Memory Access Scheduling and Binding Considering Energy Minimization in Multi-Bank Memory Systems

Chun-Gi Lyuh
Basic Research Laboratory,
Electronics and Telecommunications
Research Institute,
Daejeon, Korea
cglyuh@etri.re.kr

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

ABSTRACT

Memory-related activity is one of the major sources of energy consumption in embedded systems. Many types of memories used in embedded systems allow multiple operating modes (e.g., active, standby, nap, power-down) to facilitate energy saving. Furthermore, it has been known that the potential energy saving increases when the embedded systems use multiple memory banks in which their operating modes are controlled independently. In this paper, we propose (a compiler-directed) integrated approach to the problem of maximally utilizing the operating modes of multiple memory banks by solving the three important tasks simultaneously: (1) *assignment of variables to memory banks*, (2) *scheduling of memory access operations*, and (3) *determination of operating modes of banks*. Specifically, for an instance of tasks 1 and 2, we formulate task 3 as a shortest path(SP) problem in a network and solved it optimally. We then develop an SP-based heuristic that solves tasks 2 and 3 efficiently in an integrated fashion. We then extend the proposed approach to address the limited register constraint in processor. From experiments with a set of benchmark programs, we confirm that the proposed approach is able to reduce the energy consumption by 15.76% over that by the conventional greedy approach.

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

General Terms: Algorithms, Design

Keywords: low energy design, scheduling, binding

1. INTRODUCTION

Lowering down the energy consumption is a major design objective in many of today's embedded system-on-chips (SoCs) design. However, traditionally, various sophisticated processor-to-memory architectures have been designed and used to better meet the performance requirement (rather than energy requirement) of the target applications. Among them, the architecture with *multiple memory banks* is one of the most effective features in increasing the system's performance. For example, *Motorola DSP 56000* and *NEC 77016 DSP* support this feature, and increase memory bandwidth by allowing multiple data memory accesses to occur concurrently when the variables to be accessed belong to different memory banks. An extensive research work has been done in the area of parallelizing variable accesses among multiple memory banks. [1, 2]

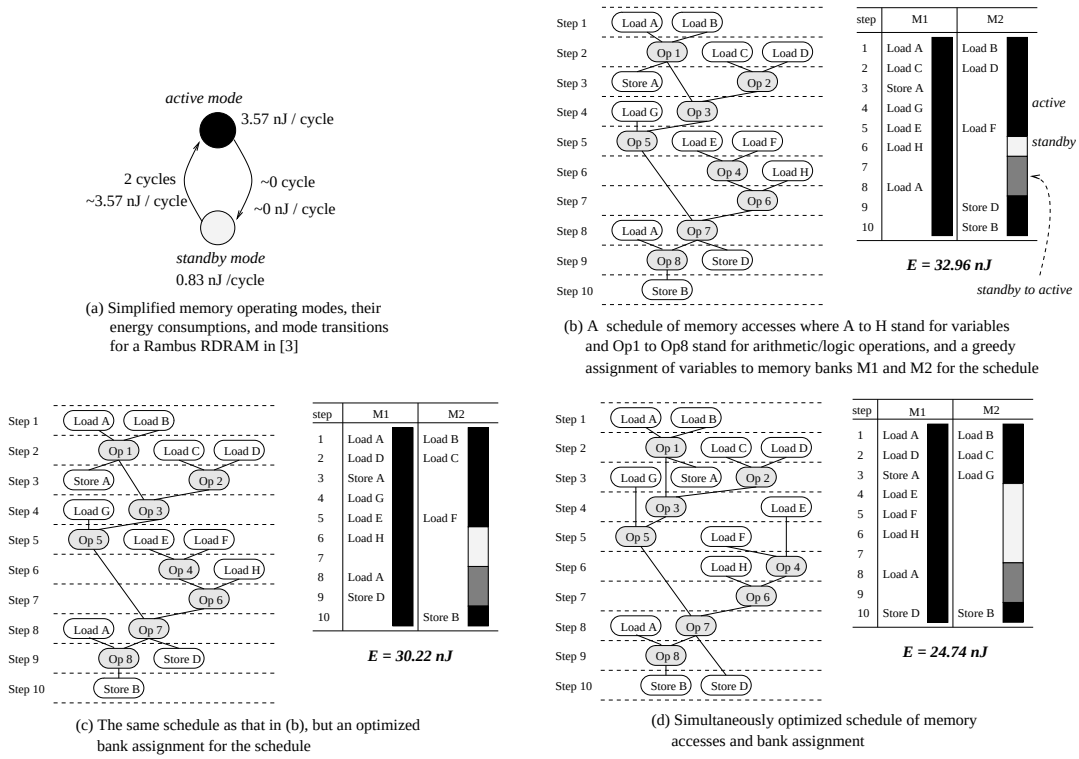
Recently, advanced low power memory architectures are designed to operate in multiple states (e.g., active, standby, nap, and power-down modes). However, the issue of minimizing energy consumption by exploiting the architecture with multiple memory banks combined with the operating modes of memory banks has not been (relatively) fully addressed in the literature, but the issue is very useful or critical in saving energy consumption in embedded systems, particularly those for computation-intensive applications. De La Luz, Kandenir, and Kolcu [3] proposed an automatic data migration to reduce energy consumption in multiple memory banks by exploiting the temporal affinity among data. It assumes array is a basic unit of data to be considered for migration. It also assumes that the schedule of memory accesses is fixed. Benini, Macii, and Poncino [4] proposed an algorithm for low-energy partitioning of memory into multiple banks for a given schedule of memory accesses in code. In terms of utilizing the power modes, Benini *et al*. [5] proposed an execution profile based energy-optimal algorithm for the automatic partitioning of on-chip SRAMs into multiple banks. In addition, Lu, Benini, and De Micheli [6] proposed a low-power task scheduling technique for multiple devices with multiple power modes. De La Luz *et al*. [7] proposed an operating system (OS) based power mode transition scheme in memory power-mode management. They [8] also proposed a number of compiler directed techniques and applied them sequentially to solve the problem of bank assignment and operating mode selection.

