Improving Memory Energy Using Access Pattern Classification [*]

Mahmut Kandemir
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802
kandemir@cse.psu.edu

Ugur Sezer
ECE Department
University of Wisconsin
Madison, WI 53706
sezer@ece.wisc.edu

Victor Delaluz
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802
delaluzp@cse.psu.edu

ABSTRACT

In this paper, we propose a data-driven strategy to optimize the memory energy consumption in a banked memory system. Our compiler-based strategy modifies the original execution order of loop iterations in array-dominated applications to increase the length of the time period(s) in which memory banks are idle (i.e., not accessed by any loop iteration). To achieve this, it first classifies loop iterations according to their bank access patterns and then, with the help of a polyhedral tool, tries to bring the iterations with similar bank access patterns close together. Increasing the idle periods of memory banks brings two major benefits; first, it allows us to place more memory banks into low-power operating modes, and second, it enables us to use a more aggressive (i.e., more energy saving) operating mode for a given bank. Our strategy has been evaluated using seven array-dominated applications on both a cacheless system and a system with cache memory. Our results indicate that the strategy is very successful in reducing the memory system energy, and improves the memory energy by as much as 34% on the average.

1. Introduction and Motivation

Energy consumption is an important consideration in embedded system design as reducing it helps maximize battery life and reduce heat dissipation [9]. The increasing role of energy consumption demands that the energy constraints should be taken into account early in the system design process, along with other metrics such as performance and form factor.

Memory components of an embedded system are known to consume a large percentage of the overall system energy [2, 6]. This is particularly true for systems designed to run image and video processing applications in which large multi-dimensional arrays of signals are manipulated using nested loops. These data-intensive applications might put a great pressure on the memory subsystem and make the memory performance the primary factor shaping the runtime energy behavior of the system. For a given memory system, size (capacity) has a large impact on the energy consumption. Typically, large memory components (e.g., banks) consume more *energy per access* than small memory components, mainly due to longer bitlines and wordlines. Consequently, employing several small memory banks instead of a single large memory bank might reduce the per access energy cost. Moreover, unused memory banks can be turned off to further increase energy savings. Turning off a memory bank in this context means placing it into a *low-power operating mode* (an *energy-saving mode*) in which the bank consumes much less energy than it would have if it remained in the fully-active mode. While such a strategy can, in general, improve the memory energy consumption, it is possible to obtain even larger savings by re-ordering data elements and/or loop iteration accesses to take better advantage of the multi-bank nature of the memory subsystem.

This paper presents a compiler-based strategy that modifies the original execution order of loop iterations in array-dominated applications to increase the length of the time period(s) in which memory banks are idle (i.e., not accessed by any loop iteration). Such an optimization can bring two main benefits. First, as a result of longer idle period, we can put a bank into a low-power operating mode (while, in the original case, it may not be possible to do so). Second, even in the original case it was possible to use a low-power mode, after the transformation we might be able to use a more aggressive (i.e., more energy-saving) operating mode, and thus obtain larger savings. Major contributions of this paper can be summarized as follows:

- We present a data-centric access pattern classification technique using which an optimizing compiler can classify loop iterations based on their memory bank access patterns.
- We discuss a transformation technique that modifies the original loop accesses to implement the classification determined by the compiler. The transformation technique operates with both fully-parallel loops and loops with data dependences.

Our experimental results indicate that our approach is very successful in optimizing the memory system energy, and reduces the memory energy by as much as 34% on the average.

We proceed with a brief review of multi-bank memory system and low-power operating modes. Next, in Section 3, we present a loop iteration classification strategy based on bank access patterns. We discuss an access pattern optimization strategy in Section 4. Preliminary experimental results are given in Section 5. Finally, we summarize the main points of this work in Section 6.

