Energy-Aware Variable Partitioning and Instruction Scheduling for Multibank Memory Architectures

ZHONG WANG and XIAOBO SHARON HU
University of Notre Dame

Many high-end DSP processors employ both multiple memory banks and heterogeneous register files to improve performance and power consumption. The complexity of such architectures presents a great challenge to compiler design. In this article, we present an approach for variable partitioning and instruction scheduling to maximally exploit the benefits provided by such architectures. Our approach is built on a novel graph model which strives to capture both performance and power demands. We propose an algorithm to iteratively find the variable partition such that the maximum energy saving is achieved while satisfying the given performance constraint. Experimental results demonstrate the effectiveness of our approach.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; C.3 [**Special-Purpose and Application-Based Systems**]—*Signal processing systems*; B.5.1 [**Register-Transfer-Level Implementation**]: Design—*Memory design*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Multiple memory banks, nonorthogonal architecture, instruction scheduling, operating mode, parallelism and serialism balance, runtime and energy saving tradeoff

1. INTRODUCTION

To meet the ever increasing demands for higher performance and lower power on embedded systems, domain-specific processors with sophisticated architectures are being designed and deployed to better match target applications. One such architecture, often referred to as a *nonorthogonal architecture* [Cho et al. 2002], is characterized by irregular data paths comprising of a heterogeneous register set and multiple memory banks. A number of embedded DSP processors, for example, Analog Device ADSP2100, Motorola DSP56000, and NEC uPd77016, are based on this architecture. Compared to a large, centralized homogeneous register file, a heterogeneous (in terms of instruction usage) register

Authors' addresses: Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556; email: {zwang1,shu}@cse.nd.edu.

set organized in a distributed fashion can reduce both access time and power, as well as simplify the control logic and chip layout design [Desoli 1998]. The use of multibank memory can potentially improve the exploitation of instruction level parallelism, which in turn may decrease memory access time and energy consumption compared to a single large memory [Benini et al. 2000].

Harvesting the benefits provided by the nonorthogonal architecture hinges on effective compiler support. Parallel operations afforded by multibank memory give rise to the problem of how to maximally utilize the instruction level parallelism. Similarly, heterogeneous register sets increase the difficulty in deciding which register set to use for a certain instruction. A good compiler should consider the heterogeneous register set assignment and instruction scheduling together, since the two are closely related [Zeithofer and Wess 2001]. It is not difficult to see that compilation techniques for general-purpose architectures are not adequate to handle the irregularity in the architecture. In this article, we focus on two critical steps in the compilation process, that is, partitioning variables (or data) among the memory banks, and scheduling memory access operations. The decisions made in these two steps can have a significant impact on the overall program code size, execution time, and energy consumption.

A number of articles (e.g., Sudarsanam and Malik [2000]; Saghir et al. [1996]; Lorenz et al. [2001]; Cho et al. [2002]; Leupers and Kotte [2001]; Zhuge et al. [2001]; Wuytack et al. [1969]) have investigated the use of multibank memory to achieve maximum instruction level parallelism (i.e., optimize performance). These approaches differed in either the models or the heuristics (which will be discussed in more detail in later sections). However, none of these works considered the combined effect of performance and power requirements.

It is well known that memory components in embedded systems, particularly those for data-intensive applications, are a major power consumer [Catthoor et al. 1998]. To help ease the energy demands by memory, advanced memory modules are designed to operate in different modes, for example, active, idle, and sleep [Rambus 1999; Micron 1999], which have different operating currents. The exploitation of different operating modes together with multiple memory banks further complicates the problem of variable partitioning and memory operation scheduling. On top of this, performance requirement often conflicts with energy savings. Previous works have studied the effects of multiple memory operating modes at the higher levels such as program basic blocks, system tasks, or processes. However, significant energy savings and performance improvements can be obtained by exploiting memory operating modes and multibank memory simultaneously at the instruction level (which we will illustrate with an example in Section 3, as well as in the experimental results section).

In this article, we present our approach to variable partitioning and memory access operation scheduling in the presence of multibank memory and multiple memory operating modes for maximizing energy savings without sacrificing performance. We reveal some observations to help categorize different cases. A novel memory access graph model, which simultaneously captures potential energy savings as well as potential performance improvements, is proposed to overcome the weakness of previous techniques. Based on this model, we have devised an iterative technique to find best energy savings while satisfying

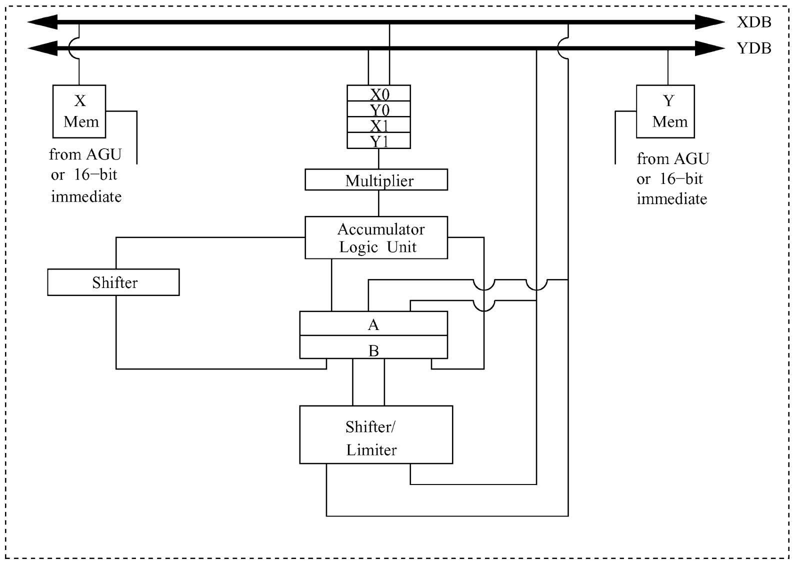


Fig. 1. The architecture of DSP56000.

the performance constraint. Experimental results show that our technique can achieve an average code size improvement of 14.15% over the unoptimized programs (for benchmarks in DSPStone [Zivoljnovic et al. 1994]) and 7.71% over the programs optimized by the original SPAM compiler (Princeton Spam Compiler Project; `web site: www.ee.princeton.edu/spam/`). Code size improvements translate directly to shorter execution times. Such improvements are quite significant compared with those obtained by existing approaches. In terms of energy savings from memory modules, our results on average outperform those from SPAM by around 47%. The experiments on the benchmark programs also showed that our algorithm runs much more efficiently than the original algorithm of SPAM.

The rest of the article is organized as follows. Section 2 presents the background material and reviews previous work. Sections 3 and 4 describe the energy savings strategy and graph model, respectively. The variable partitioning and instruction scheduling algorithm is discussed in Section 5. Section 6 provides experimental results and, finally, Section 7 concludes the article.