In this paper, we propose a compiler-directed integrated approach to the problem of maximally utilizing the power modes of memory banks by solving the following three important tasks *simultaneously*: (1) *variable assignment to memory banks*, (2) *scheduling memory access operations*, and (3) *selecting operating modes of banks*. Those three tasks are closely interrelated and their results affect each other in significant ways. Consequently, our integrated approach is very necessary and desirable for achieving a globally maximal utilization of power modes of memory.

2. MOTIVATING EXAMPLE

We give an example to illustrate how scheduling of memory accesses and variable assignment to multiple banks affect the energy consumption of the memory. Consider a Rambus DRAM (RDRAM) module [9], which is designed to operate in two differ-

DAC 2004, June 7–11, 2004, San Diego, California, USA.



(a) Simplified memory operating modes, their energy consumptions, and mode transitions for a Rambus RDRAM in [3]

(b) A schedule of memory accesses where A to H stand for variables and Op1 to Op8 stand for arithmetic/logic operations, and a greedy assignment of variables to memory banks M1 and M2 for the schedule

(c) The same schedule as that in (b), but an optimized bank assignment for the schedule

(d) Simultaneously optimized schedule of memory accesses and bank assignment

Figure 1: Motivating examples illustrating the effects of memory access scheduling and memory binding on energy consumption.

ent modes: *active* and *standby* modes; Memory access operations will be performed only when the memory is in active mode. The energy consumption in active mode is 3.57 nJ per cycle. If the memory is in standby mode, it consumes much less energy, 0.83 nJ per cycle, but is required to be switched to an active mode to perform a memory access operation. The mode transition from active to standby takes negligible amounts of time and energy, but transition from standby to active takes two clock cycles and almost the same amount of energy as that in active mode [9].[1] Figure 1(a) describes the mode transition behavior of a RDRAM module [9].

Suppose a designer uses two RDRAM banks (say $M1$ and $M2$) in [9], that follows the mode transitions specified in Figure 1(a). Figure 1(b) shows a schedule of variable access operations and the assignment of the variables according to the order of first appearance in code with the schedule to memory banks $M1$ and $M2$. The shaded boxes at the right side of the figure indicate the operating modes of the corresponding memory banks and clock steps. The total energy consumed in $M1$ and $M2$ is 32.96 nJ. On the other hand, Figure 1(c) shows an energy-optimized memory binding for the same schedule as that in Figure 1(b). The energy consumption is reduced from 32.96 nJ to 30.22 nJ, which is 8.3% reduction. A further energy saving can be achieved by considering scheduling and memory binding together, as indicated in Figure 1(d), in which the energy consumption is reduced to 24.74 nJ, which is 24.9% and 18.1% reductions over those in Figures 1(b) and (c), respectively. This example clearly reveals that it is very necessary to take into account memory access scheduling and memory binding simultaneously to fully exploit the memory operating modes to minimize the energy consumed in memory.

It should be noted that embedded processors with multiple memory banks have a limited number of registers to hold the values of variables that are accessed from/to memory. That means memory access scheduling should be constrained by the registers available to use. For example, in Figures 1(d), *Load G* operation is executed at clock step 3, and the value of G is stored to a register until clock step 5 since the value will be used at clock step 5 as input to *Op5*. We can easily see that a schedule of memory access operations determines the intervals of clock steps at which the values of variables accessed should be stored temporarily. The intervals of clock steps for the schedule in turn determine the minimum number of registers required to store the values of variables.[2]

3. THE PROBLEM FORMULATION

We assume that scheduling of computation steps for arithmetic/ logic operations to functional units has already been done. In other words, the clock steps at which the data values of variables that are read from memory are to be used and the values that are to be written to memory are generated are known. However, from/to which memory banks and at which clock steps the memory access operations are executed have not yet been determined. For conciseness of presentation, we consider dual memory banks and non-array variable accesses.[3]

Let $\mathcal{S}(G)$ be a schedule of memory access operations in a data-flow graph (DFG) G and $\mathcal{B}(\mathcal{S})$ be the memory bank binding of variables whose accesses are executed according to $\mathcal{S}(G)$. Furthermore, $f_{mode}(\mathcal{S}, \mathcal{B}, M_i, c_j)$ denotes the operating mode of memory bank M_i at clock step c_j for schedule $\mathcal{S}$ and binding $\mathcal{B}$, and $E(f_{mode}(\mathcal{S}, \mathcal{B}, M_i, c_j))$ denote the amount of energy consumed at c_j in bank M_i with memory mode $f_{mode}(\cdot)$.

