Energy-Oriented Compiler Optimizations for Partitioned Memory Architectures

V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin
Microsystems Design Laboratory
Pennsylvania State University
University Park, PA 16802
mdl@cse.psu.edu

Abstract

Due to low power requirements of many embedded/portable devices such as mobile phones and laptop computers and dramatic increases in clock frequencies of general-purpose processors, low-power software technology is becoming increasingly important in system design. Many applications from image and video processing as well as from dense linear algebra are array-dominated and data-intensive, thereby spending a major portion of their execution time and energy in the memory subsystem. This paper presents a compiler-based optimization framework that targets reducing the energy consumption in a partitioned off-chip memory architecture that contains multiple memory banks by organizing the order of computations and the layout of data. The optimizations considered in this work take advantage of low-power operating modes and the partitioned (multi-bank) structure of the off-chip memory. Our preliminary experiments show that the proposed framework improves memory energy by up to 86% over a scheme that keeps all the memory banks in the active (fully-operational) operating mode all the time, and up to 70% over a scheme that utilizes low-power operating modes without doing any loop and data optimizations.

1 Introduction

Even though the computer science and information technology community has devoted a lot of time and effort to the advance of computing systems that are high-performance desktops and servers, it is estimated that more than 90% of the processor market today is in the embedded systems area [16]. Considering that many embedded/portable devices are battery powered, it is easy to see that energy consumption is an important issue for these devices. Reducing energy consumption not only increases the lifetime of the battery, but it also improves the reliability of the device by decreasing its heat dissipation.

The question of where the energy is spent in a particular architecture depends largely on the type of the architecture (its hardware components and interconnects) as well as the software that runs on it. Many embedded image and video processing applications as well as linear algebra kernels are data-intensive, accessing large multi-dimensional arrays in small multi-level nested loops. Previous research reports that the embedded systems that run these applications spend up to 90% of their energy in the memory system [2]. The problem of reducing memory energy consumption might even be more critical for systems with energy-optimized processors.

Motivated by these observations, this paper presents a compilation approach to minimize the energy consumed in a partitioned off-chip memory (DRAM) architecture that consists of multiple memory banks, each of which can be operated in a number of operating modes (power modes). In such a memory architecture, an important issue is to restructure code and data such that as many memory banks as possible can be put into a low-power operating mode without impacting execution time (performance). The compilation technique presented in this paper attempts to address this issue using a set of loop (iteration space) and memory layout (data space) optimizations in a unified framework. Specifically, we make the following contributions:

- We summarize the operation of a multi-bank memory system and explain how low-power operating modes can reduce its energy consumption.
- We present a data placement (array allocation) algorithm that places the arrays used in an application across memory banks in an energy-conscious way to minimize the energy spent in the memory system. Informally, the approach tries to cluster the arrays with *similar lifetime patterns* in the same (set of) bank(s) to power-down the said bank(s) when the lifetime of the arrays ends.
- We present a transformation framework that uses two loop optimization techniques (loop fission and loop splitting) and a data optimization approach (which we call *array renaming*) to increase the effectiveness of the array allocation algorithm. While these optimizations, themselves, are not new, their application to the problem of memory energy reduction is novel.
- We give experimental data showing that it is possible to obtain energy savings up to 86% over a scheme that keeps all the memory banks in the active (fully-operational) state all the time, and up to 70% over a scheme that utilizes low-power operating modes without doing any loop and data optimizations.

The data allocation part of our approach has been implemented within the SUIF [20] compilation framework. Our experimental data shows that the loop and data transformations can be of great help in optimizing memory energy. Moreover, since these energy benefits come without performance loss and with only a small increase in datapath (core) energy, the proposed approach can even be used in high-end desktops and servers with an appropriate help from the operating system (OS).

The remainder of this paper is organized as follows. Section 2 discusses a multi-bank memory subsystem and shows how low-power operating modes work. Section 3 presents our optimization framework in detail and explains how it benefits the low-power operating modes and multi-bank (partitioned) memory system. Section 4 introduces our experimental platform and presents energy results. Section 5 discusses related work, and Section 6 concludes the paper with a summary and an outline of the planned future research.

2 Partitioned Memory and Low-Power Operating Modes

The target system for our approach is a partitioned memory architecture with multiple banks and low-power operating modes. Each bank operates independently and when not in active use it can be placed into a low-power operating mode to conserve energy. Each operating mode works by activating specific portions of the memory circuitry and can be described using two related metrics: *energy consumption* and *re-synchronization time*. The energy consumption is the amount of energy consumed *per cycle* in a given operating mode. The re-synchronization time is the time (in cycles) it takes to bring a memory bank from a low-power mode to the active (fully-operational) mode. Typically, the smaller the energy consumption, the higher the re-synchronization time. Consequently, the selection of low-power operating mode has both energy and performance impact and usually involves a tradeoff between them.

For the purposes of this paper, we assume five operating modes: one *active* mode (the only mode during which the memory read or write activity can occur) and four low-power modes, namely, *standby, napping, power-down,* and *disabled.* The energy consumptions and re-synchronization times for these modes are given in Figure 1.[1] It should be noted that a naive use of these modes can incur a significant performance loss. Let us assume, for example, that we place an idle (inactive) bank into the *power-down* mode. At the end of the idle period, in order to bring the bank into the active mode, we need to wait 9,000 cycles (see Figure 1) during which we also consume some amount of energy (whether it is full active mode energy or low-power mode energy or some amount between these two is system-dependent). To alleviate this problem, the compiler can use a *pre-activation strategy*, which activates a bank (in the low-power mode) *before* it is needed so that it will be ready (in the active mode) when it is actually needed. This approach bears similarity to software prefetching techniques in the optimizing compiler literature [10], and in fact, the compiler can use a similar algorithm to insert mode-setting commands (i.e., the commands that change the operating mode at runtime) in the code.

The compiler analysis to insert mode-setting commands starts with a pre-processing pass through which the program being optimized is transformed into a form with as many independent perfectly-nested loops as possible.[2] Then, for each nested loop, the compiler determines the arrays accessed by the nest and the memory banks that hold those arrays. These memory banks need to be active during the execution of the nest. The remaining memory banks, however, can be put into low-power operating mode. A sophisticated compiler analysis [4] determines the number of loop iterations before the end of the nest that these banks should be activated so that they will be in the fully-operational state (i.e., the active mode) for the execution of the next loop (i.e., no performance penalty).[3] The exact low-power operating mode to be used (as we have four of them) is determined based on the energy consumption and re-synchronization time of each possible mode and the duration of the idleness (i.e., the time for the nested loop to execute minus the time needed to pre-activate the bank). The duration of the idleness is estimated by the compiler by taking into account the loop bounds and the computational cycle costs of each type of operation performed in the nest.