2. PROBLEM FORMULATION AND RELATED WORK

In this section, we briefly discuss essential features of the nonorthogonal architecture. We then formulate our problem and review related work.

2.1 Target Architecture and Problem Formulation

Our target architecture consists of multiple memory banks and a heterogeneous register set. Associated with each memory bank is an independent set of address bus, data bus, and address generation unit (AGU). Figure 1 shows an example of such an architecture, that of Motorola DSP56000. DSP56000 has three sets of register files ({X0, X1}, {Y0, Y1}, {A, B}) and two memory banks (X, Y). We used this architecture in our experiments. However, our algorithm can be easily extended to architectures with a homogeneous register set or more memory banks.

We consider memory modules used in the memory banks to have two operating modes, that is, the active mode and the low-current mode (standby or sleep) [Micron 1999]. The operating mode transition is controlled by the memory controller, whose states can be modified through a set of configuration registers [Delaluz et al. 2001]. The detailed discussion of controlling memory operating mode transition is beyond the scope of this article. In the active mode, a memory module performs normal read/write, while in the low-current mode, the memory module does not perform any memory operation and consumes much lower current than in the active mode. A memory module can switch between the two operating modes by incurring a certain time overhead. The memory module supply current during the mode transition is the same as in the active mode. For instance, for a Rambus RDRAM module [Rambus 1999], it takes a negligible amount of time to switch from the active mode into the standby (low-current) mode with the dynamic energy consumed in a cycle[1] being reduced from 3.57 nJ to only 0.83 nJ, but it takes two clock cycles to switch back to the active mode. For a Micron SyncBurst SRAM module [Micron 1999], it takes two cycles to switch the module from the active mode into the snooze (low-current) mode with the dynamic energy consumed in a cycle[2] changing from 5.61 nJ to only 0.17 nJ, and it takes another two cycles to switch back to the active mode. Clearly, in order to save energy by putting a memory module in the low-current mode, the consecutive idle time should be long enough to compensate for the transition time overhead. Furthermore, it is more beneficial to lump the idle times into a single long idle period than to disperse them. This presents some unique challenges to the problem we want to solve, which is formally defined as follows.

Definition 2.1. Given a program (in the form of an intermediate code) and a nonorthogonal architecture specification, generate an instruction schedule which maximizes the memory operation parallelism and energy saving.

It is not difficult to envision that increasing parallelism could have an adverse effect on energy savings. Our goal is to devise a methodology to trade off performance, that is, operation parallelism, and energy savings in the Pareto optimal solution set.

2.2 Related Work

To our best knowledge, no existing work has investigated the problem defined in Definition 2.1. However, a number of researchers have studied different aspects of this problem, for example, maximizing memory operation parallelism, exploiting the memory module operating modes, etc. We briefly review them below to help clarify our unique contributions.

[1]The dynamic energy in a cycle is obtained from the measured supply current values associated with memory modules documented in the data sheets (for a 3.3-V, 2.5-ns cycle time, 8-MB module).
[2]The dynamic energy in a cycle is calculated for a 3.3-V, 5.0-ns cycle time, 1-MB module.

2.2.1 *Related Work on Operation Parallelism.* Previous related work can be roughly divided into two main categories: those that use *compacted* intermediate code as the starting point (e.g., Sudarsanam and Malik [2000]; Saghir et al. [1996]; Lorenz et al. [2001]; Cho et al. [2002]), and those that start with *uncompacted* intermediate code (e.g., Leupers and Kotte [2001]; Zhuge et al. [2001]). *Compacted* intermediate code refers to the intermediate code that is compacted or scheduled by some heuristics such as list scheduling, to increase the instruction level parallelism without considering the data dependency. Since scheduling is done prior to exploring memory bank assignments, it is obvious that some memory-operation-pair combinations may be left out of consideration no matter which heuristic is used to compact the code. Thus, the approaches in the first category often fail to exploit many optimization opportunities. Techniques in the second category overcome this problem by using the uncompacted code to explore all possible pairs of memory operations as long as there are no dependencies between them. Therefore, we adopt the same philosophy as these techniques, that is, starting with the uncompacted code.

Most existing techniques to explore parallelism adopt some graph model for variable partitioning. A major distinction between those techniques lies in the graph model definition. Reviewing these graph models can help explain why these models are not adequate.

Given a program represented by a control data flow graph (CDFG), an undirected graph can be constructed to model the relationship among the variables in the program. The nodes in the graph represent all the local variables stored in memory. Partitioning the nodes in the graph into different groups then leads to partitioning the corresponding variables to different memory banks. The effectiveness of such an approach relies on modeling edge weights properly to capture all relevant information.

A straightforward way of assigning edge weights is to connect two nodes with an edge of weight 1 if the two corresponding variables do not have data dependencies and the memory operations involving the variables can potentially overlap [Leupers and Kotte 2001] (as accessing such two variables in parallel may decrease the schedule length). However, such potential parallelism may not be always realizable due to certain timing constraints on the associated memory operations. (Recall that the operations are to be scheduled later for uncompacted code.)

Zhuge et al. [2001] introduced the concept of possibility weight to capture the likelihood of parallelizing pairs of instructions. The model does improve on the simple graph model above, but it still has some deficiencies. For instance, to derive the edge weight between a pair of variables, they simply summed the possibility weights of all pairs of memory operations involving this variable pair in the entire procedure. Simply adding the possibility weights from different pairs of memory operations cannot correctly capture the scheduling freedom difference between the operation pairs. We will discuss these deficiencies in more detail in Section 4.

None of the existing graph models consider how to exploit serialism in instruction execution to trade off performance for energy savings. In this article, we propose to use two lists to describe the edge weight in the graph model. By introducing one more dimension to the graph edge weight, we not only capture the serialism information among operations, but also overcome the deficiencies of previous models.

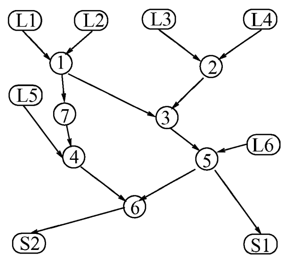
2.2.2 *Related Work on Energy Savings.* A number of research results have been published regarding saving energy through exploiting operating mode changes. The key idea is to distribute idle times judiciously through good scheduling. This can be achieved at various abstraction levels or design stages. For example, an on-line, low-power, task scheduling algorithm for multiple devices was presented in Lu et al. [2000]. An operating system- OS- based solution was proposed in Delaluz et al. [2002] where the OS scheduler manages power mode transitions by keeping track of module accesses for each process in the system. Several articles have been published to exploit the benefit of memory operating mode control. In Delaluz et al. [2001], a compiler-directed scheme was presented to reschedule the basic blocks such that longer consecutive memory idle times can be obtained. The techniques in Benini et al. [2000] and Luz et al. [2002] considered data organization in multibank memory such that data accesses can be concentrated in a small number of banks while other banks can be left in the low-current mode. No instruction scheduling was considered in these works.