[1] Note that the mode transition overheads (i.e., time, energy) as well as the number and types of memory operation modes vary depending on the technologies used in designing memories. Our proposed approach can be parameterized with the varying factors, so that it is applicable to any kind of memories with multiple operating modes.

[2] The issue of addressing the register constraint will be discussed in section 4.2.

[3] Extension of our proposed technique to the problem with an arbitrary number of memory banks and/or array variable accesses is rather straightforward.

Then, the optimization problem we want to solve is to find a schedule $\mathcal{S}$, a binding $\mathcal{B}$, and modes $f_{mode}(\cdot)$ that minimize the cost function:

$$E_{total} = \sum_{i=1}^{K} \sum_{j=1}^{T} E(f_{mode}(\mathcal{S}, \mathcal{B}, M_i, c_j)) \quad (1)$$

where K (=2) is the number of memory banks and T is the latency of G.

4. THE PROPOSED APPROACH

We propose an iterative technique, which consists of two phases:

• *Initial Phase*: We obtain an initial result of $\mathcal{S}$ and $\mathcal{B}$ by applying a simple list scheduling to memory access operations in G and a subsequent application of a greedy binding to the variables, from which we compute, for each memory bank, f_{mode} at each clock step *optimally*, minimizing E_{total} of the $\mathcal{S}$ and $\mathcal{B}$.

• *Refinement Phase*: We divide scheduling task into two subtasks, called *r-scheduling* and *a-scheduling*. R-scheduling determines a *relative (linear) order* among the memory access operations to a bank, and a-scheduling determines an *absolute order* among the operations that have relatively been ordered by the r-scheduling. Given an instance of binding $\mathcal{B}$ of variables and an r-schedule $\mathcal{S}_r$ of $\mathcal{B}$, we generate as many alternative bindings of variables and r-schedules which are slightly different from $\mathcal{S}_r$ of $\mathcal{B}$ as possible, and obtain the corresponding E_{total} value by calculating an *optimal* a-schedule $\mathcal{S'}_a$ and f_{mode} for each of the r-schedules of the alternative bindings. Among the candidate bindings and r-schedules, we choose the binding $\mathcal{B'}$ and r-schedule $\mathcal{S'}_r$ of $\mathcal{B'}$ that minimize the value of E_{total}. By setting $\mathcal{S}_r$ = $\mathcal{S'}_r$ and $\mathcal{B}$ = $\mathcal{B'}$, we repeat the process until there is no more reduction in the quantity of E_{total}.

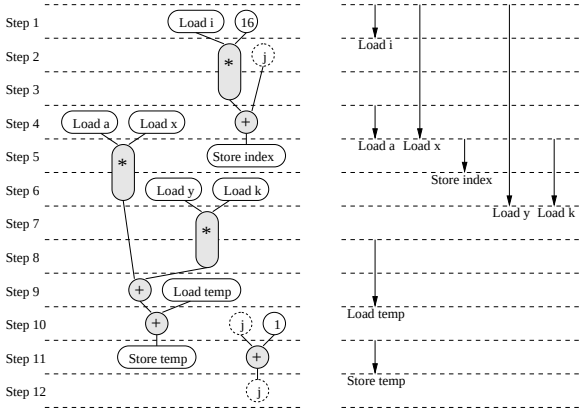
In the following, we provide the details of the proposed technique. First, we assume that the number of registers for storage is sufficiently large enough to hold all data values that are accessed from/to memory banks at any clock steps. That means the memory access operations are to be scheduled to any of the clock steps only if the load accesses occur before the use of their values and the store accesses occur after the creation of their values. [4]

We describe our stepwise refinement technique using the example in Figure 2. Consider the DFG G in Figure 2(a), in which the arithmetic/logic operations have already been scheduled. Figure 2(c) shows an instance of memory binding $\mathcal{B}$ and r-schedule $\mathcal{S}_r$ of the memory access operations in G, which was initially derived by using a simple greedy scheme. That is, $\mathcal{B}$ is $\{M1$: i, a, $index$, y, $temp$; $M2$: x, $k\}$, and $\mathcal{S}_r$ is $\{M1$: *Load_i* → *Load_a* → *St_index* → *Load_y* → *Load_temp* → *St_temp*; $M2$: *Load_x* → *Load_k*$\}$. Figure 2(d) shows our network formulation for solving an optimal a-schedule, $\mathcal{S}_a$, of $\mathcal{B}$ and $\mathcal{S}_r$. (The detailed network construction and solution will be presented in section 4.1.) Let us define two types of primitive operations that can be applied to $\mathcal{B}$ and $\mathcal{S}_r$.