The algorithm for power mode selection is quite involved and can be found in [4]. This paper focuses more on the use of loop and data transformations and allocation of arrays to banks to take better advantage of low-power operating modes. As a simple example illustrating the working of low-power operating mode management, consider the program fragment shown below (next column) assuming that each array resides in a separate memory bank. In this code, the first nest accesses the arrays `a` and `c`. Therefore, the memory

[1] The energy values shown in this figure have been obtained from the measured current values associated with memory banks documented in memory data sheets (for a 3.3 V, 2.5 ns cycle time, 8 MB memory bank) [13]. The re-synchronization times are also obtained from data sheets.

[2] A perfectly-nested loop is a loop nest in which all statements are contained in the innermost loop [21].

[3] Actually, only the banks that will be used in the next nested loop are activated.

	Active	Standby	Napping	Power-Down	Disabled
Energy Consmp. (nJ)	3.570	0.830	0.320	0.005	0.000
Re-sync. Time (cyc.)	0	2	30	9,000	NA

Figure 1: Energy consumptions and re-synchronization times.

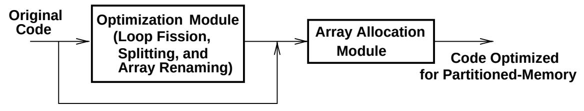

Original Code
Optimization Module (Loop Fission, Splitting, and Array Renaming)
Array Allocation Module
Code Optimized for Partitioned-Memory

Figure 2: Optimization framework.

banks that hold these arrays should be in the active mode during the execution. On the other hand, the memory bank that holds array `b` can be put into a low-power operating mode as this array is not accessed in the first nest. The second nest accesses `b` and `c`; consequently, before it starts execution, the bank that holds array `b` should be activated (In fact, as mentioned earlier, this activation should occur before the first nest terminates; the exact timing is determined by the compiler [4]). At the same time, since array `a` is not referenced in the second nest, its memory bank can be placed into a low-power mode. The compiler-based approach explained in [4] inserts these bank activation/deactivation (mode-setting) commands in appropriate places in the code (by calculating the exact points and modifying the code if necessary) and determines which low-power operating mode to be used for an idle bank to achieve maximum energy savings.

```
for(i=0;i<N;i++)
 {a[i],c[i]}
for(i=0;i<N;i++)
 {b[i],c[i]}
```

3 Optimization Framework

As shown in Figure 2, our optimization framework consists of two modules: an *optimization module* (that uses loop and data transformations) and an *array allocation module.* The array allocation module can be used with or without the optimization module and is responsible from assigning memory locations (hence banks) to array elements. The optimization module, on the other hand, uses both loop and data transformations to further increase the effectiveness of the array allocation module.

3.1 Array Allocation Module

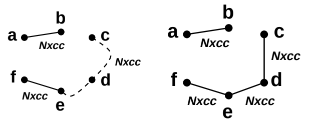
Informally, the idea behind the array allocation (data placement) algorithm is to place the arrays with the same (or similar) access patterns into the same (set of) bank(s). This is reasonable as such arrays are usually used (or not used) at the same time during which the memory bank(s) that hold them can be turned on (or off). An important issue then is to determine the arrays with similar access patterns and cluster them together. To achieve this, we use a graph-based approach which is sketched in Figures 3 and 4. Our approach operates in two steps. The first step (given in Figure 3), which is similar in spirit to the formulation of offset assignment problem [8] for scalar variable placement, builds a graph (called *array relation graph* or ARG for short) in which the nodes represent the arrays declared in the program and the weight of an edge (resp. hyper-edge) represents the number of times (in cycles) two (resp. multiple) arrays that are incident on the edge (resp. hyper-edge) are accessed in the *same* nested loop. It then runs an algorithm on this ARG that generates a *maximum weight cover* (the ‘while loop’ in Figure 3).

The maximum weight cover of a graph is path (which includes all the nodes in the graph but not necessarily all the edges) whose sum of all the edges in the path is the maximum among all covers. Basically, the algorithm in Figure 3 selects (in each iteration of the 'while loop') an edge with the maximum weight (among all unselected edges) as long as the selected edges do not form a cycle. At the end, for each connected component of the ARG being analyzed, this algorithm returns a path (referred to as C_i).

Each path C_i contains arrays (corresponding to nodes) that should be placed if possible into the same (set of) bank(s) as they are used together (in the same computation). Note that this step of the approach is independent of the underlying memory bank organization. The second step of the approach (given in Figure 4) takes the memory organization into account and determines the exact locations of array elements. While doing this, this step also uses array alignment [5] to prevent an array from crossing the bank boundaries (hence from keeping multiple banks in the active state) if this is not strictly necessary. The only restriction is that all elements of a given array are stored consecutively, in an attempt to prevent address calculation complexity that would occur otherwise (Note that such address calculation complexities would increase the energy expended in the datapath). More specifically, the algorithm in Figure 4 runs for each path returned by the algorithm in Figure 3. For each path, it first traverses the path from one end to the other and places the arrays consecutively into available banks starting from the first location of the first bank. Then, within the 'while loop', it tries to put the arrays whose corresponding vertices are the end points of edges with large weights into the same bank by aligning the arrays (this can be thought of sliding the arrays to the right to ensure that the two arrays in question reside in the same bank if possible). While processing a given edge (hence its arrays), special attention is paid not to distort any placement made for a prior edge (with a higher weight).

