Power Aware Variable Partitioning and Instruction Scheduling for Multiple Memory Banks *

Zhong Wang
Dept of Comp. Sci. & Engr
University of Notre Dame
Notre Dame, IN 46556, USA
zwang1@cse.nd.edu

Xiaobo Sharon Hu
Dept of Comp. Sci. & Engr
University of Notre Dame
Notre Dame, IN 46556, USA
shu@cse.nd.edu

Abstract

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 paper, 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.

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 non-orthogonal architecture [7], is characterized by irregular data paths comprising of a heterogeneous register set and multiple memory banks. A number of embedded DSP processors, e.g., Analog Device ADSP2100, Motorola DSP56000 and NEC uPd77016, are based on this architecture.

Harvesting the benefits provided by the non-orthogonal architecture hinges on sufficient compiler support. Parallel operations afforded by multi-bank 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. 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 paper, we focus on two critical steps in the compilation process, i.e., partitioning variables (or data) among the memory banks, and scheduling memory access operations. The decisions made in the two steps can have a significant impact on the overall program code size, execution time and energy consumption.

A number of papers (e.g., [3,7,10,11,15,16,18]) have investigated the use of multi-bank memory to achieve maximum instruction level parallelism (i.e., optimize performance). These approaches differ in either the models or the heuristics (to be discussed in more detail later). However, none of these works consider 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 [6]. To help ease the energy demands by memory, advanced memory modules are designed to operate in different modes, e.g., active, idle and sleep [1, 2]. The exploitation of different operating modes together with multiple memory banks further complicate the problem of variable partitioning and memory operation scheduling. On top of this, performance requirement often conflicts with energy saving. Previous works have studied the effects of the multiple memory operating modes at higher levels such as program basic blocks, system tasks or processes. However, significant energy saving and performance improvements can be obtained by exploiting memory operating modes and multi-bank memory simultaneously at the instruction level.

In this paper, we discuss our approach to variable partitioning and memory operation scheduling in the presence of multi-bank memory and multiple memory operating modes for maximizing energy saving without sacrificing performance. We present observations to help categorize different cases. A novel memory access graph model, capturing simultaneously potential energy savings and potential performance improvements, is proposed to overcome the weakness of previous techniques. Based on this model, we devised an iterative technique to find best energy saving while satisfying the performance constraint.

2. Problem Formulation and Related Work

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). Motorola DSP56000 is an example of such architecture, with three sets of register files and two memory banks. We will use it in our experiments. However, our algorithm can be easily extended to architectures with a homogeneous register set or more memory banks.

* This work is in part supported by NSF under grant number CCR02-08992.

1530-1591/04 $20.00 (c) 2004 IEEE

We consider memory modules used in the memory banks to have two operating modes, i.e., the active mode and the low-current mode (standby or sleep) [2]. The operating mode transition is controlled by memory controller, whose states can be modified through a set of configuration registers [8]. The detailed discussion of controlling memory operating mode transition is beyond the scope of this paper. 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 certain time overhead (and with neglectful current overhead, which can be found in both following data sheets). Two typical examples are Rambus RDRAM module [1] (with 2 clock cycles mode switching time) and Micron SyncBurst SRAM module [2] (with overall 4 clock cycles mode switching time). 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 to a single long idle period than disperse them. This presents some unique challenges to the problem we want to solve, which is formally defined as follows.

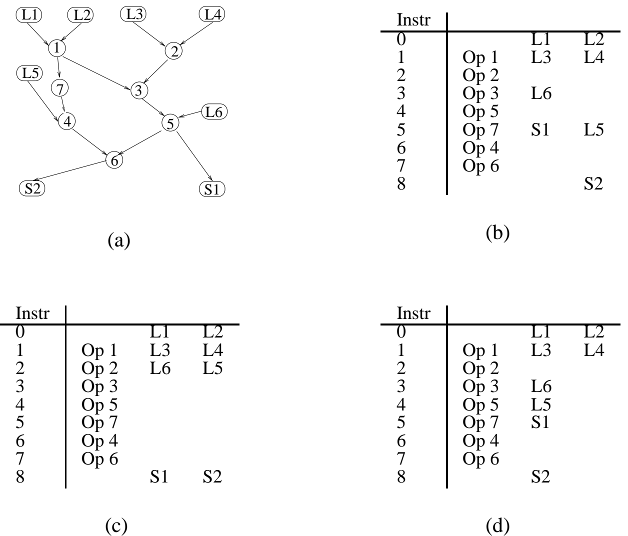
Definition 2.1 *Given a program (in the form of an intermediate code) and a non-orthogonal architecture specification, generate an instruction schedule which improves the memory operation parallelism and energy saving to the largest extent.*