Definition: switch, denoted by $M1Ci \leftrightarrow M2Cj$, switches the i^{th} access operation in $\mathcal{S}_r$ of M_1 with the j^{th} access operation in $\mathcal{S}_r$ of $M2$, and move, denoted by $M1Ci \rightarrow M2Cj$ (or $M2Ci \rightarrow M1Cj$), moves the i^{th} access operation in $\mathcal{S}_r$ of $M1$ (or $M2$) to the position of the j^{th} access operation in $\mathcal{S}_r$ of $M2$ (or $M1$). Note that the legal switch and move operations should not violate the operation dependencies in DFG.

Our iterative process consists of two loops, one nesting the other: The *inner-loop* performs the following: It first generates the in-

[4]Our extension to the memory access scheduling and bank binding with the register constraint will be discussed in section 4.2.

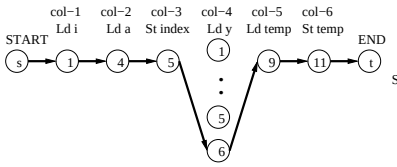


(a) A (arithmetic/logic operation−scheduled) DFG. The numbers in circles represent constants, and the dotted circles represent register access operations.

(b) The intervals of clock steps at which the memory memory access operations in (a) can be executed. (i.e., the mobility of memory access operations).

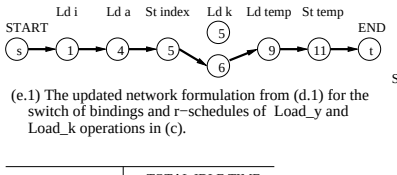
M1 : {Load i > Load a > Store index > Load y > Load temp > Store temp} M2 : {Load x > Load k}

(c) A result by greedy memory binding and r−schedule



(d.1) The network formulation for a−schedule of the binding and r−schedule in memory M1 of (c).

(d.2) The network formulation for a−schedule of the binding and r−schedule in memory M2 of (c).



(e.1) The updated network formulation from (d.1) for the switch of bindings and r−schedules of Load_y and Load_k operations in (c).

(e.2) The updated network formulation from (d.2) for the switch of bindings and r−scehdule of Load_y and Load_k operations in (c).

SWITCH/MOVE	TOTAL IDLE TIME		
	M1	M2	M1+M2
M1C1 −> M2C1	36.53	28.31	64.84
M1C2 −> M2C2	36.53	28.31	64.84
M1C3 −> M2C2	36.53	28.31	64.84
M1C4 −> M2C1	36.53	28.31	64.84
M1C4 −> M2C2	36.53	28.31	64.84
M1C4 −> M2C3	36.53	28.31	64.84
M2C1 −> M1C2	39.27	22.83	62.10
M1C1 <−> M2C1	39.27	25.57	64.84
M1C2 <−> M2C1	39.27	28.31	67.58
M1C4 <−> M2C2	39.27	20.09	**59.36** √
(M1C5 −> M2C3) (M1C6 −> M2C4)	31.05	33.79	64.84

(f) The optimal idle time period for minimum energy consumption for each of the posssible switches (denoted by <−>) and moves (denoted by −>), as shown the example in (e), from the binding and r−scheudle in (c). The maximum energy saving is obtained when Load_y operation in the fourth column in (d.1) is switched with Load_k operation in the second column in (d.2) (i.e., M1C4 <−> M2C2).

step	M1	M2
1	Load i	Load x
2		Load y
3		
4	Load a	
5	Store index	
6	Load k	
7		
8		
9	Load temp	
10		
11	Store temp	

(g) The resultant binding, a−schedule, and the memory states by M1C4<−> M2C2 (The active, transition, idle−states are indicated by black, heavy gray, and light gray colors).

Figure 2: An example illustrating our iterative refinement procedure of binding and scheduling.

stances of all bindings and r-schedules that can be obtained by the application of a switch or a move operation to $\mathcal{B}$ and $\mathcal{S}_r$, and find their optimal a-schedules. Then, among the instances of bindings and schedules, it chooses the one with the least energy consumption, and applies the corresponding switch or move operation to $\mathcal{B}$ and $\mathcal{S}_r$, producing $\mathcal{B}^{new}$ and $\mathcal{S}_r^{new}$. The access operations that were involved are then locked. This process repeats from $\mathcal{B}^{new}$ and $\mathcal{S}_r^{new}$, and the iteration stops when all the access operations are locked or no more switch or move can be applied without violating the operation dependencies in DFG. The best instance of binding and schedule is the one with the least energy consumption among the instances obtained during the iterations. The *outer-loop* then unlocks all the access operations, and sets the best instance to be an initial instance of the next *inner-loop*. The *outer-loop* stops when there is no more reduction in energy consumption.