As an example, let us consider the following program fragment whose ARG with and without hyper-edge is shown next to it. Note that there are two edges, (`a`,`b`) and (`e`,`f`), and one hyper-edge (`c`,`d`,`e`) (shown in the first graph as a dashed path). This hyper-edge is then transformed into two (normal) edges (`c`,`d`) and (`d`,`e`) (as shown in the last graph), all with the same edge weight $N \times cc$ assuming that each nest has a cycle count of cc. Two obvious paths in this last graph are `a-b` and `c-d-e-f`. Assuming that the arrays are of the same size and a memory bank can hold two arrays, the algorithm in Figure 4 places `a` and `b` into the first bank, `c` and `d` into the second bank, and `e` and `f` into the third.

```
for(i=0;i<N;i++)
  {a[i],b[i]}
for(i=0;i<N;i++)
  {c[i],d[i],e[i]}
for(i=0;i<N;i++)
  {e[i],f[i]}
```



3.2 Optimization Module

The optimization module uses three techniques to improve the effectiveness of the array allocation module: *loop fission, loop splitting,* and *array renaming.* The first two techniques are loop-based optimizations whereas the third one is a data transformation method. Note that these optimizations are by no means exhaustive; on the contrary, we believe that many other loop and data transformations can be adapted/redirected to target partitioned-memory architecture with multiple operating modes. We focus on these three optimizations as our preliminary experiments using them show significant savings in memory system energy without an observable increase in datapath (core) energy. We do not present datapath energy results here due to lack of space.

```
INPUT: A program with an original array declaration
OUTPUT: A program with the optimized array declaration
Begin
  Find the arrays used in each loop nest
  Find the weight of each loop nest
    • this weight will be the cycles per iteration
    multiplied by the number of iterations of the loop
  Create the undirected weighted graph $ARG = G\langle V, E\rangle$ as follows:
    • $V$ represents the arrays and $e = (v_1, v_2) \in E$
    if the arrays corresponding to $v_1$ and $v_2$ are used in the same
    loop nest. If more than two arrays are used in the
    same loop, a hyper-edge (spanning multiple nodes) will be formed
  Decompose hyper-edges into normal edges as follows:
    • a hyper-edge such as $(v_i, v_{i+1}, v_{i+2}, v_{i+3}, \dots, v_j)$ is divided into
    edges $(v_i, v_{i+1})$, $(v_{i+1}, v_{i+2})$, $(v_{i+2}, v_{i+3})$,...
    • each edge will have the same weight as the original hyper-edge
    • if the same $e$ needs to be created for different loop nests, then its weight
    will be the sum of the weights of the loop nests
    in which the two associated vertices are used
    • note that this might be a non-connected graph, in which
    case we have $i$ independent graphs where $G_i\langle V_i, E_i\rangle$ denotes the $i^{th}$ graph
  For each ARG $G_i$
      Create an empty graph $C_i\langle V_i', E_i'\rangle by V_i' \leftarrow \emptyset, E_i' \leftarrow \emptyset$
      $E_{sort} \leftarrow$ sorted list of edges in $E_i$ in descending
        order of weight
      While $E_{sort} \neq \emptyset$
          Choose $e$=$(v_1, v_2)$ at the beginning of $E_{sort}$
          $E_{sort} \leftarrow E_{sort} - e$
          If ($e$ does not cause a cycle in $C_i$) and
            (any vertex in $V_i'$ has degree$> 2$)
              Add $e$=$(v_1, v_2)$ to $E_i'$ and $v1, v2$ to $V_i'$
          Else
              Discard $e$
          EndIf
      EndWhile
      • the declaration order of arrays will be obtained by
      traversing $C_i$
      If a vertex $v$ is not considered in the ordering,
        put that vertex at the end of the ordering
  EndFor
  If some arrays are not considered in the orderings previously
    obtained, put them at the end
      • These arrays are referred to as 'wildcards'
  EndIf
End
```

Figure 3: An algorithm that clusters arrays with similar lifetime patterns.

3.2.1 Loop Transformations

Loop Fission: Loop fission (also known as loop distribution [21]) takes a nested loop that contains multiple statements in it and creates multiple nested loops each with a subset of the original statements. An optimizing compiler can use loop fission for a number of reasons which include improving instruction cache locality and enhancing iteration-level parallelism. This optimization first builds a statement-level dependence graph (in which the nodes denote statements and the edges correspond to data dependences between them) for the body of the nested loop. Then, if there are no cycles in the dependence graph, the optimization creates a separate loop for each statement. Otherwise, the statements in a cycle remain in the same loop. If the nested loop contains multiple loops, loop fission is applied starting from the innermost position, and in fissioning the outer loops, the inner loops are treated as single block statements. Note that even in cases where each statement can be put into a separate loop nest an optimizing compiler may choose not to do so for some other reason such as improving data cache locality or reducing code expansion.

Loop fission helps to improve the effectiveness of the array allocation module by allowing a finer-granular control over the allocation of arrays. For instance, in the example shown below (assuming that the arrays are of the same size and each memory bank can hold at most two arrays), with the original nest, the two banks that contain the four arrays should be in the active mode throughout the entire execution. After the loop fission, on the other hand, only a single bank needs to be in the active mode during the execution of each loop (assuming that the array allocation module places `a` and `b` into one bank, and `c` and `d` into the other). The other bank can be put into a low-power mode, thereby saving energy.

```
for(i=0;i<N;i++)                  for(i=0;i<N;i++)
 {                                  {a[i],b[i]}
  {a[i],b[i]}           ==>        for(i=0;i<N;i++)
  {c[i],d[i]}                       {c[i],d[i]}
 }
```

The overall algorithm for loop fission for energy is given in Figure 5. The purpose of this algorithm is to try different loop fissioning strategies for a given nested loop. If there are K instructions within the loop nest, the main 'for loop' in the algorithm enumerates $K-1$ alternatives. The ith alternative is formed by breaking up the nest into two nests after the ith statement from the beginning of the nest (if it is legal to do so). We also add two more alternatives to these $K-1$ alternatives: the original nest and a code with K nested loops each with its own statement (again, if this is legal). The $K+1$ alternatives considered by the algorithm in Figure 5 are as follows:

```
                        for(...)
           for(...)      {                    for(...)
for(...)    S_1           S_1                  {          for(...)
 {         for(...)       S_2                   S_1        S_1
  S_1       {            }                      S_2       for(...)
  S_2        S_2        for(...)                S_3        S_2
  S_3    ,   S_3     ,   {          ,....       ...   ,   for(...)
  ...        ...          S_3                   S_{K-1}    S_3
  S_{K-1}    S_{K-1}      ...                  }           ...
  S_K        S_K          S_{K-1}             for(...)    for(...)
 }          }             S_K                   S_K        S_K
                         }
```