2. Energy Consumption in Memory Banks

We can think of a memory system as a group of banks each of which can be controlled individually and be put in a low-power operating mode (energy-saving mode) when it is not in active use. Typically, a number of low-power operating modes

[*]This work is supported by the NSF Career Award 0093082.

exist, and selection of a specific mode (for a given duration of idleness) involves a tradeoff between performance and energy consumption. More specifically, each operating mode can be characterized using two parameters: *per cycle energy consumption* and *re-activation (re-synchronization) cost,* the latter of which is the time (in cycles) it takes to bring the bank back to the active (fully-operational mode). Typically, the more aggressive the operating mode (in saving energy), the higher the re-activation cost. Therefore, the low-power mode to be used should be chosen with care. An important parameter in choosing the most suitable mode is the estimated duration of idleness. If the idleness duration is too short, it may not be a good idea to place a bank into an aggressive energy-saving mode. It is important to note that many array-dominated codes from image and video processing applications allow accurate estimation of bank idleness, thanks to frequent occurrence of nested loop structures and compile-time determinable mappings of array elements to memory banks.

In Active (normal operation) mode, the memory bank is ready to immediately service a memory request without any delay. In the low-power modes (Standby, Nap, and PowerDown), energy consumption can be reduced by shutting-off increasingly larger parts of the bank. The major parts of a memory bank are the clock generation circuitry, row address/control decode circuitry, column address/control decode circuitry, control registers and power mode control circuitry, together with the memory (DRAM) core consisting of the precharge logic, memory cells, and sense amplifiers. The operating modes used in this study are very similar to those proposed in the Direct Rambus architecture for mobile PCs [1]. While in a low-power operating mode, a memory request (read or write) causes the bank to transition to the Active mode to service the request. Note that at a given time different memory banks can be in different low-power operating modes.

One of the objectives of the low-power operating mode management is selecting the most suitable operating mode for a given duration of idleness. The compiler needs to detect idle periods for each bank and transition each idle bank into a low-power mode. A naive way of doing this is to select an operating mode arbitrarily and keep the bank in this mode until it needs to be accessed. Obviously, such a strategy pays the re-activation (re-synchronization) cost which we want to avoid. Instead, we use a bank pre-activation strategy that eliminates the potential performance penalty associated with employing low-power operating modes. Given that we have a menu of low-power modes to transition into, the compiler can evaluate all possible choices (low-power modes) based on the operating mode energy, corresponding re-activation costs, and the length of the idle period to select the best choice. A discussion of compiler-directed operating mode selection can be found in [3].

Note that maximizing the duration in which a memory bank is idle is beneficial (from an energy angle) as the bank in question can be placed into a more energy-saving (most aggressive) operating mode. This can be achieved using smart array layout strategies that place array elements with similar life-patterns into the same banks, or by re-ordering the computation to isolate the data accessed (within a time frame) into a small number of banks. This paper takes the latter approach and presents an automatic (compiler-based) technique that first classifies the iterations of a given loop according to their bank access characteristics, and then exploits this classification to restructure the execution order of loop iterations.

3. Data-Centric Access Pattern Classification

In this section, we present our computation model and discuss the relationship between loop iterations and memory bank accesses. We assume that the compiler is in full control of physical address management and there exists no virtual memory support. Consequently, it is possible to calculate at compile-time the physical memory location accessed for a given array reference and loop iteration.

Each execution of loop body is represented using an iteration vector $\bar{I} = [i_1, i_2, ..., i_n]^T$, where n is the number of loops. The loop bounds can be described using a system of affine inequalities of the form $H\bar{I} \le \bar{h}$, where H is a $k \times n$ matrix and $\bar{h}$ a k-dimensional vector. Both H and $\bar{h}$ have constant entries and together they define a polyhedron (iteration space) which contains all loop iterations [11]. The storage form of an array can also be viewed as a (rectilinear) polyhedron. Each array element can be identified by its index $\bar{J} = [j_1, j_2, ..., j_m]^T$, where m is the dimensionality of the array. Then, the polyhedron (index space) which defines the possible indices for a given array can be written as $S\bar{J} \le \bar{s}$, where S and $\bar{s}$ are a $2m \times m$ matrix and a $2m$-dimensional vector, respectively.

The subscript expressions of a given reference to an m-dimensional array define an affine access function (F) from iteration space to index space; that is, $F(\bar{I}) = L\bar{I} + \bar{l}$. In this formulation, L is an $m \times n$ matrix (called access or reference matrix [11]) and l is a m-dimensional constant vector (called offset vector).