Our work focuses on the instruction level. By integrating energy consideration into the instruction scheduling stage, we can achieve additional energy savings without sacrificing performance. Note that our work complements the above mentioned energy savings techniques since it can be applied together with these other techniques.

2.2.3 *Other Related Work.* Some research related to multiple memory banks concerns with memory partitioning [Benini et al. 2000; Wuytack et al. 1999]. Given the memory access pattern of a class of programs, memory partitioning finds the best memory bank configuration, for example, the number of memory banks, the size of each bank, the number of ports for each bank, etc., from the viewpoint of instruction level parallelism or energy savings. It is a different problem from the one considered in this article in that the memory configuration is given in our architecture model, and we concentrate our effort on variable partitioning and instruction scheduling among memory banks. It is worth noting that Wuytack et al. [1999] deployed the model of conflict graph and conflict probability, which derives the graph information from uncompacted intermediate code. Though it addressed a different problem, it also showed that working on uncompacted code reveals more optimization opportunities.

3. IDLE TIME EXPLORATION

As mentioned earlier, memory operating mode transition does not come for free. Extra clock cycles are needed to change between the active and low-current modes. Therefore, to exploit the low-current mode, longer consecutive idle times are more desirable for a memory bank. However, variable partitioning and instruction scheduling with only performance considerations may not lead to the best schedule in term of idle time distribution. For example, for the data flow



(a)

Instr			
0		L1	L2
1	Op 1	L3	L4
2	Op 2		
3	Op 3	L6	
4	Op 5		
5	Op 7	S1	L5
6	Op 4		
7	Op 6		
8			S2

(b)

Instr			
0		L1	L2
1	Op 1	L3	L4
2	Op 2	L6	L5
3	Op 3		
4	Op 5		
5	Op 7		
6	Op 4		
7	Op 6		
8		S1	S2

(c)

Instr			
0		L1	L2
1	Op 1	L3	L4
2	Op 2		
3	Op 3	L6	
4	Op 5	L5	
5	Op 7	S1	
6	Op 4		
7	Op 6		
8		S2	

(d)

Fig. 2. (a) An example DFG, where L (respectively, S), followed by an integer i represents a *LOAD* (respectively, *STORE*) operation on variable i. Other nodes are nonmemory operations. Edges denote the precedence constraint between operations. (b) Schedule with only performance consideration. (c) Schedule when mode transition time is two cycles. (d) Schedule when mode transition time is four cycles.

graph (DFG) shown in Figure 2(a), a schedule with only performance consideration is shown in Figure 2(b), while better schedules with respect to both performance and energy are shown in Figures 2(c) and 2(d). In Figure 2(b), the memory modules cannot be switched to the low-current mode because all idle times are too short. In the latter two schedules, the idle slots are put together such that one or more memory modules can change to the low-current mode during the idle periods. In Figure 2(c), both memory banks can be put into low-current mode during the control steps $3 \rightarrow 5$ under the assumption that Rambus RDRAM is used, while in Figure 2(d), the second memory bank can be put into low-current mode during the control steps $4 \rightarrow 6$ under the assumption that a Micron SRAM module [Micron 1999] is adopted. Thus, we gain energy savings without affecting the schedule performance.

Memory operation scheduling for energy savings is tightly related to that for maximum parallelism, but their different goals can lead to totally different schedules. For example, one could easily sacrifice all the parallelism by putting all variables in one memory bank, which gives the longest idle times for other memory banks. Therefore, a tradeoff exists between energy savings and performance. We consider the problem of maximizing the energy savings without degrading the performance (i.e., program execution time).

In the following, we examine an ideal scenario in which no register constraint exists. In other words, all the variables can be loaded at the earliest time and stored at the latest time. The importance of this case will become clear in Section 5, where we will show that an operation schedule can be regarded as the ideal scenario after the mobility is calculated with the register constraint in mind. Given a control data flow graph (CDFG) representing the behavior of a program segment, assume that the desired schedule length is t, the number of memory operations in the ith memory bank is n_i and the overhead for memory module mode transition is m clock cycles. For a given t, there exist three cases depending on the relationship of t, n_i and m.

Case 1. $\min(t - n_i) > m, \forall i$.

Maximal energy savings can be readily achieved by Lemma 3.1. The correctness of Lemma 3.1 is easy to prove and is omitted.

LEMMA 3.1. *If* $\min(t - n_i) > m, \forall i$, *by simply pushing the LOAD (respectively, STORE) operations to the beginning (respectively, end) of the schedule, the maximal energy saving is achieved.*

The schedule in Figure 2(b) belongs to this case when the operating mode transition time is two cycles. The schedule with optimal energy savings can be readily obtained by Lemma 3.1 and is shown in Figure 2(c).

Case 2. $\min(t - n_i) \leq m, \exists i$ and $t \geq \frac{n+m}{N}$.

In the above conditions, N denotes the number of memory banks, and n is the total number of memory operations, that is, $n = \sum_{i=1}^{N} n_i$. These two conditions mean that consecutive idle times, which are long enough to change the memory module to low-current mode, can be formed in some but not all of the memory banks. To improve energy savings, one might consider moving the memory operations between banks to serialize more operations in one or more banks while leaving other banks with longer idle times. The goal is then to maximize "serialism" without degrading the performance. The example in Figure 2(d) illustrates such a thought for the SRAM memory module. The desire to increase serialism in this case complicates the variable partitioning problem.

Case 3. $t < \frac{n+m}{N}$.

N and n have the same meaning as in Case 2. No further optimization can be obtained in this case. So long as the schedule length is maintained, not enough idle time can be formed in any memory module.

For a given problem, deciding the schedule length is not an easy task. Even if we have a schedule, Case 2 still presents quite a challenge. In the following, we present our approach to tackling the problem.

4. GRAPH MODELING APPROACH

As mentioned in Section 2.2.1, the edge weight assignments introduced in the previous works all fail to capture some important information, which may lead to suboptimal solutions. In this section, we describe our edge weight assignment

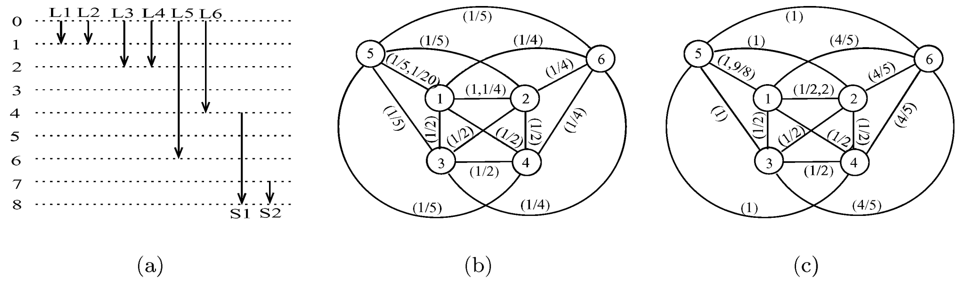


Fig. 3. (a)Memory operation mobility graph for DFG in Figure 2(a). (b) Memory access graph with the first dimensional weight; each node number corresponds to the associated variable number. (c) Memory access graph with the second dimensional weight.

approach by examining the requirement of a desirable graph model and analyzing the deficiencies of previous graph models [Zhuge et al. 2001].