For example, consider switch operation $M1C4 \leftrightarrow M2C2$ to the binding and r-schedule in Figure 2(c). That means, *Load_y*, which is the fourth access operation in $M1$ is switched with *Load_k*, which is the second access operation in $M2$. We then determine an optimal a-schedule for the updated binding and r-schedule corresponding to $M1C4 \leftrightarrow M2C2$ using network formulation, as indicated in Figures 2(e.1) and (e.2). (The details about the formulation will be described in section 4.1.) The table in Figure 2(f) summarizes the energy consumptions for the rebindings and reschedules produced by the application of all legal switch/move operations to the binding and schedule in Figure 2(c). Among the switch/move operations, $M1C4 \leftrightarrow M2C2$ produces the result with the least energy consumption. Finally, Figure 2(g) shows the resultant binding and schedule corresponding to $M1C4 \leftrightarrow M2C2$. The *Load_y* and *Load_k* operations are then locked, and, the process will repeat from the binding and r-schedule of Figure 2(g).

The key feature of our procedure is to compute the energy consumption for every instance of bindings and r-schedules *quickly*, but *optimally*. This feature, which is the subject of the next subsection, enables us to fully explore the search space in a reasonable amount of time to find globally good solutions.

We call the overall procedure of the proposed technique MACCESS-lp. The time complexity of the *inner-loop* is theoretically bounded by $O(N^5T^4)$ where N is the number of memory access operations and T is the latency of G. However, the complexity of *inner-loop* is reduced to $O(N^4T^2)$ due to the incremental fast network flow computation, and in practice, the complexity is further reduced to that close to (N^3) since the mobility of an access operation is much shorter than T due to the fixed schedule of arithmetic/logic operations. Note that the value of N (i.e., the number of access operations) is less than 100 in practice.

4.1 Optimal Cost Computation

• **SP-based network formulation**: Let us consider the binding $\mathcal{B}'$ and r-schedule $\mathcal{S}'_r$ in Figure 2(e) which is a local update of $\mathcal{B}$ and $\mathcal{S}_r$ in Figure 2(d) by switch operation: $M1C4 \leftrightarrow M2C2$ (i.e., *Load_y* $\leftrightarrow$ *Load_k*). We will describe an SP-based network formulation using the example in Figure 2(e.2) to find a-schedule, $\mathcal{S}'_a$, of $M2$ for minimum energy consumption.

We construct network G'_{sp} of $M2$ for $\mathcal{B}'$ and $\mathcal{S}'_r$ as follows: For the i^{th} access operation op_i to $M2$ in $\mathcal{S}'_r$, if op_i can be scheduled to any of k_i clock steps, G'_{sp} contains k_i nodes, one for each clock step. The k_i nodes are then arranged vertically, placing the nodes corresponding to the first access operation in $\mathcal{S}'_r$ to the first column of G'_{sp} and the nodes corresponding to the last in $\mathcal{S}'_r$ to the last column. We then create two dummy nodes, *start* and *end* nodes, and place them to the front and end of G'_{sp}, respectively. Figure 2(e.2) shows the node arrangement. For example, since *Load_x* can be scheduled to any of clock steps 1, 2, 3, and 4 for execution, G'_{sp} contains 4 nodes in the first column. For each pair of nodes $op_{i,s1}$ and $op_{i+1,s2}$ where $op_{i,s1}$ is a node in the i^{th} column of clock step $s1$, $op_{i+1,s2}$ is a node in the $i+1^{th}$ column of clock step $s2$, and $s1 < s2$, G'_{sp} has an arc from $op_{i,s1}$ to $op_{i+1,s2}$. G'_{sp} also contains arcs from *start* node to every node in the first column, and arcs from every node in the last column to *end* node. We then assign cost to each arc. The cost assigned to arc $op_{i,s1} \rightarrow op_{i+1,s2}$ represents the amount of energy that can be minimally consumed during the clock step period of $[s1+1, s2]$.[5] For example, in Figure 3, the cost assigned to the arc from node a to b is 11.54, which is the amount of energy dissipated minimally in $M2$ for the clock step period of [3, 6]. Precisely, $M2$ stays in standby mode in clock step 3 and performs a mode transition in steps 4 and 5, and stays in active mode in step 6 to execute *Load_y*, resulting in the total amount of energy consumption for the period of [3, 6] being 0.83 (for standby) + 3.54×2 (for mode transition) + 3.54 (for active) = 11.54 nJ.

Then, the problem of finding an energy-minimal a-schedule can be solved by finding a shortest path from *start* to *end* in G'_{sp}. The heavy arrows in Figure 3 indicate a shortest path, whose length is 20.09, which is equivalent to the amount of minimal energy consumed in $M2$ for $\mathcal{B}'$ and $\mathcal{S}'_r$ using memory operating modes.

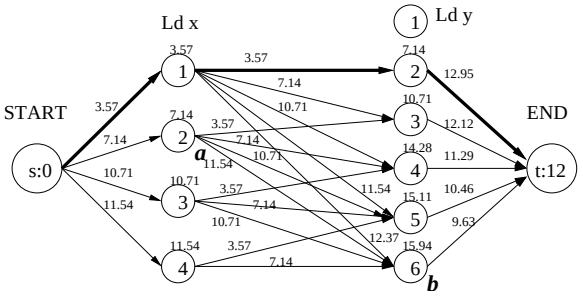


Figure 3: The SP-based network with arc costs for the example in Figure 2(e.2).