Assuming a row-major storage form for multi-dimensional arrays, the address of the array element $U[j_1, j_2, ..., j_m]$ can be computed as

$$addr(U[j_1, j_2, ..., j_m]) = B_u + j_1(|S_2||S_3|...|S_m|) +$$

$$j_2(|S_3|...|S_m|) + ... + j_{m-1}|S_m| + j_m$$

under the assumptions that the lower bound for all index positions (subscript positions) is 1 and that $|S_p|$ is the extent (the number of elements) in the p^{th} dimension and that B_u is the base address for array U.

Without loss of generality, let us assume the existence of K memory banks of equal sizes ($size$). Under this assumption, a *bank mapping function* (BMF) G maps a given address into a memory bank and can be written as $G(addr(U[j_1, j_2, ..., j_m])) = addr(U[j_1, j_2, ..., j_m])/size$ (where the symbol / denotes integer division). Consequently, given an array element $U[F(\bar{I})]$ (accessed by iteration vector $\bar{I}$), $G(addr(U[F(\bar{I})]))$ gives the bank that it is mapped to.

An iteration vector $\bar{I}$ is said to access bank i $(1 \le i \le K)$ if at least one of the array elements it touches is mapped into bank i. In mathematical terms, we say $bank(i, \bar{I})$ is *true* if and only if there exists at least one array U and an access function $F(\bar{I})$ in the nest such that $G(addr(U[F(\bar{I})])) = i$; otherwise, we say that $bank(i, \bar{I})$ is *false*.

Note that we can *classify* the iterations of a given nested loop according to the (subset of the) banks they access (i.e., their bank access patterns). As an example, let us focus on a scenario where a nested loop (possibly imperfectly nested) accesses arrays stored in a memory system that consists of two banks. We can divide the iterations in this nest into three groups:

$$\{I\}_{1\bar{2}} = \{\bar{I}|\ bank(1, \bar{I}) = true \text{ and } bank(2, \bar{I}) = false\};$$

$$\{I\}_{\bar{1}2} = \{\bar{I}|\ bank(1, \bar{I}) = false \text{ and } bank(2, \bar{I}) = true\};$$

and,

$$\{I\}_{12} = \{\bar{I} |\ bank(1, \bar{I}) = true \text{ and } bank(2, \bar{I}) = true\}.$$

Informally, the first group ($\{I\}_{1\bar{2}}$) corresponds to the iterations that access only the first bank whilst the second group ($\{I\}_{\bar{1}2}$) corresponds to iterations that access only the second bank. The third group ($\{I\}_{12}$), on the other hand, consists of the iterations that access both the banks. Note that the original iteration space, $\{I\}_{all}$, is $\{I\}_{1\bar{2}} \cup \{I\}_{\bar{1}2} \cup \{I\}_{12}$ (assuming that each loop iteration accesses at least one bank). A subscript notation such as $1\bar{2}$ indicates that the iterations in the class subscripted using this access only the first bank. Other subscript notations can be interpreted in a similar fashion.

In general, assuming K memory banks, a given iteration space $\{I\}_{all}$ can be divided into $2^K - 1$ disjoint groups (called *classes* in the rest of this paper): $\{I\}_{1\bar{2}\bar{3}...\bar{K}}$, $\{I\}_{\bar{1}2\bar{3}...\bar{K}}$, ..., $\{I\}_{\bar{1}\bar{2}\bar{3}...K}$, $\{I\}_{12\bar{3}...\bar{K}}$, $\{I\}_{1\bar{2}3...\bar{K}}$, ..., $\{I\}_{123...\bar{K}}$,..., $\{I\}_{123...K}$. This grouping is called *access pattern classification* (for a given, possibly imperfectly-nested, loop nest) and forms the basis of our optimization method presented in the next section.

4. Access Pattern Optimization

In this section, we show how to optimize the memory accesses based on the classification given above. The main idea is to re-order the loop iterations such that all the iterations that belong to the same class are executed one after another (successively).