The construction of the graph model is based on the CDFG representation of an application. The information about memory-operation scheduling freedom can be derived from the CDFG and fed into a later stage to assign the graph edge weights.

From the CDFG representation of a program, one can readily derive both the as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) schedules, considering the constraints of computation units. Let the control steps of a memory operation, a, be $t_s(a)$ and $t_l(a)$ according to ASAP and ALAP, respectively. The mobility, that is, the scheduling freedom of a, defined as $[t_s, t_l]$, represents the time interval in which a can be scheduled without introducing additional delay. Only when the mobilities of two memory operations have some overlap may parallelizing the two corresponding variables be beneficial (in terms of improving performance). Clearly, the larger the overlap between two mobilities, the higher the potential of the two variables being able to be parallelized. If the mobilities of two operations are both small and their overlap is relatively large, parallelizing the corresponding variables is more likely to improve the schedule length. In other words, if such variables are put in the same bank, accessing the two variables is forced to be sequentialized, which is very likely to increase the overall schedule length. Zhuge et al. 2001 assigned a possibility weight defined below to an edge to model this property.

Definition 4.1. Given two memory operations, a and b, let their mobilities be $[t_s(a), t_l(a)]$ and $[t_s(b), t_l(b)]$, and the maximum overlap between these two mobilities be the interval $[t_1, t_2]$. The possibility weight assigned to the edge between the two variables accessed in operations a and b is $\frac{t_2-t_1+1}{(t_l(a)-t_s(a)+1)(t_l(b)-t_s(b)+1)}$.

Figure 3(b) shows an example of this possibility weight assignment for the memory operations given in Figure 3(a) which captures the memory operation mobilities for the DFG in Figure 2(a). In Figure 3(a), there are six variables and eight memory operations. The line beneath or above each operation represents its mobility. For example, $L3$ has a mobility of [0, 1]. In Figure 3(b), more than one possibility weight may be associated with an edge. These come from different pairs of memory operations. For instance, between variables v_1

and v_5, 1/5 comes from $(L1, L5)$ pair, while 1/20 from $(L5, S1)$ pair. In Zhung et al. [2001], such numbers were simply added together. Moreover, if the same operation pattern (e.g., overlapping of $L1$ and $L5$) occurs in another *mobility range*,[3] the possibility weight is again added to the edge weight between v_1 and v_5.

One deficiency of the possibility weight model in Zhung et al. [2001] is associated with simple summation of the possibility weights mentioned earlier. Consider a simple example of an edge possibility weight of 1. This may come from two operations such as $L1$ and $L2$ in Figure 3(a). It may also come from two occurrences of the operation pair such as $L3$ and $L4$ in Figure 3(a). Though both edges have a weight of 1, it is not difficult to see that the variables in the first case should be given a higher priority to be parallelized since the schedule length will definitely be increased if the two variables are put in the same memory bank (while the variables in the second case have an additional slack cycle). To overcome this problem, we advocate maintaining as a list the possibility weights from different operations involving the same variable pair instead of adding them together. (We will discuss how to manipulate this list later.) For example, a list, which contains two elements, that is, 1/5 and 1/20, is maintained to reflect the parallelism weights for edge between variables $v1$ and $v5$ in Figure 3(b).

Another problem with the possibility weight model is that it does not distinguish mobility overlaps within a single mobility range from those in different mobility ranges. Consider the following example. Given three memory operations, a, b, c, in one mobility range, each has the same mobility $[0, 1]$. The corresponding graph model contains an edge with weight 1/2 between the variables in a and b and between those in a and c. Assume in the same procedure, memory operations a', b' in a different mobility range, have the mobility $[3, 4]$, and memory operations a', c' have the mobility $[7, 8]$ in yet another mobility range. Then, the associated graph has an edge with weight 1/2 between the variables in a' and b' and between those in a' and c'. Obviously, variables in a, b, c should be given a higher priority to be parallelized than variables in a', b', c' since putting the former in one memory bank will definitely introduce an additional delay (while putting the latter in the same bank does not necessarily introduced additional delay since operations on variables in a', b' are in a different mobility range from those on variables in a', c'). However, the model in Zhuge et al. [2001] treats the two groups indiscriminately.

Besides the above shortcomings, the possibility weight model has no consideration about energy savings because the work in Zhuge et al. [2001] focused only on performance. From the point of view of energy savings, one would prefer to serialize memory operations as much as possible so as to leave more “long” idle intervals for the low-current mode (see the discussion of Case 2 in Section 3). Clearly, this preference toward serialism may run against the requirement of improving performance.

[3]A mobility range is a period of consecutive scheduling steps which may cover several variables' mobility. In this article, whenever we talk about two different mobility ranges, we refer to independent nonoverlapping mobility ranges.

To capture the tradeoff between the desire of parallelism and that of serialism, we propose to use two lists of weights. The first one is the one discussed above, that is, the list of possibility weights, which are referred to as *parallelism weights*. The second one is a new one and the weights are referred to as *serialism weights*. The goal of serialism weights is to model the possibility of serializing a pair of operations without sacrificing performance. To derive the serialism weight, observe that, given a certain mobility range, the more operations in the range, the more difficult it is to serialize the operations without increasing the total delay. Taking the example above, serializing three operations a, b, c increases the schedule length, while serializing a' and b' (or a' and c') has no negative effect. Based on this observation, we formally define the serialism weight as follows.

Definition 4.2. Assume the mobilities of two memory operations, a and b, are $[t_s(a), t_l(a)]$ and $[t_s(b), t_l(b)]$, respectively, their union is $[t_1, t_2]$, and the number of operations whose mobilities are contained in $[t_1, t_2]$ is n. The serialism weight for the edge between the variables accessed in a and b is $\frac{t_2-t_1+1}{n}$.

An example of the serialism weight is shown in Figure 3(c). Note that, similar to the parallelism weights, more than one serialism weight may be associated with an edge due to the multiple occurrences of the memory operations involving the corresponding variable pair. Taking the serialism weight between variables v_1 and v_5 as an example, 1 comes from the (L_1, L_5) pair (six operations exist in mobility range $[0, 5]$ which is the shortest range to cover both L_1 and L_5's mobility), while 9/8 comes from the (L_5, S_1) pair (eight operations exist in mobility range $[0, 8]$). A list is also maintained for each edge to record these serialism weights.