• **Fast incremental SP computation**: The time complexity of a single-source SP computation in a network is $O(N^2)$ [10] where N is the number of nodes in the network. However, if we examine the structure of our SP-based network, the SP computation can be done in a linear time.

The structure of the SP-based network is a sequence of columns of nodes, in which arcs exist only among the nodes in two consecutive columns. In this special structure, for each node n_i, we compute two numbers l_{start} and l_{end}, where l_{start} is the length of a shortest path from *start* to n_i, and l_{end} is the length of a shortest path from n_i to *end*. We compute the l_{start} values of nodes from *start* to *end* in $O(N)$ time,[6] and in the same way, compute the l_{end} values of nodes from *end* to *start* in $O(N)$ time.[6] The shortest path length from *start* to *end* is then the l_{end} value of *start* or the l_{start} value of *end*. We call this SP computation is called a *sweep-based* SP computation.

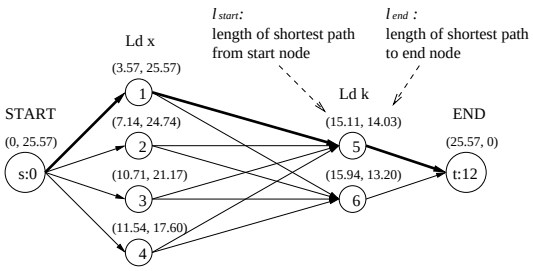
Figure 4(a) shows an example of l_{start} and l_{end} values of nodes in the SP-based network in Figure 2(d.2). Now, suppose a column of nodes is inserted to the SP-based network by the application of move operation to *Load_y*. Figure 4(b) shows the resultant SP-

[5]For simplicity, we assume that each access operation takes one clock cycle. In addition, without loss of generality, we can assume that *start* and *end* nodes are scheduled in clock steps 0, and T (latency), respectively.

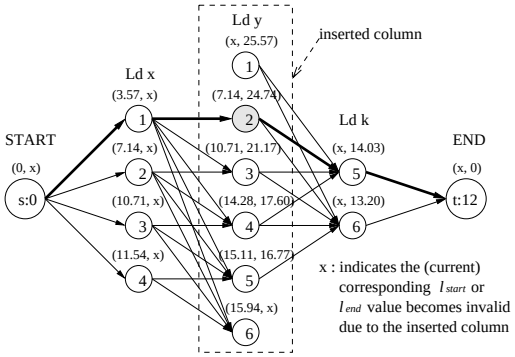
[6]We assume the number of nodes ($<< T$, in practice) in each column is constant bounded.

based network. The l_{start} and l_{end} values of nodes in the inserted columns can be computed in a constant time. Then, the length of the shortest path on the updated network is the minimum value of $l_{start} + l_{end}$ among the nodes in the inserted column, which is $17.14 + 24.74$ ($l_{start} + l_{end}$ of shaded node).

To perform the next iteration of the trials of switch/move operations from an updated network, we need to compute the l_{start} or l_{end} values corresponding to the x's in Figure 4. That is, the l_{end} values of the nodes in the left side (i.e., towards *start* node) of the inserted column and the l_{start} values of the nodes in the right side (i.e., towards *end* node) of the inserted column can be obtained in $O(N)$ time by using the sweep-based SP computation.



(a) Calculating the lengths of shortest paths from start to nodes (i.e., l_{start}) and from nodes to end (i.e., l_{end})



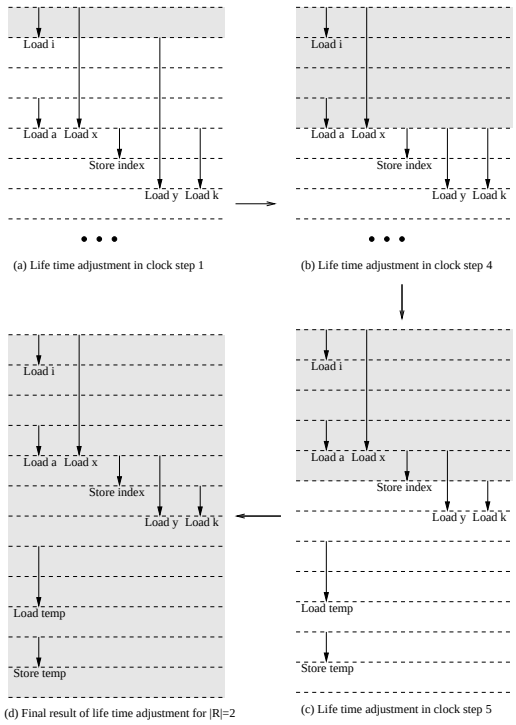
(b) Updating the lengths of shortest paths l_{start} and l_{end} corresponding to the nodes in the inserted column.

Figure 4: The SP-based network construction for the example in Figure 2(e.2).