For each alternative, the estimated energy consumption is calculated, and the alternative with the minimum energy consumption is selected.

```
INPUT:
  A program which uses arrays $v_1, v_2, \ldots, v_n$ whose
    orderings were computed with algorithm in Figure 3
  $L$ memory banks, each one is of size $M$ (the total available
    memory is $LM$ bytes)
  $\hat{v}$ : size of array $v$.
  The total memory required for the arrays $M_{tot}$
  We assume that $LM > M_{tot}$
OUTPUT: An optimized program with with arrays mapped into
  memory banks
  Begin
  For each graph (path) $C_i\langle V_i', E_i'\rangle$:
      Traverse $C_i$ from one end to the other and place the
      corresponding arrays consecutively into available banks
      starting from the first location of the first bank
      Assign the ordering in sequential order to a new bank as follows:
      $E_{sort} \leftarrow$ sorted list of edges in $E_i'$
      in descending order of edge weights
      Mark all $v \in V_i'$ as 'not fixed'
      While $E_{sort} \neq \emptyset$
          Choose $e = (v_1, v_2)$ at the beginning of $E_{sort}$
          $E_{sort} \leftarrow E_{sort} - e$
          If (($v_1, v_2$ are mapped to different banks) and
            (($\hat{v_1} + \hat{v_2}) < M$) and
            ($v_1, v_2$ are not fixed) and
            (there are banks remaining)
              If by shifting $v_1, v_2$ to the right they are mapped
                to the same bank and this action does not distort
                any previous mapping
                  Fix the mapping of $v_1, v_2$
                  Mark $v_1, v_2$ as 'fixed'
              EndIf
          EndIf
      EndWhile
  EndFor
  Assign the 'wildcards', one at a time
    • If their size allow, they are mapped into used banks;
    otherwise, new banks are used
End
```

Figure 4: An algorithm that places arrays into memory banks.

```
INPUT: A perfect loop nest containing $K$ instructions
OUTPUT:
  An energy-optimized loop-fissioned code
  (B contains the number that identifies the loop fissioned
  version with minimum energy)
Begin
  Let $E_{min}$ be the least computed energy consumed
    by the loop nest
  Let $B$ be the number of the more energy saving option.
  $E_{min} \leftarrow$ The energy consumed by the original loop nest
    (without fission)
  $B \leftarrow 0$
  For i = 1 to K - 1
      Create two loop nests instead of the original loop nest
        • the first containing the first $i$ instructions and
        the second the remaining $K - i$ instructions
      Compute the energy $E_i$ consumed by this new program.
      If $E_{min} > E_i$
          $E_{min} \leftarrow E_i$
          $B \leftarrow i$
      EndIf
  EndFor
  Create K loop nests, instead of the original loop nest
    • each one contains one instruction of the original
    nested loop
  Compute the energy $E_K$ for this new program
  If $E_{min} > E_K$
      $E_{min} \leftarrow E_K$
      $B \leftarrow K$
  EndIf
End
```

Figure 5: An algorithm that improves memory energy consumption using loop fission.

```
INPUT:
  A loop nest that accesses the arrays $v_1, v_2, \ldots, v_n$
  The memory layout is assumed to be row-major
OUTPUT: An optimized splitted loop nest
Begin
  For each array used in the loop nest
      compute the tuple(array name, number of memory
        banks used for it), i.e., $(v_i, M_i)$
  EndFor
  Create a list $L$ of tuples $(v_i, M_i)$ sorted in
      descending order of number of banks used
  Create a list $I$ of loop indices used in the nest, from the
      innermost loop to outermost loop.
  While ($I$ != $\emptyset$)
      Choose $i$ at the beginning of $I$
      flag $\leftarrow$ 0
      pointer $\leftarrow$ beginning of $L$
      While ((pointer != NULL) and (flag != 0))
          $x(v, M) \leftarrow$ L(pointer)
          If (the leftmost index in all appearances
            of array $v$ in the body of the loop is $i$)
              Split the loop corresponding to index $i$
              into M loops
              flag $\leftarrow$ 1
              Delete $x$ and all tuples $B_j$ in which array
                $a_j$ has $i$ as its leftmost index
          Else
              Advance the pointer
          EndIf
      EndWhile
      $I \leftarrow I - i$
  EndWhile
End
```

Figure 6: An algorithm that improves memory energy consumption using loop splitting.

Loop Splitting: This optimization (also called index set splitting [21]) divides the index set of a nested loop into two or more disjoint parts. It was originally developed to break some data dependences by placing the source iteration of the dependence and the target iteration into separate loops. Note that since this optimization does not change the execution order of loop iterations, it is always legal.

Loop splitting has an important use in our framework for large arrays that span multiple banks. For the example shown below, assume that each of a and b spans two memory banks (i.e., total four memory banks are needed). During the execution of the original loop (the left one), all four banks should be in the active state. After the loop splitting, however, only two memory banks need to be in the active mode during the execution of each loop.

```
                              for(i=0;i<N/2;i++)
for(i=0;i<N;i++)               {a[i],b[i]}
 {a[i],b[i]}          ==>     for(i=N/2+1;i<N;i++)
                               {a[i],b[i]}
```

Figure 6 gives the algorithm for loop splitting. This algorithm tries to modify the structure of a nested loop by creating several new (splitted) nests out of the original one using loop splitting. The algorithm works from the innermost loop index to the outermost loop index (in the first 'while loop'). For each loop index, it determines a suitable splitting strategy considering the array references that use this index in their first (leftmost) dimension (assuming row-major memory layout) and the array and memory bank sizes. The complete algorithm is quite involved and omitted here due to lack of space.

3.2.2 Array Renaming

The optimizations in the previous subsection modify the access pattern of nested loop to affect the bank activation/deactivation pattern. Array renaming is an optimization that exploits the result of *live variable analysis* [11] to reuse the same memory space for storing multiple array variables whose lifetimes are disjoint. The code fragment below shows an example case with two arrays of disjoint lifetimes (a and b). That is, we assume that after the first nest, the array a is not needed, hence its memory space can be *reused* for some other array (in our case, the array b). Supposing that the arrays are of the same size and each bank can hold two arrays, in the original code (the left one), the array allocation module might place a and b into the same bank and c into another bank. In this case, during the execution of the first nest, only the first bank will be in the active mode and during the execution of the second nest, two banks will be in the active mode. On the other hand, if we reuse the same space for a and b (as shown on the right in the code), only one memory bank will be used during the both nests.

```
for(i=0;i<N;i++)          for(i=0;i<N;i++)
 {a[i],c[i]}               {a[i],c[i]}
 .....             ==>     .....
for(i=0;i<N;i++)          for(i=0;i<N;i++)
 {b[i],c[i]}               {a[i],c[i]}
```

A sketch of the array renaming algorithm is given in Figure 7. To rename the arrays, first the array and memory bank usages for each loop nest are found. Then, for each array the first and last nest of usages are determined (i.e., the liveness information for each array). Next, if the first-usage loop nest for an array b is greater than the last-usage loop nest of another array a and the size of a is larger than that of b, the array b is dropped and its uses in the code are replaced by that of a. At the end of the algorithm, a new array declaration is found. Note that if the arrays with disjoint lifetimes have different dimensionalities and/or sizes, this optimization might demand sophisticated renaming operations (e.g., converting from a three-dimensional array to a two-dimensional array). Recent developments in data transformations area (e.g., [7, 12]) can be exploited to achieve this.