We now formally define the *memory access graph* (MAG) used in our approach.

Definition 4.3. A memory access graph (MAG), $G = (V, E, \mathbf{F})$, is a multiweighted undirected graph, where V is the set of nodes representing the variables in the given code, $E \in V \times V$ is the set of edges, $\mathbf{F} = (\vec{w_p}, \vec{w_s})$ is a function from E to R^{2m} representing the weight lists between the corresponding two nodes, and $\vec{w_p}$ and $\vec{w_s}$ are the parallelism and serialism weight lists, respectively.

Though we are able to capture the requirements of performance and energy savings through introducing both parallelism and serialism weights, we need to be able to use them effectively in partitioning the variables. The problem of variable partitioning for k memory banks is equivalent to the maximum k-cut problem, which is NP-complete [Ausiello et al. 1999]. A number of excellent heuristics exist for solving the maximum k-cut problem [Ausiello et al. 1999]. To use such heuristics, we need to reduce the two lists of weights associated with an edge to a single weight value. To reflect the tradeoffs between performance and energy savings, we use a weighted sum formula to compute the average weight of an edge. Specifically, the average weight of an edge $e(i, j)$ is defined as

$$w(i,j) = \sum_{h=1}^{m(i,j)} \lambda_p w_p(h,i,j) - \lambda_s w_s(h,i,j), \tag{1}$$

where λ_p, λ_s are two coefficients representing the tradeoffs between parallelism and serialism, $w_p(h,i,j)$ (respectively, $w_s(h,i,j)$) is the parallelism (respectively, serialism) weight associated with the hth pair of operations involving variables i and j, and $m(i,j)$ is the total number of such pairs. The rationale behind the subtraction used in Equation (1) is that $w_p(h,i,j)$ and $w_s(h,i,j)$ can be viewed as measures of two opposite forces, parallelism and serialism. Different coefficient values, λ_p and λ_s, reflect the preference between the two forces, and hence help trade off performance with energy savings.

As we can see from the definition, the parallelism weight can be regarded as the local measurement which captures the scheduling information between two memory operations, while the serialism weight can be regarded as the global measurement which reflects the scheduling freedom for memory operations belonging to the same mobility range. The integration of these two factors through Equation (1) provides a complete measure of benefit which can be obtained from a certain variable partition.

Each member of $\vec{w_p}$ is always smaller than 1, while that of $\vec{w_s}$ may be larger than 1. In order to ensure these two values are in the same range, each member $w_s(i,j)$ is normalized with the formula $w_s(i,j) = \frac{w_s(i,j)-w_{\min}}{w_{\max}-w_{\min}}$, where $w_{\min}$(respectively, $w_{\max}$) is the minimum (respectively, maximum) value of all $w_s(i,j)$ values in the corresponding mobility range to which this $w_s(i,j)$ belong. Keeping a linked list to record all weights enabled us to do this normalization on the basis of mobility range.

It is important to point out that the average weight defined in Equation (1) indeed can overcome the shortcomings mentioned earlier in the model introduced in Zhuge et al. [2001]. Consider the shortcoming associated with simply adding possibility weights for the two discussed scenarios that the possibility weight is not able to distinguish, under our average weight model, that the edge weight in the first case is $(\lambda_p - \frac{1}{2}\lambda_s)$, while the edge weight in the second case is $(\frac{1}{2}(\lambda_p - \lambda_s) + \frac{1}{2}(\lambda_p - \lambda_s))$. (We assume that no other variables have operations overlap with the mobilities under consideration.) Given the same λ_p and λ_s values, the former is always greater than the latter, which correctly reflects the fact that it is more beneficial to put the two variables in the first case into separate memory banks as they have a more stringent timing requirement. For the shortcoming due to mobility range differentiation, according to our model, the average weight on the edge between a and b and that between a and c is $(\frac{1}{2}\lambda_p - \frac{2}{3}\lambda_s)$, while the average weight on the edge between a' and b' and that between a' and c' is $(\frac{1}{2}\lambda_p - \lambda_s)$ (assuming no other operations overlap these mobilities). Again, the edges between a, b, and c have a larger weight for a given pair of coefficients, and hence the variables associated with these operations are favored for putting into separate banks (which is exactly what one would like to see).

There are still several unanswered questions. For example, how should one select the coefficients in the average weight definition for a given application?

```
**Algorithm 1**
Input: Intermediate Code, Register Constraints
Output: Code optimized for energy and performance
  1. Derive the CDFG from the intermediate code. Calculate the mobility for each
  operation.
  2. Construct the memory access graph (MAG) //refer to Section 4
  3. $\lambda_p = 1$, $\lambda_s = 0$, $\lambda'_s = 0$,[4] calculate the average weight for each edge and schedule the
  program.
  4. Set the minimum schedule length $L_{\min}$ as the schedule length of the current
  schedule, $T_{\max} = 0$.
  WHILE ( ) do
    5. Find the maximum cut. Allocate variables to memory banks according to the cut
    result.
    6. Schedule the program according to the above allocation result while maximizing
    the consecutive idle time. Set $L_{schedule}$ and $T_{schedule}$ //refer to Section 3
    7. if $L_{schedule} - L_{\min} \leq \phi$ and $T_{schedule} - T_{\max} \leq \eta$ then
      $N_{stable}++$;
      Record the corresponding variable partition and schedule;.
    end if
    8. if $L_{schedule} - L_{\min} \leq \phi$ and $T_{schedule} - T_{\max} > \eta$ then
      $L_{\min} = \min(L_{\min}, L_{schedule})$,
      $T_{\max} = \max(T_{schedule}, T_{\max})$,
      $\lambda_s = \frac{\lambda_p+\lambda_s}{2}$,
      $N_{stable} = 0$
    end if
    9. if $L_{schedule} - L_{\min} > \phi$ then
        $\lambda_s = \frac{\lambda_s+\lambda'_s}{2}$,$\lambda'_s = \lambda_s$, $N_{stable} = 0$
    end if
    10. if $N_{stable} \geq \sigma$ then
        break;
    end if
    11. Recalculate the average weight of MAG.
  ENDWHILE
  12. Output the corresponding variable partition and schedule
```

Also, how does one handle the register constraints? We shall discuss these issues in the next section, where we present our complete algorithm.

5. ALGORITHM

Our variable partitioning and instruction scheduling algorithm is intended to be used in the back end of a compiler to optimize the intermediate code. The algorithm operates on the CDFG representation of a given piece of intermediate code. As the first step, it calculates the mobility for each operation with the register constraint in mind. Then the MAG is constructed based on the mobility information. With this MAG, the steps of *average weight calculation, maximum cutting, variable partitioning, and instruction scheduling* are iterated for a number of times to find the best values of λ_p, λ_s. The algorithm framework is shown in Algorithm 1.

