MLB.

Figure 7 shows the steps of finding path cover by Solve-MLP. First, we select the arc $\langle e,a\rangle$ because it has the largest arc weight and set $C = \{e,a\}$ as shown in Figure 7(a). Then, we collect the arcs with the next largest weight. Those arcs are $\langle c,e\rangle$ and $\langle a,d\rangle$. Among them we select $\langle c,e\rangle$ and add it to path cover C because the value of its $attract\_out\_burst(C,\cdot)$ is smaller, as indicated in Figure 7(b). Similarly, in the next iteration we include arc $\langle b,d\rangle$ to C as shown in Figure 7(c). Finally, there remains only one candidate arc $\langle d,f\rangle$, and the arc is added to C as shown in Figure 7(d), resulting the final path cover being $\{c \to e \to a,\ b \to d \to f\}$.

5. EXPERIMENTAL RESULTS

We conducted a set of experiments to check the effectiveness of the proposed memory layout optimization algorithms Solve-MLP and Solv-MLB. Our algorithm was implemented in C++ and executed on a Pentium-3 500MHz Linux machine. We tested our algorithm on a set of benchmark programs in [16, 17, 18], and then compared our results with that produced by the OFU (the order of first use) variable assignment and the CGB (the closeness-graph-based) algorithm [1, 2]. OFU is a naive approach which places variables to memory in the order as they apear in the code, and CGB algorithm clusters variables into pages based on the degree of 'closeness' between nodes (variables) in a graph. Note that the edge weight formulation in the closeness-graph model is totally different with that of our access graph models (See Figure 3).

• **Memory optimization for maximizing DRAM's page accesses**: Table 1 summarizes the number of page accesses used by OFU, CGB and Solve-MLP with various page sizes. $|V|$ and $|S|$ in the first column of the table represent the number of variables and the access sequence length, respectively. BIQUAD and FIR are taken from DSPstone suite [16]. COMP and ELLIP are taken from [17]. GAULEG, GAUHER, GAUJAC, CHEBEV and LMS are taken from [18]. The entries marked with '-' indicate that a single page contains all the variables of the corresponding design. In summary, the average improvements by Solve-MLP over OFU and CGB [1, 2] are 20.0% and 9.9%, respectively.

• **Memory optimization for maximizing DRAM's burst accesses**: Table 2 summarizes the number of burst accesses used by OFU, CGB and Solve-MLB with various page sizes. The average improvements by Solve-MLB over OFU and CGB are 54.0% and 86.6%, respectively. The large reduction numbers cleraly indicate that Solve-MLB is performing well in maximizing the number of burst accesses.

• **Checking the reductions on total memory access latency and energy consumption**: Figure 8 graphically shows how much the total memory access latency and energy consumption are actually saved by our Solve-MLP over the conventional methods. We used the access figures in IBM *Cu-11 Embedded DRAM* [9] macro as reference in the experiments, in which a random (i.e., normal) access cycle is 12ns at best and a page access is 4ns at best. In addition, under the supply voltage of 1.5V an active current is 59.19mA on average for a random access, and 11.21mA for a page access.

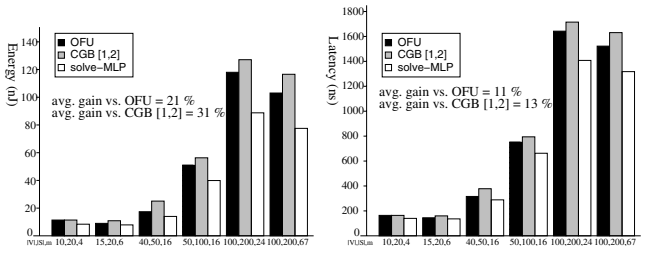

Figure 8: The comparisons on total DRAM access energy consumption and latencies.

6. CONCLUSIONS

In this paper, we proposed a set of comprehensive solutions to the problems of storage assignment for variables to achieve a full utilization of two efficient DRAM access modes: page mode and burst mode. Since memory access is one of the major bottlenecks in embedded systems' performance, the proposed techniques can be effectively used in DRAM-access-intensive embedded system applications. The contributions are, for each of page and burst modes: (1) We showed that the problem is NP-hard; (2) We propose direct and exact formulation of the problem and *efficient memory layout algorithms*, called Solve-MLP for the page mode and Solve-MLB for the burst mode; From a set of experiments with benchmark examples, it was shown that the proposed techniques used on average 9.9%-20.0% more page accesses and 54.0%-86.6% more burst accesses over the existing techniques.