4 Experiments

We evaluated the effectiveness of the proposed framework using eleven *array-dominated* benchmarks. Figure 8 gives the important characteristics of these benchmark codes. Our benchmarks include two codes (matvec and mxm) widely used in image-processing applications and a motion-estimation code (phods). The figure also shows the total data size (input size) of each application, the memory bank configuration used, and the energy consumption for that configuration assuming that all the memory banks are active all the time (i.e., whether they are in active use or not). To keep the memory energy optimization problem tractable, this paper bases the experimental results on a single program environment and does not consider the virtual memory system (i.e., we assume that the compiler directly deals with physical addresses). Exploring the influence of multi-programmed executions to study the impact of colocation of data structures *across* programs and the presence of a virtual address translation are part of our future planned research. It should be noted that many embedded environments [6] operate without any virtual memory support, and the results from this paper would directly apply in those cases. While most of the evaluations in this paper have used a partitioned multi-bank memory system without a cache, we have also experimented with two cache configurations (Section 4.6).

Figure 9 gives percentage (%) energy consumption for each nested loop in each benchmark. It is easy to see that for many

```
INPUT:
  A program which uses the arrays $v_1, v_2, \ldots, v_n$
  $S_i$: size in bytes of array $v_i$
  $A$: the list of arrays declared in the program
  $M$: number of memory banks and $M_s$ the size
    of each memory module
OUTPUT: A new declaration order ($A$) and optimized program
Begin
  Obtain the array usage per loop nest
  Create a list $L_i$ for array $i$, containing the nests in which
  array $i$ is used throughout the program
  For each loop nest $C$
    compute the number of memory banks ($B_C$) used
    in the loop nest
  EndFor
  For each list $L_i$
    If ($L_i == \emptyset$)
      $A \leftarrow A - a_i$
      Update $U_g$ of all arrays $g$ to the right of $a_i$ in $A$
    EndIf
  EndFor
  identify which are the first ($f_i$) and the last ($l_i$) nest number
      where array $i$ is used
  Create $A_{sorted}$, a list of arrays in $A$, sorted in increasing
      order of first-usage loop
  $O \leftarrow \emptyset$
  While ($A_{sorted} \neq \emptyset$)
    Select $i$ at the beginning of $A_{sorted}$
    flag = 0
    For each array $j$ in $O$
      If ($f_i > l_j$) and ($S_i \leq S_j$) and ($B_{f_i}$ is not increased)
        Create a new array $k$ with memory used
          by array $j$
        $A \leftarrow A - i$
        Update $U_g$ of all arrays $g$ to the right of $i$ in $A$
        $S_k \leftarrow S_i$ ; $l_k \leftarrow l_i$ ; $f_k \leftarrow f_i$
        $O \leftarrow O + k$
        $S_j \leftarrow S_j - S_i$
        flag = 1
        Break
      EndIf
    EndFor
    If (flag == 0)
      $O \leftarrow O + i$
    EndIf
  EndWhile
End
```

Figure 7: An algorithm that improves memory energy consumption using array renaming.

Benchmark	Source	Data Size (MB)	Bank Configuration	Energy Consumption (mJ)
adi	Livermore	48.0	8× 8MB	3.38
aps	Perfect Club	41.5	8× 8MB	2.56
bmcm	Perfect Club	3.0	8× 0.5MB	1,040.34
btrix	Spec'92	47.3	8× 8MB	2.49
eflux	Perfect Club	33.6	16× 4MB	826.46
matvec	[1]	16.0	8× 8MB	675.86
mxm	Spec'92	48.0	8× 0.5MB	10,572.57
phods	[3]	33.0	8× 8MB	1,137.38
tomcatv	Spec'95	56.0	8× 8MB	119.78
vpenta	Spec'92	60.0	32× 2MB	2,026.66
wss	Perfect Club	3.0	8× 0.5MB	7,032.03

Figure 8: Benchmark codes and their important characteristics.

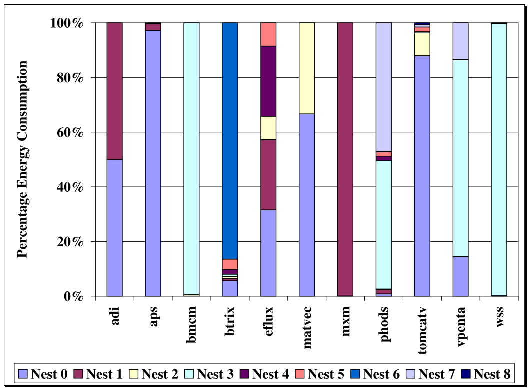


Figure 9: Energy consumption per loop nest.

benchmarks the energy consumption is dominated by a couple of nested loops. In fact, in eight of the eleven codes, there is a single dominating nest (energy-wise). Consequently, it seems reasonable to concentrate on these nests and try to optimize them. This approach is expected to improve the overall energy consumption, possibly at the expense of an increase in energy consumption of the other nested loops. In the remainder of this section, the dominating nest (i.e., the nest that consumes the most energy) in each code will be referred to as the *most costly nest.* If there are two or more nested loops that consume exactly the same amount of energy, we designate the first one (in the textual program order) as the most costly one.

4.1 Evaluation of Array Allocation

We first evaluated the effectiveness of our array allocation module. It should be noted that the layout placement of arrays across the banks affects the energy behavior of *all* program parts that use these arrays. Consequently, in deciding an appropriate layout for an array, we should ideally consider the entire application using a global approach. However, since a given array can be accessed by multiple nested loops in different fashions (e.g., column-wise versus row-wise), reaching a program-wide optimal layout appears to be very difficult. In fact, although not presented here due to lack of space, this optimal array allocation problem like many other optimal data placements problems [9] is NP-hard. Therefore, we focus our attention on deriving acceptable layouts that improve the energy behavior in most of the cases encountered in practice. The argument in the preceding paragraph indicates that it might be reasonable to focus only on the most costly nest (for a given code) and layout the arrays based on the requirements of that nest.