Previous related work can be roughly divided into two main categories: those that use the *compacted* intermediate code as the starting point (e.g., [3, 7, 11, 15]), and those that start with the *uncompacted* one (e.g., [10, 18]). The former approaches often fail to exploit many optimization opportunities, while the latter approaches can explore all possible pairs of memory operations as long as there are no dependencies between them. Therefore, we adopt the practice of starting with the uncompacted code. However, these existing techniques not only pay no attention to energy saving but also have some flaws. Discussing these flaws requires an in-depth discussion of the graph models used, which we postpone to Section 4.

Regarding saving energy through exploiting operating mode changes, a number of research results have been published. The key idea is to distribute idle times judiciously through good scheduling. This can be achieved at various abstraction levels or design stages, e.g., task level for multiple devices [12], process level for scheduling in operating system [9], program basic block level during compile [8] and data memory layout [5, 13]. In contrast, our work focuses on the instruction level. By integrating energy consideration into the instruction scheduling stage, we can achieve additional energy saving without sacrificing performance. Our work complements the above mentioned techniques since it can be applied together with these other techniques.

3. Idle Time Exploration

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 graph (DFG) in Fig 1(a) (in which L (resp., S) followed by an integer i represents a *LOAD (resp., STORE)* operation on variable i. Other nodes are non-memory operations. Edges denote the precedence constraint between operation.), a schedule with only performance consideration is shown in Fig 1(b), while better schedules with respect to both performance and energy are shown in Fig 1(c) and 1(d). In Fig 1(b), the memory modules cannot be switched to the low-current mode because all idle times are too short. In Figure 1(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 1(d), the second memory bank can be put into low-current mode during the control steps $4 \rightarrow 6$ under the assumption that Micron SRAM is adopted. Thus, we gain energy saving without affecting the schedule performance.



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)

Figure 1. (a) An example DFG (b –d) Various schedules

Memory operation scheduling for energy saving 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. A tradeoff exists between energy saving and performance.

In the following, we examine an ideal scenario in which no register constraint exists. The importance of this case will become clear in Section 5, where 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) 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 are three cases depending on the relationship of t, n_i and m.

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

Maximal energy saving can be achieved by Lemma 3.1 (whose correctness is easy to prove and is omitted).

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

For example, the schedule in Fig 1(b) belongs to this case when the operating mode transition time is 2 cycles. The schedule with optimal energy saving in Fig 1(c) can be directly obtained.

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, i.e., $n = \sum_{i=1}^{N} n_i$. 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 saving, one may consider moving memory operations between banks to serialize more operations in one or more banks while leave other banks with longer idle times. The goal is then to maximize "serialism" without hurting performance. The example in Figure 1(d) illustrates such thought for the SRAM memory module. The desire to increase the 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 more optimization can be obtained in such situation. 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 tackle the problem.

4. Graph Modeling Approach

Similar to existing approaches, we use a graph to model our problem constraints and objective. 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. The edge weight assignments introduced in the previous works all have some weaknesses.