4.1 Optimizing Dependence-Free Accesses

Let us first assume that the nested loop we want to optimize is fully-parallel (i.e., does not have any loop-carried data dependence). In this case, the iterations can be executed in any order. Consider, for example, the following abstract nested loop assuming two banks:

```
for each $\bar{I} \in \{I\}_{all}$
    $\{U_1[F_{u1}(\bar{I})], U_2[F_{u2}(\bar{I})], U_3[F_{ur}(\bar{I})], ..., U_r[F_{ur}(\bar{I})]\}$
```

U_1, U_2, ..., and U_r in this loop are different arrays, and F_{uk} is the access function (access matrix plus offset vector) for U_k (it is straightforward to extend the idea to the case where there exist multiple references to the same array). Let $\{I\}_{1\bar{2}}$, $\{I\}_{\bar{1}2}$, and $\{I\}_{12}$ be the classes defined as above. Then, we can transform this loop nest into:

```
for each $\bar{I} \in \{I\}_{1\bar{2}}$
    $\{U_1[F_{u1}(\bar{I})], U_2[F_{u2}(\bar{I})], U_3[F_{ur}(\bar{I})], ..., U_r[F_{ur}(\bar{I})]\}$
for each $\bar{I} \in \{I\}_{\bar{1}2}$
    $\{U_1[F_{u1}(\bar{I})], U_2[F_{u2}(\bar{I})], U_3[F_{ur}(\bar{I})], ..., U_r[F_{ur}(\bar{I})]\}$
for each $\bar{I} \in \{I\}_{12}$
    $\{U_1[F_{u1}(\bar{I})], U_2[F_{u2}(\bar{I})], U_3[F_{ur}(\bar{I})], ..., U_r[F_{ur}(\bar{I})]\}$
```

Each nested loop in this code executes the iterations from a single class and such a division of the iteration space (into classes) is called *classification*. Note that it is not necessary that these three nested loops should be executed in this order. In fact, as compared to the original code above, any execution order of these new nests (as long as the iterations in a class are executed successively) will usually bring an improvement. To see this impact in a concrete example, consider the following C-like code that consists of a two-dimensional imperfectly-nested loop nest:

```
for($t = 1; t \le T; t++$)
  {
    for($i = 1; i \le 10; i++$)
      $k1 += W[30 - i] - 1$;
    for($j = 1; j \le 10; j++$)
      $k2 += (U[j] + V[j])/2$;
    for($k = 1; k \le 10; k++$)
      $k3 += V[k + 10] + 1$;
    for($l = 1; l \le 10; l++$)
      $k4 += W[l] + V[16]$;
    for($m = 1; m \le 10; m++$)
      $k5 += V[15 - m] + 1$;
    .........
  }
```

assuming the following array mapping (to a memory system of two banks each holding, for illustrative purposes, 45 array elements):

$$\text{Bank 1:}\{U[a] | 1 \le a \le 30\} + \{V[a] | 1 \le a \le 15\},$$

$$\text{Bank 2:}\{V[a] | 16 \le a \le 30\} + \{W[a] | 1 \le a \le 30\}.$$

We also assume here that the scalar variables $k1$ through $k5$ are stored in registers. Figure 1(a) shows the bank access pattern for this code. We clearly identify five different regions corresponding to five inner loops and see that four opportunities exist for putting a memory bank into low-power mode: one in the first region (for bank 1), one in the second region (for bank 2), one in the fourth region (for bank 1), and one in the last region (for bank 2).

Let us now consider the following equivalent code where iterations of the nested loop are classified according to their bank access patterns:

```
for($t = 1; t \le T; t++$)
  {
        $\{I\}_{\bar{1}2}$ :
    for($i = 1; i \le 10; i++$)
      $k1 += W[30 - i] - 1$;
    for($l = 1; l \le 10; l++$)
      $k4 += W[l] + V[16]$
        $\{I\}_{12}$ :
    for($k = 1; k \le 10; k++$)
      $k3 += V[k + 10] + 1$;
        $\{I\}_{1\bar{2}}$ :
    for($j = 1; j \le 10; j++$)
      $k2 += (U[j] + V[j])/2$;
    for($m = 1; m \le 10; m++$)
      $k5 += V[15 - m] + 1$;
    .........
  }
```