In Algorithm 1, $T_{schedule}$ represents the number of consecutive idle cycles for the *current schedule*, while $T_{\max}$ represents the maximal value of all $T_{schedule}$.

[4] λ'_s is used to remember the value of λ_s in the previous loop iteration.

$L_{schedule}$ and $L_{\min}$ represent the current and minimum schedule lengths, respectively. ϕ is a user-specified parameter to indicate the latency constraint and defined as the allowed difference between the final and minimum schedule lengths. η is a user-defined threshold to measure whether $T_{schedule}$ has a significant change. The algorithm will finish after $T_{schedule}$ has not shown significant changes for σ number of loops.

In Line 1 of Algorithm 1, we use the technique in Zeithofer and Wess [2001] to deal with the heterogeneous register set and register constraint. By dividing the register mapping into two stages, register allocation (before scheduling) and register assignment (after scheduling), the algorithm in Zeithofer and Wess [2001] obtains the benefit, but avoids the difficulty of considering register mapping and scheduling together. A heterogeneous register set is dealt with by transforming physical registers into virtual registers such that all virtual registers can be regarded as homogeneous. The concept of virtual registers provides a powerful methodology to check if a feasible register assignment exists for a specific schedule without the necessity of generating one. This allows the flexibility of considering the register constraint during scheduling, by simply checking if enough virtual register resources are available in each schedule step. Lots of effort can be saved by determining the detailed register assignment for the final schedule instead of every possible intermediate schedule. The approach in Zeithofer and Wess [2001] is similar to the register class concept in Jung and Paek [2001] , but with the advantage that the number of available virtual registers can be derived to constrain the mobility of each variable.

In Line 5 in Algorithm 1, the well-known maximum spanning tree (MST) algorithm [Prim 1957] is used as the maximum-cut heuristic. Then variables are allocated to the memory banks under the rule that two variables having a MAG edge belonging to the cut must be in different banks. Note that any heuristic maximum-cut algorithm can be used at this step. The MST algorithm is preferred because of its popularity and easy implementation.

The WHILE loop of the algorithm is used to find a point where the maximal energy savings is achieved for the desired performance. Because of the opposite forces of parallelism and serialism, more parallelism (larger λ_p and smaller λ_s) may bring better performance and less energy savings, while more serialism (smaller λ_p and larger λ_s) may bring more energy savings, but a possible deteriorated performance. Therefore, we introduce a process analogous to the binary search into the algorithm, trying to reach the best tradeoff point, that is, a set of λ_p and λ_s values to achieve maximal energy savings for a given desired performance. In Step 8, if the performance is still in the desirable scope and more energy savings can be achieved through the last change of λ_s, we then push λ_s further toward the direction of serialism in the hope of getting more energy savings without degrading the performance. On the other hand, if the performance degrades too much, we move back λ_s toward parallelism in Step 9 in the hope of recovering the performance but still maintaining the gained energy savings. Finally, when the alteration of λ_s becomes too small to bring any meaningful change on either performance or energy savings, the algorithm exits from the WHILE loop.

```
MOVE main.(y) R0
MOVE main.(ph), R1
MOVE main.(px), R2
MOVE main.(px2), R3
DO #15 TMP14
MOVE X:(R1)-, X0
MOVE X:(R2), X1
MOVE X:(R0), A
MAC X0 X1 A
MOVE A X:(R0)
MOVE X:(R3)-, X0
MOVE X0, X:(R2)-
TMP14:
```

(a)

```
MOVE main.y R4
MOVE main.(ph), R0
MOVE main.(px), R1
MOVE main.(px2), R2
MOVE Y:(R4) A
DO #15 TMP14
MOVE X:(R0)-, X0
move X:(R1), X1
MAC X0 X1 A              X(R2)-, X0
MOVE X0 X:(R1)-
TMP14:
MOVE A Y:(R4)
```

(b)

Fig. 4. An example to illustrate the result of applying Algorithm 1. (a) The original assembly code given by SPAM compiler. (b) The assembly code after applying Algorithm 1.

The complexity of the algorithm depends on two factors, the schedule length (L) and the number of variables (N). In the WHILE loop, step 6 takes $O(L^2)$ time, while the MST algorithm can be done in $O(N^2 \log N)$ since there are at most $O(N^2)$ edges in an MAG graph. Therefore, the algorithm complexity is $O(w(L^2 + N^2 \log N))$, where w is the number of iterations of the WHILE loop body. It is shown in the experimental section that w is normally quite small for reaching the final result.

The algorithm can be easily extended to other systems with different architectural parameters. For instance, if a system has a different register file set, the technique in Zeithofer and Wess [2001] can still be used. The only difference will be the number of available virtual resources. By replacing the MST algorithm with some polynomial maximum k-cut heuristic [Goldschmidt and Hochbaum 1998], this algorithm framework can be extended to the system with more memory banks.

Figure 4 shows an example assembly code which is the loop segment of a FIR filter in a DSPStone benchmark suite [Zivoljnovic et al. 1994]. Figure 4(a) is the assembly code obtained from SPAM. Figure 4(b) is the result after applying Algorithm 1, shown in the instruction format of *opcode*, *operands*, and two possible parallel *move* fields. There are 10 nodes and 34 edges in the MAG graph for the FIR filter. It takes four iterations to reach the final result. The reader is referred to Wang and Hu [2004] for the complete code example. In this example, variable y is put into Y memory bank due to the global variable partition consideration. The loop body length is reduced from seven to four clock cycles. The improvement is achieved by increasing the program parallelism (see instruction 9 in Figure 4(b)) and memory operation scheduling (moving the operations of loading (respectively, storing) variable y to the beginning (respectively, end) of its mobility according to Lemma 3.1, thus out of the loop body). The final code is a tradeoff between energy savings and schedule length. If the performance is the only emphasis, the loop body can be further reduced to three clock cycles by moving data array px to the Y memory bank.

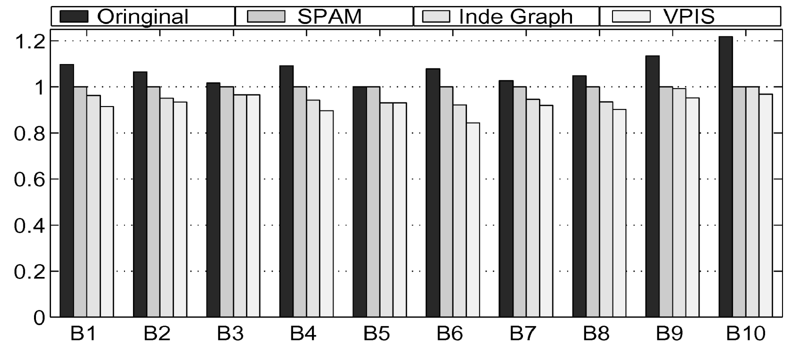


Fig. 5. Assembly code size results. The results are normalized with respect to the SPAM results.