A straightforward way of assigning edge weights is to connect two nodes with an edge of weight 1 if the memory operations involving the two corresponding variables do not have data dependencies [10]. However, such potential parallelism may not be always realizable due to certain timing constraints on the associated memory operations. To overcome this difficulty, the authors in [18] introduced the concept of possibility weight. The model does improve on the simple minded approach above, but it still has some flaws. To see why this is the case, we briefly review the possibility weight idea below. 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, i.e., the scheduling freedom of a, defined as [t_s(a),t_l(a)], represents the time range in which a can be scheduled without introducing additional delay. According to this definition, we can esisly derive the mobility for each memory operation in Fig 1(a), i.e., $[0,0]$ for L_1, L_2, $[0,1]$ for L_3, L_4, $[0,4]$ for L_5, $[0,3]$ for L_6, [4,7] for S_1 and [7,7] for S_2. Only when the mobilities of two memory operations have some overlap may parallelizing the two corresponding variables be beneficial (in terms of improving performance). 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. The work in [18] assigns a possibility weight defined below to an edge to model this property.

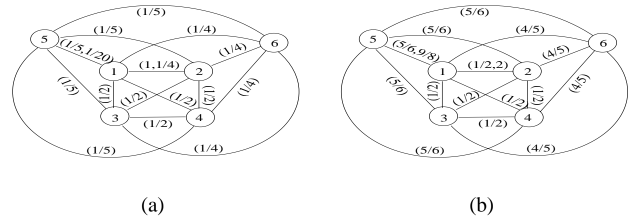


Figure 2. (a) Parallelism weight (b) Serialism weight

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* $\frac{t_2 - t_1 + 1}{(t_l(a) - t_s(a)+1)(t_l(b) - t_s(b)+1)}$.

Fig 2(a) shows an example of this possibility weight assignment for the memory operations given in Fig 1(a). In Fig 2(a) (each node number corresponds to the associated variable number), 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 [18], such numbers are simply added together. Moreover, if the same operation pattern (e.g., overlapping of $L1$ and $L5$) occurs in another *mobility range* (A mobility range is a period of consecutive scheduling steps which may cover several variables' mobility. In this paper, whenever we talk about two different mobility ranges, they should be independent of each other without overlap.), the possibility weight is again added to the edge weight between v_1 and v_5.

One flaw (**flaw #1**) of the possibility weight model in [18] is the above mentioned summation of the possibility weights. Consider a simple example of an edge possibility weight of 1. This may come from two operations as $L1$ and $L2$ in Fig 1(a). It may also come from two occurrences of the operation pair such as $L3$ and $L4$ in Fig 1(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 to maintain 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.)

Another flaw (**flaw #2**) of 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 [18] treats the two groups indiscriminately.

Besides the above flaws, the possibility weight model have no consideration about energy saving because the work in [18] focuses only on performance. From the point of view of energy saving, we prefer to serialize memory operations as much as we can so as to leave more idle times for the low-current mode (see the discussion of Case 2 in Section 3). Clearly, this preference towards serialism may run against the requirement of improving performance.

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, i.e., 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. Take 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 2(b). 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. 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 multi-weighted 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, $\vec{w_p}$ and $\vec{w_s}$ are the parallelism and serialism weight lists, respectively.*

Though we are able to indicate the requirements of performance and energy saving through introducing both parallelism and serialism weights, we need to 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 [4]. A number of excellent heuristics exist for solving the maximum k-cut problem [4]. 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 saving, 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

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

where λ_p, λ_s are two coefficients representing the tradeoffs between parallelism and serialism, $w_p(h, i, j)$ (resp., $w_s(h, i, j)$) is the parallelism (resp., 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 reason behind the subtraction used in (Eqn. 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 saving.

It is important to point out that the average weight defined in (Eqn. 1) also overcome the flaws mentioned earlier in the model introduced in [18]. For **flaw #1**, under our average weight model, 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 to separate memory banks as they have more stringent timing requirement. For **flaw #2**, 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).

The complete algorithm in the following section will reveal how to select the coefficients.

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 framework is shown in Algorithm 1.

Algorithm 1

Input: Intermediate Code, Register Constraints
Output: An optimized code
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. $\lambda_p = 1, \lambda_s = 0, \lambda'_s = 0$, calculate the average weight for each edge and schedule the program. (λ'_s is used to remember the value of λ_s in the previous loop iteration.)
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

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 $T_{schedule}$. $L_{schedule}$ 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 $T_{schedule}$ have not shown significant changes for σ number of loops.