Table 1: The numbers of page accesses used by OFU, CGB[1,2] and Solve-MLP.

design ($\|V\|/\|S\|$)	#page_accesses (OFU / CGB[1, 2] / Solve-MLP) m: page size							gain of Solve-MLP (%)	
	m = 3	m =4	m =6	m =8	m =10	m =12	m =16	over OFU	over CGB
BIQUAD(10 / 38)	18/19/19	19/23/23	22/27/29	24/33/33	-	-	-	24.0	1.9
CHEBEV(12 / 33)	14/19/19	15/21/23	22/24/24	24/20/26	28/28/26	-	-	19.9	6.5
COMP(10 / 18)	6/8/8	7/7/9	12/12/12	14/14/14	-	-	-	15.5	7.1
ELLIP (45 / 100)	19/39/42	21/37/45	34/39/52	38/46/61	48/46/63	54/52/68	58/65/76	62.4	25.7
FIR(5 / 2)	10/10/14	15/15/15	-	-	-	-	-	20.0	20.0
GAUHER(11 / 59)	31/38/38	40/43/43	42/44/48	46/48/48	-	-	-	12.2	2.3
GAUJAC(23 / 148)	56/67/68	63/71/80	84/81/95	104/87/106	107/107/115	117/112/116	125/125/128	10.4	9.5
GAULEG(16 / 47)	20/27/26	26/25/34	31/31/34	36/36/41	36/42/43	43/38/44		17.7	12.3
LMS(8 / 30)	19/17/19	20/19/19	25/25/25	-	-	-	-	-1.7	3.9
avg. gain								**20.0**	**9.9**

Table 2: The numbers of burst accesses used by OFU, CGB[1,2] and Solve-MLB.

design ($\|V\|/\|S\|$)	#burst_accesses (OFU / CGB[1, 2] / Solve-MLB) m: page size								gain of Solve-MLB (%)	
	m = 3	m =4	m =6	m =8	m =10	m =12	m= 14	m =16	over OFU	over CGB
BIQUAD (10 / 38)	8/9/9	10/10/10	10/6/11	11/6/11	11/6/12	-	-	-	6.3	53.3
CHEBEV (12 / 33)	7/7/10	9/9/10	10/6/11	10/6/11	11/6/12	11/6/12	-	-	15.4	70.1
COMP (10 / 18)	5/5/6	5/3/7	6/5/7	6/4/8	6/3/8	-	-	-	28.7	92.0
ELLIP (45 / 100)	9/13/24	8/11/29	11/13/30	10/4/33	11/8/33	11/10/33	10/13/34	11/7/34	210.1	274.2
FIR (5 / 20)	5/5/6	5/5/8	-	-	-	-	-	-	40.0	40
GAUHER (11 / 59)	8/12/13	11/13/16	11/14/16	12/7/17	12/8/17	-	-	-	47.3	60.2
GAUJAC (23 / 148)	13/25/25	15/26/29	17/28/31	17/24/34	17/27/34	15/20/34	17/19/34	-	92.0	34.1
GAULEG (16 / 47)	9/13/13	13/11/15	14/8/17	14/6/17	12/11/18	15/13/18	14/13/18	15/9/18	27.7	71.6
LMS (8 / 30)	7/4/8	6/5/8	8/4/9	8/6/9	-	-	-	-	18.2	83.8
avg. gain									**54.0**	**86.6**

Acknowledgment : This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center(AITrc).
We would like to thank HS. Kim for his constructive discussion and comments on this work.

7. REFERENCES

[1] P. R. Panda *et al*., "Exploiting Off-Chip Memory Access Modes in High-Level Synthesis," *ICCAD*, 1997.
[2] P. R. Panda *et al*., "Memory Data Organization for Improved Cache Performance in Embedded Processor Applications," *ACM TODAES*, 1997.
[3] N. D. Dutt, "Memory Organization and Exploration for Embedded Systems-on-Silicon," *Inter. Conf. on VLSI and CAD,* 1997.
[4] T. Mowry *et al*., "Design and Evaluation of a Compiler Algorithm for Prefetching," *ASPLOS,* 1992.
[5] M. S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *PLDI,* 1988.
[6] A. W. Appel, *Modern Complier Implementation in C,* Cambridge, 1998.
[7] S. Wuytack *et al*., "Flow Graph Balancing for Minimizing the Required Memory Bandwidth," *ISSS*, 1996.
[8] F. Catthoor *et al*., *Custom Memory Management Methodology,* Kluwer Academic Publisher, 1998.
[9] IBM, "IBM Cu-11 Embedded DRAM Macro," *http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/4CBB96F927E2D6D287256B98004E1D98/$file/Cu11_embedded_DRAM.10.pdf,* 2002.
[10] Fujitsu, "CS70DL Embedded DRAM," *http://www.fme.fujitsu.com/products/asic/pdf/CS70DLFS.pdf,* 1999.
[11] A. Khare *et al*., "High-Level Synthesis with Synchronous and RAMBUS DRAMs," *SASIMI*, 1998.
[12] P. Grun *et al*., "Memory Aware Compilation Through Accurate Timing Extraction," *DAC*, 2000.
[13] P. Grun *et al*., "APEX: Access Pattern Based Memory Architecture Exploration," *ISSS*, 2001.
[14] K. Ayukawa *et al*., "An Access Sequence Control Scheme to Enhance Random-Access Performance of Embedded DRAMs," *IEEE Journal of Solid- State Circuits*, 1998.
[15] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness,* W. H. Freeman and Company, New York. pp.209, 1979.
[16] V. Zivojnovic, *et al*., "Dspstone: A DSP-oriented Benchmarking Methodology," *International Conference on Signal Processing Applications and Technology*, 1994.
[17] "Bench mark Archives at CBL," *http://www.cbl.ncsu.edu /CBL_Docs/Bench.html*
[18] W. H. Press, *et al.* (Editors), *Numerical Recipes in C: The Art of Scientific Computing,* Cambridge University Press, pp.152,154-155, 1993.