6. EXPERIMENTAL RESULTS

We have implemented our algorithm in the SPAM compiler environment to replace the simulated annealing algorithm [Sudarsanam and Malik 2000] originally used by the Princeton project [(see Website www.ee.princeton.edu/spam/)]. The benchmarks used are from the DSPstone benchmark suite [Zivoljnovic et al. 1994], which contains C source code for various DSP kernels: *Least Mean Square (B1), FIR (B2), N Real Update (B3), IIR Biquad (B4), Convolution (B5), N Complex Update (B6), 2-Dimensional FIR (B7), Matrix Multiplication (B8), 1st Adapted Predictor (B9), and Tone Detector (B10) routine in ADPCM*. The intermediate code is generated by SUIF front end followed by SPAM code generation program, then fed as input to our algorithm to obtain the optimized assembly code.

The assembly code size results are shown in Figure 5. The data include the original code size (*Original*), code size generated by the constraint-graph method (*SPAM*), code size generated by Zhuge et al. [2001] (*Inde Graph*[5]), and code size generated by our algorithm (VPIS). Figure 5 reveals that the methods of independence graph and our algorithm can both perform better than SPAM. This improvement can be attributed to exploiting more potential memory operations parallelism. Accredited to our comprehensive graph model and judicious selection of weight coefficients, our algorithm demonstrates a superior performance to the method of independence graph, as demonstrated in Figure 5. The execution time of the assembly code is correlated to the code size [Leupers and Kotte 2001], since the assembly code can be directly mapped to the schedule for the basic block. Moreover, due to the existence of loops in the DSP benchmark, we are able to observe even larger improvements when comparing the execution time of assembly code. The results are shown in Figure 6.

We compare the energy savings results of our algorithms those with SPAM. Results from *Inde Graph* method are not included in this comparison, since it does not have the energy savings consideration. In fact, it can be regarded as a

[5]Their article used a greedy heuristic algorithm, similar to the MST algorithm, to partition the variables. Their results reported in this section were obtained from the MST algorithm for the sake of fairly comparing graph models.

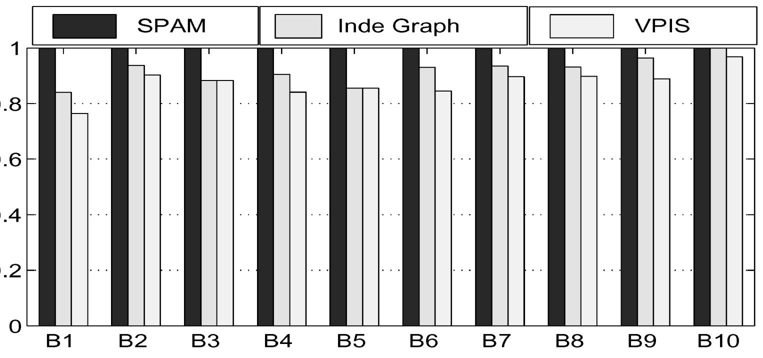


Fig. 6. Execution time of assembly code, The results are normalized with respect to the SPAM results.

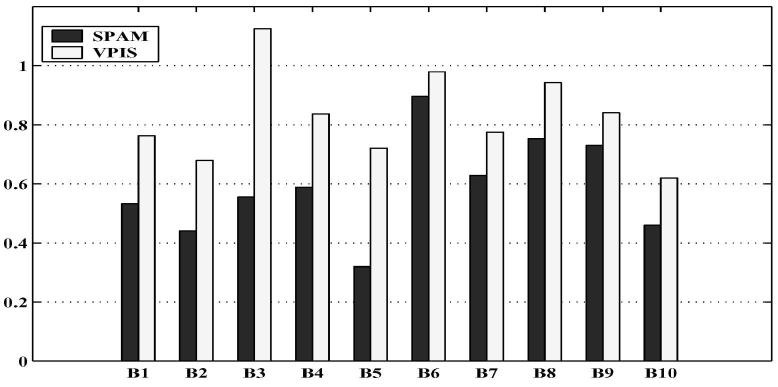


Fig. 7. Percentage of low-current cycles over the total execution clock cycle.

special case of our algorithm with the restriction of $\lambda_p = 1, \lambda_s = 0$. As a simple comparison, we examine the generated assembly code. By counting the number of consecutive idle cycles (which must be larger than the operating mode transition time) for the two schemes in each basic block, the absolute number of idle cycles in which the memory module can be put into low-current mode is obtained by summing all such numbers in the entire procedure. The ratio of these idle cycles to the overall code size can be calculated. The average improvement of this ratio is 19.84%. This ratio comparison can give us the initial estimation of at least how much improvement can be achieved from the algorithm. With the multiple execution times of loop bodies, a larger improvement should be expected, which is demonstrated in the following comparison.

As a more elaborate comparison, we have simulated the execution of the generated assembly code with Sim56000 (Motorola's DSP56000 simulator). From the run profile, all the usable idle times are added together to get the total idle cycles during which the memory module can be put into low-current mode. By dividing this total of idle cycles by the benchmark's total of execution clock cycles, the ratios of memory energy savings are derived for all benchmarks. Note the upper bound for this ratio equals the number of memory banks. The results are shown in Figure 7.

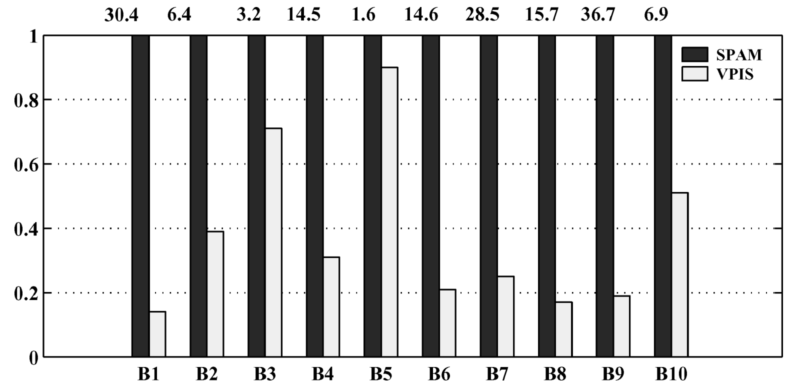


Fig. 8. Algorithm execution time.

Our algorithm achieves larger improvements for data-intensive (e.g., computation-intensive loop body) than control-intensive (e.g., procedure call, procedure initialization) application code, since data-intensive code involves more ALU operations which operate only on the register file. Furthermore, data-intensive code is usually executed many times, as in the computation loops in DSP applications. Based on these facts, we can see a larger energy savings in Figure 7 than the initial estimation given above. The average improvement of our approach over SPAM is 47.55%.