Figure 10 gives the normalized memory energy consumption for three versions of each code in our experimental suite. In the first version, all the memory banks are kept in the active mode all the time; in the second version, we only enable operating mode control as explained in [4]. In these two versions, the arrays are laid out across memory banks in the order that they are declared in the original program. In the third version, we active our array allocation module to take advantage of low-power operating modes better. These results reveal that by just enabling low-power mode control we achieve a 38.1% reduction in memory energy (over the first version). The most costly nest-centric array allocation brings a 6.1% additional improvement (on the average) and reduces the energy as compared to the second version in seven codes. In three other codes, we do not see any improvement, and in one code (`vpenta`), our allocation approach increases the energy consumption. The reason for this increase in `vpenta` is that in the default array layout (the declaration order) the two most costly arrays of this code hap-

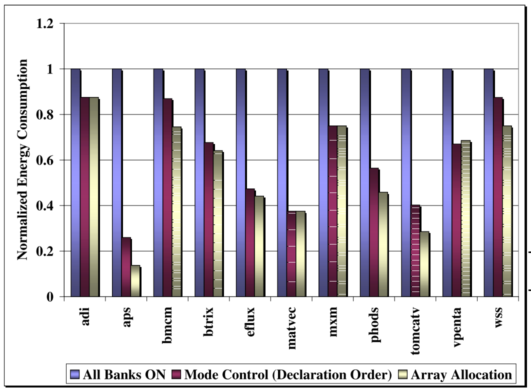


Figure 10: Effectiveness of the array allocation module.

Benchmark	Alternative Fission Strategies for the Most Costly Nest								
	#1	#2	#3	#4	#5	#6	#7	#8	#9
adi	47.0%	47.0%	61.2%						
aps	48.5%	48.5%	48.5%	48.5%					
eflux	47.0%	45.2%	43.3%	66.9%					
matvec	49.8%	33.2%	16.6%	58.2%					
tomcatv	49.5%								
vpenta	8.2%	23.5%	16.6%	24.9%	18.0%	16.6%	20.7%	8.2%	48.4%

Figure 11: Percentage energy improvements due to different loop fission alternatives.

Benchmark	Loop Nests								
	#0	#1	#2	#3	#4	#5	#6	#7	#8
adi	61.2%	0.0%							
apsi	48.5%	0.0%	-33.0%						
eflux	47.0%	19.9%	24.7%	19.9%	19.9%	25.0%			
matvec	58.2%	-48.1%	0.0%	0.0%					
tomcatv	49.5%	0.0%	0.0%	0.0%	-1.5%	0.0%	16.3%	0.0%	0.0%
vpenta	-33.2%	-39.8%	0.0%	48.4%	-33.3%	-33.3%	-0.3%	25.0%	

Figure 12: Percentage variations in energy of all nested loops in the benchmarks.

pen to go to the same memory bank, and when their use is over, this bank is put into a low-power mode. After the array allocation, however, these two arrays go to different banks, and this causes an increase in energy consumption. Nevertheless, array allocation appears to be successful in most of the cases. In the remainder of this section, when we say 'mode control', we mean 'mode control *plus* array allocation'.

4.2 Evaluation of Loop Fission

In optimizing the codes using loop fission, we again focussed only on the most costly nests and evaluated a limited set of loop fissioning alternatives (options) for these nests as mentioned earlier. These options represent a subset of all legal (data dependence-preserving) fissioning alternatives for the nest in question (see the algorithm in Figure 5 and the explanation in Section 3.2.1). Figure 11 shows the percentage decrease in energy (when the *entire application* is considered) brought about by each fission alternative over mode control. It should be noted that we consider only the codes that can benefit from loop fission and different most costly nests (of different codes) have different number of fission options (alternatives). The main reason that prevented the application of loop fission to other codes was the fact that the most costly nests in these codes contains only a single instruction; so, there was only one option, which is the original nest. We also observed that applying loop fission to the *second most costly nest* (instead of the most costly nest) in general increased the overall energy consumption as the array layouts suggested by the second most costly nest are usually not suitable for the most costly nest.

Consequently, we selected the most effective alternative for each benchmark and applied it. The average improvement (%) over all benchmarks (that are amenable to loop fission) was around 55.5%. Since the array allocation module considers only the most costly nest, there might be a negative impact (energy-wise) on other nests if the best layout strategy for the most costly nest is not suitable for the other nests. Figure 12 shows the energy impact of this most costly nest-centric optimization on *other nests*. We observe that different benchmarks behave quite differently. For example, in `eflux`, all the other nests (in addition to the most costly nest) show energy improvement. For `vpenta`, on the other hand, the second nest experience a 39.8% increase in energy consumption. Overall, the nests *other than the most costly ones* show only a 2.8% increase in energy. Thus, we conclude that focusing only on the most costly nest and performing an array allocation based on that works well with loop fission.

4.3 Evaluation of Loop Splitting

In this subsection, we present the energy results obtained through application of loop splitting. As in the case of loop fissioning, our approach focuses on the most costly nest and the array allocation module performs bank allocation based on that. Overall, we found that seven of the eleven application in our experimental suite take advantage of loop splitting. Figure 13 shows the normalized energy consumption due to loop splitting. On the average, with respect to a naive alternative where all the memory banks are kept in the active mode for all the time, this optimization brings a 61.5% reduction in energy. Further, with respect to an approach that performs only mode control and array allocation, it brings a 42.8% energy improvement on the average.

Note that we applied loop splitting only to the nested loops that contain at least one array reference of the form `a[`i_k`][*][*]..[*]` where i_k is a loop in the nest and `[*]` denotes to any subscript expression. Using loop splitting to handle the nests that do not contain such references is in our future agenda.

4.4 Evaluation of Array Renaming

Figure 14 presents the normalized energy consumption due to array renaming. We found that only two benchmarks (`btrix` and `phods`) take advantage of this optimization. However, we believe that a more sophisticated (inter-procedural [19]) analysis can find more opportunities for this optimization in large codes that use many temporary arrays.

Figure 15 summarizes the energy impact of our optimizations in each code. Apart from individual optimizations, it also gives the combined memory energy effect of the optimizations. From these results, we observe that for many benchmark codes, some combination of these three optimizations brings further benefits over individual optimizations. For instance, in the `adi` code, using loop fission with loop splitting leads to more than 50% energy saving. The benchmark `vpenta` requires a special attention. As we have seen earlier, this was the only benchmark that could not get any benefit from array allocation. However, it shows around 70% improvement with loop splitting plus loop fission.