4.2 Consideration of Register Constraint

Real processors have a limited number of registers to use for storing temporary data values. Let $|R|$ denote the number of registers available to use for load/store. The limited number of registers implies that, for each clock step, the number of data values that have to be placed maximally in registers at that clock step is $|R|$. Consequently, our SP-based approach should take into account the register constraint. We solve this issue by adjusting the life times of memory access operations so that the SP-based approach generates solutions that are immune to the violation of register constraint.

For example, Figure 5 shows a step by step procedure of adjusting the life times of access operations, initially starting from that in Figure 2(b), from the first clock step to the last. Suppose $|R|$=2 and let D_i be the set of variables whose life times contain clock step i. Then, since $D_1 = \{Load\_i, Load\_x, Load\_y\}$ (i.e., $|D_1| = 3 > |R| = 2$, we select a variable from D_1 and delete it from D_1 to make $|D_1| = |R| = 2$. The variable to be selected is the one with the least urgency in its use. Consequently, the variable selected is *Load_y* because its use is at clock step 7 while the uses of *Load_i* and *Load_x* are at clock steps 2 and 5, respectively. Figure 5(a) shows the life times updated at clock step 1 from the original life times in Figure 2(b). The shaded region indicates the clock step that is immune to the violation of register constraint. Then, the process repeats for the next clock step j at which $|D_j| > |R|$ until the region corresponding to the entire clock steps is shaded. Figure 2(b) and (c) show the next life time adjustments in a sequence, and Figures 2(d) shows the final adjustment of life times of variables, from which we can apply our SP-based network solution safely with no violation of register constraint.



(a) Life time adjustment in clock step 1

(b) Life time adjustment in clock step 4

(c) Life time adjustment in clock step 5

(d) Final result of life time adjustment for |R|=2

Figure 5: The stepwise mobility adjustments for satisfying $|R|$=2 in Figure 2(b).

5. EXPERIMENTAL RESULTS

The proposed integrated memory binding and memory access scheduling algorithm MACCESS-lp was implemented in C++ and was executed on an Intel Pentium IV computer. We tested a set of benchmark programs in [11, 12, 13] to demonstrate the effectiveness of the proposed low-energy memory binding and scheduling algorithm. The programs we tested are shown in the first column of Table 1. DIFF is the differential equation solver and DIFF.2 is the design produced by unrolling DIFF design twice. KALMAN is the state vector computation part of the Kalman filter design, EWF is the fifth order elliptic filter design, and COMPLEX is the arithmetic part of complex number calculation. GAULEG and GAUJAC, taken from [12], are the main routines for Gaussian quadrature. BIQUAD and CHEBEV are taken from DSPstone benchmark suite in [13] and from [12], respectively.

• **The energy-efficiency of** MACCESS-lp **over the conventional approach**: Table 1 shows comparisons of the results of energy consumption in memories for the designs produced by the conventional technique that binds variables and schedules memory accesses in a greedy manner according to the order of first appearance in execution (indicated as called Greedy in Table 1) and our MACCESS-lp. $|R|$ represents the number of registers available to use for data load/store. The last five columns of the table show, for each design, the reductions of energy consumption used by MACCESS-lp varying the value of $|R|$ over that by Greedy, which is about

design	Greedy	MACCESS-lp					energy reduction (%)				
	$\|R\| < 5$	$\|R\| = \infty$	$\|R\| = 5$	$\|R\| = 4$	$\|R\| = 3$	$\|R\| = 2$	$\|R\| = \infty$	$\|R\| = 5$	$\|R\| = 4$	$\|R\| = 3$	$\|R\| = 2$
DIFF	86.34	72.64	72.64	72.64	72.64	72.64	15.87	15.87	15.87	15.87	15.87
DIFF.2	172.68	145.28	145.28	145.28	145.28	145.28	15.87	15.87	15.87	15.87	15.87
KALMAN	71.98	58.28	58.28	58.28	63.76	—	19.03	19.03	19.03	11.42	—
EWF	240.18	196.34	196.34	196.34	—	—	18.25	18.25	18.25	—	—
COMPLEX	89.00	75.30	75.30	75.30	75.30	75.30	15.39	15.39	15.39	15.39	15.39
BIQUAD	124.70	100.04	100.04	100.04	100.04	97.30	19.78	19.78	19.78	19.78	21.97
GAULEG	56.12	50.64	50.64	56.12	56.12	61.60	9.76	9.76	0.00	0.00	-9.76
GAUJAC	184.80	171.1	171.1	171.1	171.1	—	7.41	7.41	7.41	7.41	—
CHEBEV	40.10	31.88	31.88	31.88	31.88	40.10	20.50	20.50	20.50	20.50	0.00
average							**15.76**	**15.76**	**14.68**	**13.28**	**9.89**

$|R|$: The number of registers available to use for load/store.
—: indicate an infeasible (operation) schedule due to $|R|$.