In Line 1 of Algorithm 1, the technique in [17] is used to deal with heterogeneous register set and register constraint. In Line 5, the well known maximum spanning tree algorithm [14] is used as the maximum-cut heuristic.

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}(resp., w_{max}) is the minimum (resp., maximum) value of all $w_s(i,j)$ values.

The loop of Algorithm 1 is used to find a point where the maximal energy saving is achieved with the accepted 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 saving, while more serialism (smaller λ_p and larger λ_s) may bring more energy saving, but a possible deteriorated performance. Through a process analogous to a binary search, we find proper λ_p and λ_s values to adjust parallelism and serialism accordingly. The best tradeoff point is reached where both the goals of performance and energy saving are obtained.

6. Experimental results

We have implemented our algorithm in the SPAM compiler environment to replace the simulated annealing algorithm [3] originally used by SPAM. Benchmarks come

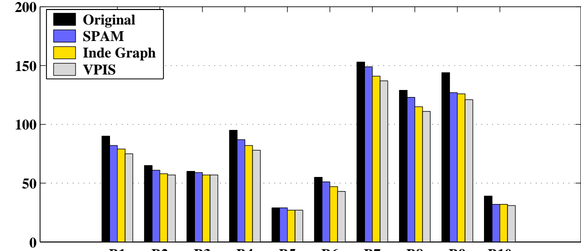


Figure 3. Assembly code size results

from DSPstone benchmark suite [19], 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 assembly code size results are shown in Fig 3, which are the comparison of *original code size (Original), code size generated by constraint graph methods (SPAM), code size generated by [18] (Inde Graph) and code size generated by our algorithm(VPIS)*.

Fig 3 reveals that the methods of independence graph and our algorithm can both perform better than SPAM. This improvement can be attributed to thoroughly exploit more potential memory operations parallelism. Due to the comprehensive graph model, our algorithm demonstrates a superior performance to the method of independence graph. The execution time of the assembly code is correlated to the code size [10], since the assembly code can be directly mapped to the schedule for the basic block. Thus, we omit the data for benchmarks' actual execution time comparison. In fact, a more significant improvement of execution time can be expected due to this code size improvement.

We compare the energy saving results of our algorithm with SPAM. Results from *Inde Graph* methods are not included in this comparison, since it does not consider energy saving. In fact, it can be regarded as a special case of our algorithm with the restriction of $\lambda_p = 1, \lambda_s = 0$. Fig 4 shows the ratio of summation of all effective consecutive idle cycles (the consecutive idle cycles after deducting the operating mode transition time) to the overall code size. One can see that our algorithm can result in more effective idle times than SPAM. The average improvement of the ratio is 19.84%. It is worthwhile pointing out that we obtain this improvement ratio with the shorter code size.

Note in this comparisons, the control flow information of the program is not included, though this information is considered in calculating the operations' mobility from CDFG.

Our algorithm can achieve the larger improvements in the data intensive than control intensive application code. We anticipate better showdown of the energy saving improvement since data intensive codes usually are executed many times.

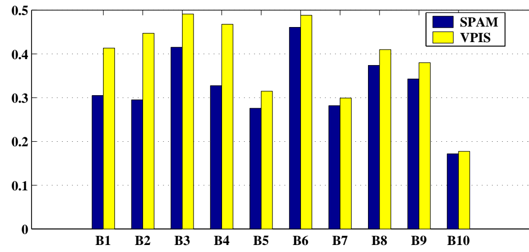


Figure 4. Percent of idle cycles over code size

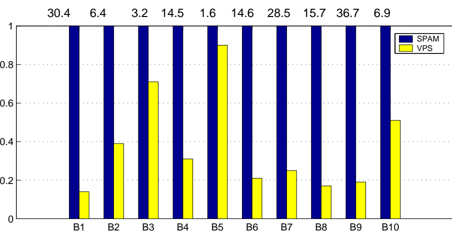


Figure 5. Algorithm execution time

As of the algorithm execution time, due to the fact that variable partitioning is not very sensitive to the change of average coefficients λ_p and λ_s (small change in these two coefficients does not change the variable partition), the algorithm generally can find the best tradeoff point in at most twenty loops. We ran the program in SUN Ultra Sparc2 and the algorithm execution time comparison is shown in Figure 5. In the figure, we normalized the algorithm execution time by the simulated annealing algorithm (adopted by SPAM) execution time and annotate its value in the unit of second on the top of each benchmark.