4.5 Sensitivity to the Bank Configuration

In this subsection, we measure the variations in memory energy as one varies the number of memory banks keeping the aggregate

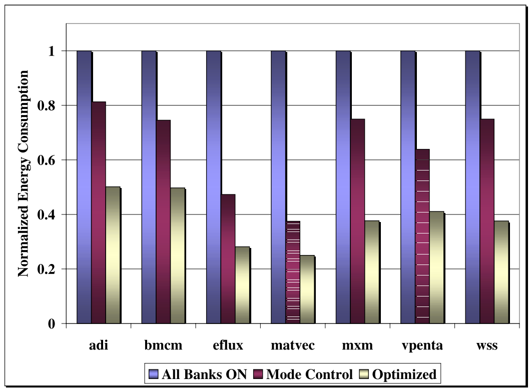


Figure 13: Energy impact of loop splitting.

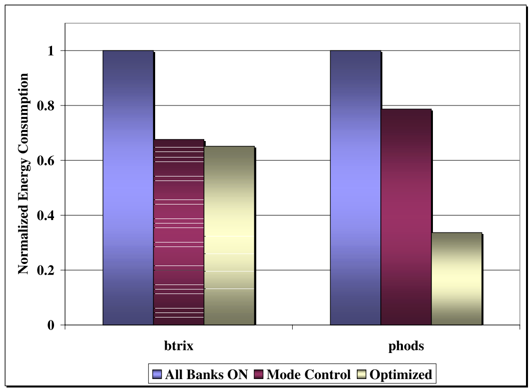


Figure 14: Energy impact of array renaming.

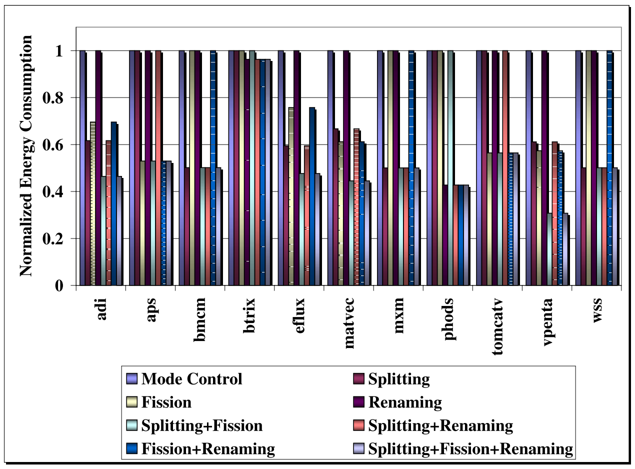


Figure 15: Normalized energy consumptions.

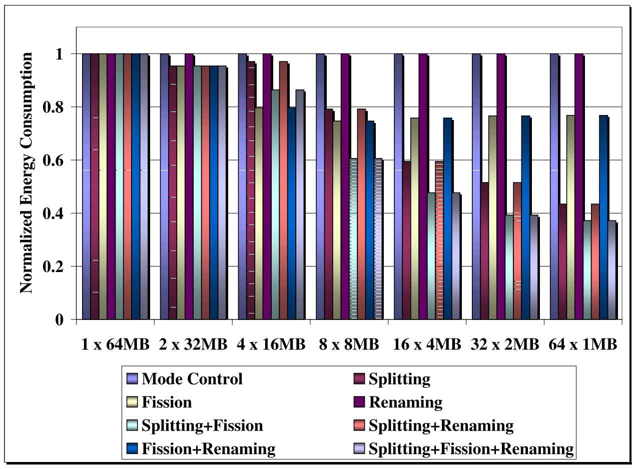


Figure 16: Sensitivity to the number of banks for a fixed memory capacity.

memory size fixed. For this purpose, we use the `eflux` benchmark. We modulate the number of banks from 1 to 64 keeping the total memory size fixed at 64 MB (Recall that the default bank configuration for this benchmark was 16 $\times$ 4 MB). The results given in Figure 16 show that increasing the number of memory banks in general increases the effectiveness of our approach. This is due to the fact that more banks gives more opportunity to the compiler to turn off larger portions of the memory address space. For example, with the configuration 8 $\times$ 8 MB, we save a 40% memory energy using all three optimizations discussed in this paper. Increasing the number of banks from 8 to 32 brings an additional 20% energy saving.

4.6 Influence of Cache

We also performed experiments with two cache configurations (a 4K, 2-way set-associative cache and a 4K, 4-way set-associative cache). The results are given in Figures 17 and 18 indicate that the results obtained using a system with cache are comparable to those obtained through a cacheless system. Note that a cache configuration can influence energy savings due to memory power mode control in two ways. First, it modifies the number of accesses to the main memory. Typically, one would expect a smaller number of references to memory with larger caches and higher associativities (That is the reason why when all the memory banks are active (ON) there is a difference between a cacheless system and a system with cache). This would imply lesser potential for savings due to power mode control. However, the cache also modifies the inter-access times (the time between memory references) for the memory banks. With fewer cache misses, one would anticipate the use of a lower power mode of operation for the memory banks for a longer time or a transition to a more energy-saving low power mode. Note also that all the numbers given in Figures 17 and 18 are normalized energy values using no cache (resp. a cache) with respect to the case where all the memory banks are active all the time without (resp. with) cache. A full-understanding of the energy behavior of compiler-directed optimizations under a partitioned memory system with cache requires further investigation and experimentation.

5 Related Work

Most of the energy-oriented software studies so far are from the design automation/high-level synthesis community and target specifically embedded applications. Memory optimizations for embedded

Benchmark	Mode Control		Splitting		Fission		Reuse		Fission+Splitting	
Name	Cache	No cache	Cache	No cache	Cache	No cache	Cache	No cache	Cache	No cache
`adi`	84%	83%	58%	50%	76%	56%			52%	37%
`aps`	26%	25%			44%	13%				
`bmcm`	74%	74%	8%	49%						
`btrix`	73%	67%					73%	64%		
`eflux`	43%	47%	26%	27%	34%	35%			24%	22%
`matvec`	37%	37%	24%	24%	25%	22%			13%	16%
`mxm`	74%	74%	37%	37%						
`phods`	52%	78%					32%	33%		
`tomcatv`	53%	40%			43%	22%				
`vpenta`	72%	67%	44%	40%	71%	38%			36%	20%
`wss`	74%	74%	37%	37%						

Figure 17: The energy consumption normalized to the energy spent with all banks ON, with and without cache (4K, 2-way), respectively.