The new bank access pattern is illustrated in Figure 1(b). We can label this access pattern as [$\{I\}_{\bar{1}2}$; $\{I\}_{12}$; $\{I\}_{1\bar{2}}$]. Figures 1(c-d), on the other hand, depict two alternate (optimized) bank access patterns corresponding to [$\{I\}_{\bar{1}2}$; $\{I\}_{1\bar{2}}$; $\{I\}_{12}$] and [$\{I\}_{1\bar{2}}$; $\{I\}_{\bar{1}2}$;$\{I\}_{12}$], respectively. It should be noted that iteration re-ordering clusters the idle regions in both the banks. Note also that these three optimized access patterns (shown in Figures 1(b), (c), and (d)) are only representative and that there are other access patterns (for this example) which cluster the idle regions.

Note that while our approach to clustering array accesses is similar to the data-centric tiling technique proposed by Kodukula et al. [8], there are three important differences. First, our approach deals with physical addresses rather than virtual

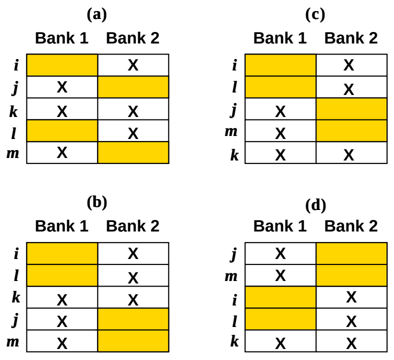

Figure 1: Different access patterns to a memory system of two banks. A 'X' indicates that the corresponding region is in use.

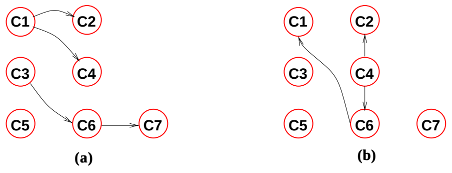

Figure 2: Two different class-level dependence graphs (CLDG).

addresses which makes expected runtime benefits more realizable. Second, instead of clustering the accesses to array blocks, we cluster accesses to memory banks. Third, our strategy clusters loop iterations for low power and is not concerned whether the iterations executed together (within a block) will have cache locality.

4.2 Optimizing Accesses with Dependences

So far, we have assumed that once the classes are determined, they can be executed in any order. This assumption holds true as long as there are no data dependences between iterations belonging to different classes. In this section, we address the problem when the loop nest to be optimized exhibits loop-carried dependences.

A class $\{I\}$ is said to be dependent on another class $\{I'\}$ if there exists two iterations, $\bar{I} \in \{I\}$ and $\bar{I}' \in \{I'\}$, such that there is a dependence from $\bar{I}'$ to $\bar{I}$. We express this relationship saying $dep(\{I'\} \rightarrow \{I\})$ is *true* (or, $\{I\}$ is *class-dependent* on $\{I'\}$). Otherwise, we say that $dep(\{I'\} \rightarrow \{I\})$ is *false*. Then, given a classification, we can define the *class-level dependence graph,* $CLDG(V, E)$, as a directed-graph where each node $v \in V$ denotes a class and each edge $e \in E$ indicates a data dependence from one class to another. That is, there is a directed edge from $\{I'\}$ to $\{I\}$ iff $dep(\{I'\} \rightarrow \{I\})$ holds true. Note that class dependences prevent the compiler from executing the classes in any order; instead, these dependences should be satisfied when selecting an execution order for classes.

Let us assume that $\{I\}_1$, $\{I\}_2$, $\{I\}_3$, ..., $\{I\}_{2^K-2}$, $\{I\}_{2^K-1}$ is the classification that we would like to optimize. As a first step, we re-number each class using a K-bit *binary number* as follows. Each bank is assigned a bit position (in the binary number) starting from the first bank. If a given class accesses bank i, then the i^{th} bit of the number associated with the class is set to 1; otherwise, it is set to 0. For example, in a four bank memory system, if a class accesses only the first and the fourth banks, it will have '1001' as its number. We express this fact by writing this class as $\{I\}_{1001}$. The binary number associated with a class (e.g., 1001 in this example) is termed as the *class number*.