With more complicated programs, the constraint graph of simulated annealing algorithm becomes larger and each step in annealing process takes longer time. The algorithm execution time increases significantly with the constraint graph size. While our algorithm, by contrast, does not have to deal with the large graph for many times (at most twenty loops for our experiments). Therefore, the execution time improvement becomes more obvious for large benchmarks.

7. Conclusion

A variable partitioning and instruction scheduling algorithm is proposed to explore the non-orthogonal architecture. 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. The memory module consecutive idle time is maximized under the constraint of the schedule performance. Experimental results demonstrate that our algorithm outperforms the previous techniques.

References

[1] 128/144-mbit direct rdram data sheet, May 1999. Rambus Inc.

[2] 1mb syncburst sram data sheet, Sept. 1999. Micron Technology Inc.

[3] S. A and T. S. Malik. Simultaneous reference allocation in code generation for dual data memory bank asips. *ACM TODAES*, 5(2), 2000.

[4] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation*. Springer Verlag, 1999. ISBN 3-540-65431-3.

[5] L. Benini, A. Macii, and M. Poncino. A recursive algorithm for low-power memory partitioning. In *International Symposium on Low power Electronics and Design*, Aug 2000.

[6] F. Catthoor, S. Wuytack, E. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology –Exploration of Memory Organization for Embedded Multimedia System* . Kluwer Academic, 1998.

[7] J. Cho, Y. Paek, and D. Whalley. Efficient register and memory assignment for non–orthogonal architectures via graph coloring and mst algorithms. In *ACM Joint Conference LCTES-SCOPES*, pages 130–138, Berlin, Germany, Jun 2002.

[8] V. Delaluz, N. V. M. Kandemir, A. Sivasubramaniam, and M. J. Irwin. Hardware and software techniques for controlling dram power modes. *IEEE Transactions on Computers*, 50(11), Nov 2001.

[9] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Scheduling techniques for embedded systems: Scheduler-based dram energy management. In *39th conference on Design automation*, June 2002.

[10] R. Leupers and D. Kotte. Variable partitioning for dual memory bank dsps. In *Proceeding of ICASSP*, 2001.

[11] M. Lorenz, D. Kottmann, S. Bashfrod, R. Leupers, and P. Marwedel. Optimized address assignment for dsps with simd memory accesses. In *Asia South Pacific Design Automation Conference (ASP-DAC),*, pages 415–420, Yokohama, Japan, Jan 2001.

[12] Y. H. Lu, L. Benini, and G. D. Micheli. Low–power task scheduling for multiple devices. In *8th international workshop on Hardware/software codesign*, May 2000.

[13] V. D. L. Luz, M. Kandemir, and I. Kolcu. Memory management and address optimization in embedded systems: Automatic data migration for reducing energy consumption in multi-bank memory systems. In *39th conference on Design automation*, Jun 2002.

[14] R. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, 36(6), 1957.

[15] M. Saghir, P. Chow, and C. Lee. Exploiting dual datamemory banks in digital signal processors. In *7th International Conference on Architecture Support for Programming Language and Operating Systems*, pages 234–243, 1996.

[16] S. Wuytack, F. Catthoor, G. D. Jong, and H. D. Man. Minimizing the required memory bandwidth in vlsi system realizations. *IEEE Trans. on VLSI Systems*, 7(4), DEC 1999.

[17] T. Zeithofer and B. Wess. Integrated scheduling and register assignment for vliw–dsp architectures. In *14th Annual IEEE International ASIC/SOC Conference*, pages 339–343, 2001.

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

[19] V. Zivoljnovic, J. Velarde, C. Schager, and H. Meyr. Dspstone - a dsp oriented benchmarking methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*, 1994.