Benchmark	Mode Control		Splitting		Fission		Reuse		Fission+Splitting	
Name	Cache	No cache	Cache	No cache	Cache	No cache	Cache	No cache	Cache	No cache
`adi`	86%	83%	60%	50%	83%	56%			58%	37%
`aps`	26%	25%			43%	13%				
`bmcm`	74%	74%	10%	49%						
`btrix`	72%	67%					71%	64%		
`eflux`	43%	47%	26%	27%	34%	35%			19%	22%
`matvec`	37%	37%	25%	24%	24%	22%			12%	16%
`mxm`	74%	74%	37%	37%						
`phods`	59%	78%					12%	33%		
`tomcatv`	59%	40%			52%	22%				
`vpenta`	72%	67%	44%	40%	71%	38%			36%	20%
`wss`	74%	74%	37%	37%						

Figure 18: The energy consumption normalized to the energy spent with all banks ON, with and without cache (4K, 4-way), respectively.

systems have been partly addressed by Shiue and Chakrabarti [14] who presented a memory exploration strategy based on three metrics, namely, processor, cycles, cache size, and energy consumption. The IMEC group [2] was among the first to work on applying loop transformations to address power dissipation in data-dominated embedded applications.

It is also possible to affect energy consumption using low-level compiler techniques. Towards this goal, Su, Tsui, and Despain [15] proposed a technique that combines Gray code addressing and instruction scheduling. Based on basic-block list scheduling, their approach uses a greedy algorithm to re-order the instructions to minimize power consumption. Tiwari, Malik, and Wolfe [17] proposed another scheduling technique that also tries to select instructions more judiciously to minimize power. Toburen et al. [18] presented a method of instruction scheduling that limits the number of instructions that can be scheduled in a given cycle depending on the power constraints. Note that all these optimizations focus only on the datapath energy.

6 Conclusions

The energy optimization problem is very complex, as there are many inter-related hardware and software issues that must be addressed. This paper addresses the problem of automatically minimizing energy consumption of a partitioned off-chip memory system with multiple banks. Our compiler-based approach consists of an array allocation algorithm and an optimization module that uses both loop and data transformations. The optimizations discussed in this paper take advantage of multi-bank nature of the memory architecture and low-power operating modes. While these optimizations, themselves, are not new, their application to the problem of memory energy reduction through the exploitation of low-power operating modes is novel. This paper also gives experimental data showing that it is possible to obtain energy savings up to 86% over a scheme that keeps all the memory banks in the active (fully-operational) state all the time, and up to 70% over a scheme that utilizes low-power operating modes without doing any loop and data optimizations. Our future work includes extending our optimization module to include more loop and data optimizations. Of particular interest is array unification which interleaves multiple arrays into a single array to enhance spatial locality of data accesses. We also plan to perform experiments using a larger set of applications and different re-synchronization costs.

References

[1] R. Bhargava, L. K. John, B. L. Evans, and R.Radhakrishnan. Evaluating MMX technology using DSP and multimedia applications. In Proc. *the IEEE Symposium on Microarchitecture*, Dallas, Texas, pp.37–46, Dec.1998.

[2] F. Cathoor, S. Wuytack, E. De Greef, F. Fransen L. Nachtergaele, and H. De Man. System-level transformations for low-power data transfer and storage. In *Low-Power CMOS Design,* R. Chandrakasan and R. Brodersen, Eds. IEEE Press, Piscataway, NJ.

[3] L-G. Chen, W-T. Chen, Y-S. Jehng, and C-T. Church. An efficient parallel motion estimation algorithm for digital image processing. *IEEE Trans. Circuits and Systems for Video Tech,* 1(4):378–385, December 1991.

[4] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Memory energy management using software and hardware directed power mode control. *Technical Report CSE-00-004,* Dept. of Computer Science and Engineering, The Pennsylvania State University, April 2000.

[5] E. D. Granston and H. A. G. Wijshoff. Managing pages in shared virtual memory systems: getting the compiler into the game. In Proc. *the 1993 international conference on Supercomputing,* July 1993, Tokyo, Japan, pages 11–20.

[6] W-M. W. Hwu. Embedded microprocessor comparison (ECE412 Class Notes). http://www.crhc.uiuc.edu/IMPACT/ece412/public_html/Notes/412_lec1/ppframe.htm.

[7] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In Proc. *International Symposium on Microarchitecture,* Dallas, TX, December, 1998.

[8] S. Y. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors.* Ph.D. Thesis, Dept. of EECS, MIT, Cambridge, Massachusetts, June 1996.

[9] M. Mace. *Memory Storage Patterns in Parallel Processing*, Kluwer Academic Publishers, Boston, MA, 1987.

[10] T. C. Mowry. Tolerating latency through software-controlled data prefetching. *Ph.D. Thesis,* Stanford University, Computer Systems Laboratory, March 1994.

[11] S. S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.

[12] M. O'Boyle and P. Knijnenburg. Integrating loop and data transformations for global optimisation. In Proc. *International Conference on Parallel Architectures and Compilation Techniques*, October 1998, Paris, France.

[13] 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., May 1999.

[14] W-T. Shiue and C. Chakrabarti, Memory exploration for low power, embedded systems, *CLPE-TR-9-1999-20, Technical Report,* Center for Low Power Electronics, Arizona State University, 1999.

[15] C.-L. Su and A. M. Despain. Cache design trade-offs for power and performance optimization: A case study, In Proc. *International Symposium on Low Power Electronics and Design*, pp. 63–68, 1995.

[16] D. Tennenhouse. *Pro-Active Computing*. http://www.darpa.mil/ito/Proceedings/DARPATech99_html.

[17] V. Tiwari and M. T-C. Lee. Power analysis of a 32-bit embedded microcontroller. In Proc. *the Asia and South Pacific Design Automation Conference,* pp. 141–148, 1995.

[18] M. C. Toburen, T. M. Conte, and M. Reilly. Instruction scheduling for low power dissipation in high performance processors. In Proc. *the Power Driven Micro-architecture Workshop in conjunction with the ISCA'98.* Barcelona, Spain, June 1998.

[19] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In Proc. *ACM SIGPLAN'86 Symposium on Compiler Construction,* Palo Alto, CA, pp. 175–185, June 1986.

[20] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices,* 29(12):31–37, December 1994.

[21] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison Wesley, CA, 1996.