The problem of optimizing bank accesses can be defined as one of determining a suitable traversal order of the nodes in the CLDG. When a node is visited, a nested loop which enumerates only the iterations in that class is constructed and inserted in the code. A traversal order is *valid* (or *legal*) if it respects all the data dependences. Note that the intra-class dependences are satisfied if we execute the iterations in a given class in their original order with respect to each other. Inter-class dependences (class-level dependences), on the other hand, are satisfied by visiting a node in the CLDG only after all the nodes that it is dependent on have been visited.

Note that a given CLDG can have several valid traversals, and selecting the most suitable one is the primary factor that determines the memory system energy consumption (through low-power mode selection). The following observation guides us in selecting a suitable order:

> *If $\{I\}_a$ and $\{I\}_b$ are the two classes that are successively visited, the variation in bank access (activation) and bank idleness (deactivation) patterns in going from $\{I\}_a$ to $\{I\}_b$ is a function of the Hamming distance between a and b.*

For instance, if we traverse $\{I\}_{1001}$ and $\{I\}_{1010}$ one after another, the first two banks preserve their states (between these traversals), whereas the remaining two banks change their states (more specifically, the third bank is activated –corresponding to a 0 to 1 transition in the associated bit– in going from the first class to the second, and the fourth bank is deactivated –corresponding to a 1 to 0 transition).

We claim that minimizing the Hamming distance between the numbers of classes that are visited successively is useful in reducing the energy consumption. In other words, for a given memory bank, in going from one class to another, it is better to keep its state the same (active or idle) as much as possible. This is because if the first state is 0 and the second is also 0, the bank will have a long idle period (which is good from an energy viewpoint); and similarly, if the both states in question are 1, this means that the active periods are clustered together, so hopefully, we will also have clustered idle periods for the bank in question (later when we visit the remaining classes). Note that then the problem of optimizing memory energy consumption becomes one of scheduling a group of nodes taking into account some constraints (inter-class dependences) to minimize (optimize) some objective function (minimizing the Hamming distance between the numbers of successively visited classes). This problem is a constrained scheduling problem and is known to be NP-hard [4]. Consequently, we propose a greedy heuristic similar to the list scheduling algorithm used in low-level (back-end) compilation to schedule the instructions in a basic block. Informally, in each step, our algorithm selects a class to schedule (visit) such that the Hamming distance between the (class) number of this class and that of the most recently scheduled one is minimum (among all alternatives). After scheduling a class, it determines all classes that can be scheduled in the next step, and evaluates their Hamming distance with respect to the scheduled class.

We can evaluate a given traversal order using two different metrics, both of which are directly related to the bank energy consumptions. The first metric is the sum of the number of bit

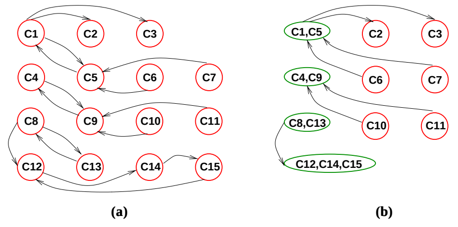

Figure 3: (a) A cyclic CLDG. (b) Transformed version of (a).

transitions between successive class numbers (i.e, the cumulative Hamming distance). Let $|a_i - a_j|$ represent the Hamming distance between the class numbers a_i and a_j. Assuming a traversal order of $\{I\}_{a_1}$, $\{I\}_{a_2}$, $\{I\}_{a_3}$, ..., $\{I\}_{a_{2^K-1}}$, the cumulative Hamming distance is defined as

$$\sum_{j=1}^{2^K-2} |a_j - a_{j+1}|.$$

Note that the scheduling technique discussed above tries to reduce this cumulative sum. The second metric is the maximum number of consecutive 0s in the bit positions of each bank. Note that, in a K-bank memory system, there might be, at the most, $2^{(K-1)}$ consecutive 0s.