One concern may be raised about the algorithm execution time because of the loop in the algorithm to find the best tradeoff point. Due to the fact that variable partitioning is not very sensitive to the change of average coefficients λ_p and λ_s (a small change in these two coefficients does not change the variable partition), the algorithm generally can find the best tradeoff point in at most 20 loops. We ran the program in SUN Ultra Sparc2, and the algorithm execution time comparison is shown in Figure 8. In the figure, we normalized the algorithm execution time by the simulated annealing (SA) algorithm (adopted by SPAM) execution time, which is given in the unit of seconds on the top of each benchmark.

Figure 8 shows that our algorithm takes much less time than the SA-based algorithm. Moreover with more complicated programs, the constraint graphs in SPAM become larger and each step in the annealing process takes a longer time. The SA algorithm execution time increases significantly with the increase in the constraint graph size, while our algorithm, by contrast, does not have to deal with the large graph for many times (at most 20 loops for our experiments). Therefore, the execution time improvement becomes more obvious for larger benchmarks.

7. CONCLUSION

A variable partitioning and instruction scheduling algorithm is proposed to exploit the architecture with multiple memory banks and heterogeneous register set. The algorithm takes into account both instruction level parallelism and reducing system energy. A novel graph model is presented to capture both parallelism and serialism scheduling information. With such a model, the maximum instruction level parallelism can be exploited to improve the schedule performance. The idle intervals of the memory module are maximized under the constraint of the schedule performance, such that the system energy is reduced by changing memory modules to low-current mode for longer time. Experimental results demonstrate that our algorithm outperforms the previous techniques.

As future work, the novel graph model presented in this article can be extended to other architectures besides multiple memory bank architecture, such as clustered architecture and distributed systems. One common characteristic for all these systems lies in the tradeoff between parallelism and serialism because of energy savings considerations, resource constraints, etc. For example, in a clustered architecture, it is important to balance the workload to all clusters to increase the performance. On the other hand, to reduce energy consumption, it is desirable to reduce the bus communication between clusters and put some clusters in low-power mode. Therefore, a graph model which can capture all such information is essential to a good scheduler. How to extend our proposed graph model to cope with other architectures is worth more research effort.

REFERENCES

AUSIELLO, G., CRESCENZI, P., GAMBOSI, G., KANN, V., MARCHETTI-SPACCAMELA, A., AND PROTASI, M. 1999. *Complexity and Approximation*. Springer Verlag, Berline, Germany.

BENINI, L., MACII, A., AND PONCINO, M. 2000. A recursive algorithm for low-power memory partitioning. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 78–83.

CATTHOOR, F., WUYTACK, S., GREEF, E., BALASA, F., NACHTERGAELE, L., AND VANDECAPPELLE, A. 1998. *Custom Memory Management Methodology—Exploration of Memory Organization for Embedded Multimedia System*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

CHO, J., PAEK, Y., AND WHALLEY, D. 2002. Efficient register and memory assignment for nonorthogonal architectures via graph coloring and MST algorithms. In *Proceedings of the ACM Joint Conference LCTES-SCOPES* (Berlin, Germany). 130–138.

DELALUZ, V., M. KANDEMIR, N. V., SIVASUBRAMANIAM, A., AND IRWIN, M. J. 2001. Hardware and software techniques for controlling DRAM power modes. *IEEE Trans. Comput. 50*, 11 (Nov.), 1154–1173.

DELALUZ, V., SIVASUBRAMANIAM, A., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. J. 2002. Scheduling techniques for embedded systems: Scheduler-based DRAM energy management. In *Proceedings of the 39th Conference on Design Automation*. 697–702.

DESOLI, G. 1998. Instruction assignment for clustered VLIW DSP compilers: A new approach. Tech. Rep. HPL-98-13. Hewlett-Packard Company, Palo alto, CA.

GOLDSCHMIDT, O. AND HOCHBAUM, D. S. 1998. Polynomial algorithm for the k-cut problem. In *Proceedings of the 29th Annual Symposium on the Foundations of Computer Science*. 444–451.

JUNG, S. AND PAEK, Y. 2001. The very portable optimizer for digital signal processors. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 84–92.

LEUPERS, R. AND KOTTE, D. 2001. Variable partitioning for dual memory bank DSPS. In *Proceedings of ICASSP*.

LORENZ, M., KOTTMANN, D., BASHFROD, S., LEUPERS, R., AND MARWEDEL, P. 2001. Optimized address assignment for DSPS with SIMD memory accesses. In *Proceedings of the Asia South Pacific Design Automation Conference* (ASP-DAC, Yokohama, Japan). 415–420.

LU, Y. H., BENINI, L., AND MICHELI, G. D. 2000. Low–power task scheduling for multiple devices. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign*. 39–43.

Luz, V. D. L., Kandemir, M., and Kolcu, I. 2002. Memory management and address optimization in embedded systems: Automatic data migration for reducing energy consumption in multi-bank memory systems. In *Proceedings of the 39th Conference on Design Automation*. 213–218.

Micron. 1999. 1mb syncburst SRAM data sheet. Micron Technology Inc., Boise, ID. Website: www.micron.com.

Prim, R. 1957. Shortest connection networks and some generalizations. *Bell Syst. Tech. J. 36*, 6.

Rambus. 1999. 128/144-mbit direct RDRAM data sheet. Rambus Inc., Losaltos, CA. Website: www.rambus.com.

Saghir, M., Chow, P., and Lee, C. 1996. Exploiting dual data-memory banks in digital signal processors. In *Proceedings of the 7th International Conference on Architecture Support for Programming Language and Operating Systems*. 234–243.

Sudarsanam, A. and Malik, T. S. 2000. Simultaneous reference allocation in code generation for dual data memory bank asips. *ACM Trans. Des. Automat. Electron. Syst. 5*, 2, 242–264.

Wang, Z. and Hu, X. S. 2004. Variable partitioning and scheduling for multiple memory banks. Tech. rep. CSE Dept., University of Notre Dame, Notre Dame, IN.

Wuytack, S., Catthoor, F., Jong, G. D., and Man, H. D. 1999. Minimizing the required memory bandwidth in VLSI system realizations. *IEEE Trans. VLSI Syst. 7*, 4 (Dec.), 433–441.

Zeithofer, T. and Wess, B. 2001. Integrated scheduling and register assignment for VLIW–DSP architectures. In *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference*. 339–343.

Zhuge, Q., Xiao, B., and Sha, E. H.-M. 2001. Exploring variable partitioning in dual data-memory bank processors. In *Proceedings of the 34th International Symposium on Micro-Architecture (MICRO-34), the 3rd Workshop on Media and Streaming Processors (MSP-3 Workshop)*. 42–55.

Zivoljnovic, V., Velarde, J., Schager, C., and Meyr, H. 1994. Dspstone—a DSP oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology*.

Received June 2004; revised October 2004, December 2004; accepted December 2004