Table 1: Comparisons of energy consumptions for the designs produced by a conventional memory binding and access scheduling and our MACCESS-lp**.**

15.76% less energy consumption when a sufficiently large number of registers is available, and 9.89% less energy even when only two registers are used. Note that the energy savings are consistent for all designs we tested.

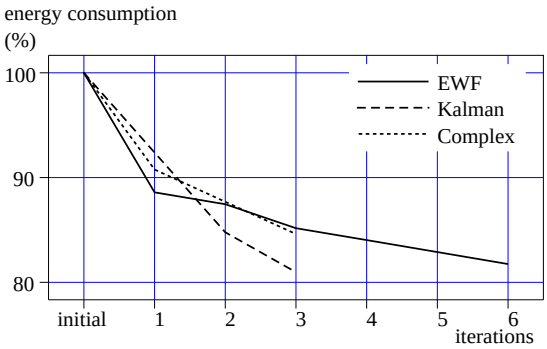


Figure 6: Curves showing the reductions of energy saving as the iterations MACCESS-lp**.**

• **The effectiveness of the procedure in** MACCESS-lp: The curves in Figure 6 show how well the proposed MACCESS-lp iteratively improves the designs. For EWF, MACCESS-lp terminates in six iterations whereas for Kalman and Complex, MACCESS-lp terminates in three iterations. On the other hand, Figure 7 shows how much energy saving MACCESS-lp can achieve from the energy consumption at the initial solution and from that by the execution of MACCESS-lp with the exploitation of memory access scheduling only. In summary, the energy saving by our integrated approach is significant, which is about 16%.

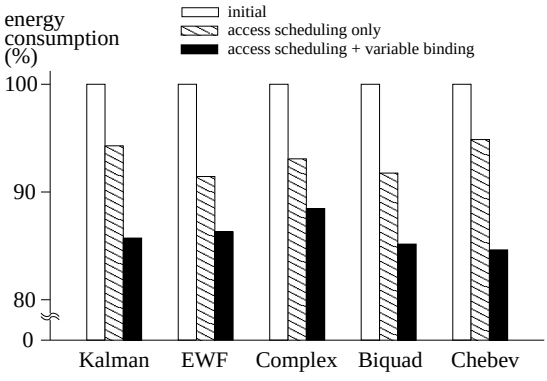


Figure 7: Graph showing the comparisons of energy consumptions used by an initial solution, access scheduling only, and simultaneous access scheduling and variable binding of MACCESS-lp**.**

6. CONCLUSIONS

In this paper, we presented a new algorithm for reducing the energy consumed by memory accesses via the exploitation of multiple memory operating modes in embedded system architecture with multiple memory banks. Specifically, we addressed the problem of maximally utilizing the operating modes of multiple memory banks by solving the three important tasks *simultaneously*: (1) *assigning variables to memory banks*, (2) *scheduling of memory access operations*, and (3) *determining operating modes of banks*. We confirmed from experiments that the proposed SP(shortest path)-based integrated technique was able to reduce the energy consumption by 15.76% on the average over the conventional greedy technique. It should be noted that even the proposed SP-based technique assumed two memory operating modes in the presentation, it can be easily extended to the case of any $m(> 2)$ operating modes while still maintaining the polynomial-time optimal computation of a-schedule.

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

7. REFERENCES

[1] A. Sudarsanam and S. Malik, "Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs," *ACM TODAES*, Vol. 5, 2000.

[2] J. Cho, Y. Paek, and D. Whalley, "Effi cient Register and Memory Assignment for Non-orthogonal Architecture Via Graph Coloring and MST Algorithms," *ACM Joint Conference LCTES-SCOPES*, 2002.

[3] V. De La Luz, M. Kandemir, and I. Kolcu, "Automatic Data Migration for Reducing Energy Consumption in Multi-Bank Memory Systems," *DAC*, 2002.

[4] L. Benini, A. Macii, and M. Poncino, "A Recursive Algorithm for Low-Power Memory Partitioning," *ISLPED*, 2000.

[5] L. Benini, L. Macchiarulo, A. Macii, and M. Poncino, "Layout-driven memory synthesis for embedded systems-on-chip," *IEEE TVLSI*, Vol. 10, 2002.

[6] Y-H. Lu, L. Benini, and G. De Micheli, "Low Power Task Scheduling for Multiple Devices," *International Workshop on Hardware/Software Codesign*, 2000.

[7] V. De La Luz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan and M. J. Irwin, "Scheduler-Based DRAM Energy Management," *DAC*, 2002.

[8] V. De La Luz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam and M. J. Irwin, "Hardware and Software Techniques for Controlling DRAM Power Modes," *IEEE TC*, Vol. 50, 2001.

[9] 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., May 1999.

[10] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[11] P. R. Panda and N. D. Dutt, "High-Level Synthesis Design Repository", *Proc. of ISSS (http://www.ics.uci.edu/ dutt)*, 1995.

[12] W. H. Press, *et al.* (Editors), *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1993.

[13] V. Zivojnovic, J. Velarde, and C. Schlager, "Dspstone: A DSP-oriented Benchmarking Methodology," *International Conference on Signal Processing Applications and Technology*, 1994.