For an example, let us consider the CLDG in Figure 2(a) for a three-bank system, assuming that the class numbers for `C1`, `C2`, `C3`, `C4`, `C5`, `C6`, and `C7` are 100, 010, 011, 101, 111, 001, and 110, respectively. An unoptimized scheme can traverse the classes in the order `C1`, `C2`, `C3`, `C4`, `C5`, `C6`, and `C7`, leading to a cumulative Hamming distance of eleven. We also see that the maximum number of consecutive 0s (as a result of this class traversal pattern) in this case is 2, 1, and 2, for bank 1, bank 2, and bank 3, respectively. However, if we use our approach, we obtain the traversal order of `C1`, `C4`, `C5`, `C3`, `C6`, `C2`, and `C7`. This new order results in a cumulative Hamming distance of 7. Also, it achieves the maximum possible consecutive 0s for the first bank (for the other banks, it achieves two consecutive 0s), a definite improvement over the unoptimized traversal. As another example, consider the CLDG given in Figure 2(b). Assuming the same class numbers used in Figure 2(a) and the same memory bank architecture, our approach selects the order of `C3`, `C5`, `C4`, `C6`, `C1`, `C7`, and `C2`, achieving a cumulative Hamming distance of seven (which is only one more from the optimal which is six) and obtaining the maximum number of consecutive 0s for two out of three banks.

In some cases, the CLDG may contain cycles which prevent a legal traversal. We call these types of graphs *cyclic CLDGs*. In order to schedule them, we need to apply some node transformations and eliminate the cycles. An example cyclic CLDG is illustrated in Figure 3(a) for a four-bank memory system. We use two types of transformations to handle these graphs, namely, node merging and node splitting, details of which are omitted due to lack of space. Figure 3(b) shows the transformed version of Figure 3(a) using node merging.

5. Experimental Results

We used the Omega library [7] to transform the programs' access patterns into bank-efficient ones. The Omega library is a polyhedral tool that allows enumeration of bounded polyhedrons and helps us to determine the iteration classes and create the loops that enumerate them.

Operating Mode	Energy Consumption (nJ)	Resynchronization Cost (cycles)
`Active`	3.570	0
`Standby`	0.830	20
`Nap`	0.320	300
`PowerDown`	0.005	9,000

Figure 4: Energy consumptions and resynchronizations costs for our operating modes. During transitions from a low-power mode to the active mode, a full active mode energy is assumed to be spent.

Benchmark	Input Size (MB)	Base1 (mJ)	Base2 (mJ)
`phods`	40.1	44,492	36,007
`seq`	44.5	50,525	39,237
`flt`	58.8	53,078	43,521
`tomcatv`	49.0	44,221	30,012
`swim`	55.2	51,660	33,964
`eflux`	33.7	86,719	71,900
`lu`	40.5	73,868	60,384

Figure 5: Benchmark codes used in the experiments and their important characteristics.

Figure 4 shows the energy consumptions (per access) and resynchronization costs for the operating modes used in our experiments.

We used seven array-dominated codes to measure the benefits of our approach. The important characteristics of these codes are given in Figure 5. `phods` and `seq` are two different motion estimation codes; `flt` is a digital filtering routine; `tomcatv` and `swim` are two array-based benchmarks from the Spec suite; `eflux` is an array-based benchmark from the Perfect Club suite; and `lu` is an LU decomposition code. Base1 and Base2 refers to *base* (*original*) *memory system energy consumptions* (data accesses only) for a cacheless system and a system with a 16KB, 2-way set-associative cache with a line size of 32 bytes, respectively. Note that the energy figures in the Base2 column include the energy expended in cache as well as main memory. In both the cases, the default memory configuration consists of eight 8MB memory banks (denoted 8 × 8MB). The remaining figures given in this section are *percentage improvements* (*reductions*) over these base figures. In all experiments, we employed a powerful back-end compiler which performs several important optimizations such as global register allocation and instruction scheduling. The use of node merging was necessary in only one case (`eflux`) to eliminate a cycle in the CLDG; we did not use node merging to reduce the loop overhead. The increase in execution times due to our optimizations was always less than 1%. Also, there was no opportunity to use node splitting.

The second column in Figure 6 (Imprv1) gives the percentage reductions over Base1 (in a cacheless system) when our approach is used. We see that our strategy based on the iteration class concept brings a 26.3% improvement on the average. The third column in the same figure reports the energy improvements (again, over Base1) when classical cache locality optimizations are used.[1] The locality optimizations used include linear loop transformations and iteration space tiling

[1]Note that in this case, these locality optimizations should normally not be applied as the system does not contain cache. Our purpose here is to see whether locality-oriented optimizations are as successful as our strategy in optimizing off-chip memory energy.

Benchmark	Imprv1	Imprv2	Imprv3	Imprv4	Imprv5
phods	31.4	13.0	27.5	14.4	33.8
seq	28.6	6.8	22.0	10.2	24.2
flt	18.9	12.1	16.3	10.6	22.1
tomcatv	34.1	13.2	29.8	11.1	41.0
swim	20.4	10.1	19.7	9.9	20.5
eflux	16.6	5.4	18.5	4.2	18.5
lu	33.8	2.5	29.9	5.6	31.3

Figure 6: Percentage energy improvements.

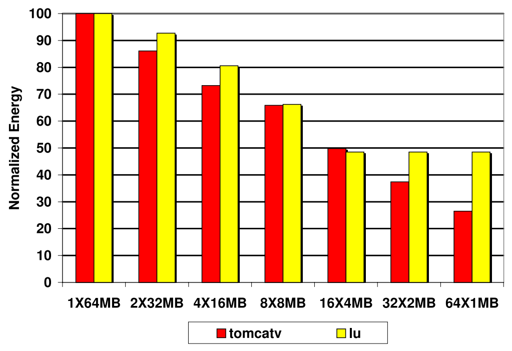

Figure 7: Impact of memory bank configuration.

(blocking) [11]. We experimented with different tile sizes and (for each benchmark) selected the one which gave the best result. In comparing the second and third columns of Figure 6, we observe that our approach, in general, performs much better than a pure locality-oriented approach. The fourth column of the figure (Imprv3) gives the percentage improvements due to our approach over Base2 (with cache). We see that the average energy improvement is about 23.4%. Using cache locality optimizations instead brings an improvement of 9.4% (see the fifth column). The last column (Imprv5), on the other hand, shows the percentage energy improvements if our current strategy is slightly modified to take cache considerations into account (and if doing so does not conflict with our iteration class based optimizations). Note that this last strategy has not been fully implemented yet and obtained here through hand optimizations. The results show that by incorporating locality based techniques into our framework, we can achieve (on average) a 27.4% improvement (over Base2). This last result motivates us to integrate our strategy with locality optimizations in the future.

So far, we have used only a single memory configuration: 8 $\times$ 8MB. Figure 7 gives *normalized energy consumptions* (with respect to Base1) for two representative codes, `lu` and `tomcatv`, when different memory configurations are used (in a cacheless system). We observe from this figure that increasing the number of banks (by keeping the total memory size constant) generally increases energy savings (as it gives more control to the compiler in placing larger sections of the address space into low-power modes). In many codes, however, beyond a certain point (depending on the access pattern), a finer-granular memory system does not bring more benefits. This happens for example with the `lu` code beyond sixteen banks (see Figure 7).

6. Conclusions

In this paper, we have presented a compiler-oriented memory energy optimization technique based on a concept called (loop) iteration classification. The objective of this technique is to increase the duration of idle periods of memory banks, thereby saving energy using low-power operating modes more aggressively. Our experimental evaluation indicates that large energy savings are possible using this technique.

7. REFERENCES

[1] 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., May 1999.

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

[3] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In *Proc. the 7th International Conference on High Performance Computer Architecture,* Monterrey, Mexico, January 2001.

[4] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, Inc., 1994.

[5] C. Ellis. The case for higher level power management. In *Proceedings of HotOS,* March 1999.

[6] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proc. the 37th Design Automation Conference,* Los Angeles, California USA, June 5-9, 2000.

[7] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and David Wonnacott. The Omega Library interface guide. *Technical Report CS-TR-3445,* CS Dept., University of Maryland, College Park, MD, March 1995.

[8] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, June 1997.

[9] J. R. Lorch and A. J. Smith. Software strategies for portable computer energy management. *IEEE Personal Communications*, pp. 60–73, June 1998.

[10] V. Tiwari, S. Malik, A. Wolfe, and T. C. Lee. Instruction level power analysis and optimization of software, *Journal of VLSI Signal Processing Systems,* Vol. 13, No. 2, August 1996.

[11] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.