Cache Miss Clustering for Banked Memory Systems*

O. Ozturk, G. Chen, M. Kandemir
Computer Science and Engineering Department
Pennsylvania State University
University Park, PA 16802, USA
{ozturk, gchen, kandemir}@cse.psu.edu

M. Karakoy
Department of Computing
Imperial College
London SW7 2AZ, UK
M.Karakoy@imperial.ac.uk

ABSTRACT

One of the previously-proposed techniques for reducing memory energy consumption is memory banking. The idea is to divide the memory space into multiple banks and place currently unused (idle) banks into a low-power operating mode. The prior studies – both hardware and software domain – in memory energy optimization via low-power modes do not take the data cache behavior explicitly into account. As a consequence, the energy savings achieved by these techniques can be unpredictable due to dynamic cache behavior at runtime. The main contribution of this paper is a compiler optimization, called the bank-aware cache miss clustering, that increases idle durations of memory banks, and as a result, enables better exploitation of available low-power capabilities supported by the memory system. This is because clustering cache misses helps to cluster cache hits as well, and this in turn increases bank idleness. We implemented our cache miss clustering approach within a compilation framework and tested it using seven array-intensive application codes. Our experiments show that cache miss clustering saves significant memory energy as a result of increased idle periods of memory banks.

1. INTRODUCTION AND MOTIVATION

One of the energy reduction techniques currently employed by memory chips is low-power operating modes. A low-power operating mode in DRAMs is typically implemented by shutting off certain components within the memory chip. It is typically used in conjunction with a banked memory system, where the memory banks that are not used by the current computation can be put into a low-power mode. Most of the memory energy management schemes via low-power modes studied in the literature, however, do not take data cache behavior explicitly into account. In other words, they operate, in a sense, a cache oblivious fashion. As a consequence, the results achieved by these techniques can be unpredictable due to dynamic cache behavior at runtime. In principle, taking data cache behavior into account can allow a more effective management of available low-power operation modes, and this can in turn boost energy savings.

The main goal of the work in this paper is to study a compiler-directed code restructuring scheme for increasing energy savings coming from low-power modes used in banked memories. The proposed scheme employs *cache miss clustering,* and tries to increase bank idle times, which allows better use of the low-power operating modes supported by the underlying memory hardware. More specifically, cache miss clustering clusters data cache misses and this helps cluster accesses to main memory (which is banked). This clustering of accesses also means clustering of memory idle cycles, which in turn means longer idle periods for banks and thus higher energy savings. While a variant of cache miss clustering has been used by the previous research in enhancing memory level parallelism [11], to the best of our knowledge, this paper presents the first study that uses cache miss clustering for increasing energy benefits in a banked memory system.

We automated our approach within an optimizing compiler built upon the SUIF infrastructure from Stanford University [6] and performed experiments with a set of embedded applications. Our extensive experimental analysis indicates that cache miss clustering is very effective in increasing average length of idle periods for memory banks. As a result of this increase in the length of idle periods, memory energy savings achieved through low-power modes are increased significantly. Our experimental results also show that cache miss clustering reduces execution cycles.

The rest of this paper is organized as follows. Section 2 revises the banked memory concept and low-power operation modes. Section 3 discusses the prior work on energy optimization for memory banks. Section 4 presents our approach in detail and gives our compiler algorithm. Section 5 presents an experimental evaluation of our approach. Section 6 concludes the paper.

2. POWER MODES OF MEMORY BANKS

In this work, we focus on an RDRAM-like off-chip memory architecture [13] where off-chip memory is partitioned into several banks, each of which can be activated or deactivated independently from others. In this architecture, when a memory bank is not actively used, it can be placed into a low-power operating mode. While in a low-power mode, a bank typically consumes much less energy than in the active mode. However, a bank in a low power need to be reactivated before its contents can be accessed, which incurs reactivation overheads. Typically, a more energy-saving low-power operating mode has also a higher reactivation overhead. Thus, it is important to select the most appropriate low-power mode to switch to when the bank becomes idle.

In this study, we assume four different operating modes: an active mode (the mode during which the memory read/write activity can occur) and three low-power modes, namely, standby, napping, and power-down. Current DRAMs [13] support up to six power modes with a few of them supporting only two modes. We collapse the read, write, and active without read or write modes into a single mode, called the active mode, in our experimentation. However, one may choose to vary the number of modes based on the target DRAM architecture. The energy consumptions and reactivation overheads for these operating modes are given in Table 1. The energy values shown in this table have been obtained from the mea-

*This work is supported in part by NSF CAREER #0093082 and a fund from GSRC.

	Energy Consumption	Reactivation Overhead
Active	3.570nJ	0 cycles
Standby	0.830nJ	2 cycles
Napping	0.320nJ	30 cycles
Power-Down	0.005nJ	9,000 cycles

Table 1: Energy consumptions (per cycle) and reactivation times for different operating modes. These numbers include both dynamic (switching) and static (leakage) components.

sured current values associated with memory modules documented in memory data sheets (for a 3.3 V, 2.5 nsec cycle time, 8 MB memory) [14]. The reactivation times (overheads) are also obtained from data sheets. Based on trends gleaned from data sheets, the energy values are increased by 30% when module size is doubled.

An important parameter that helps us choose the most suitable low-power mode is *bank inter-access time* (BIT), i.e., the time between successive accesses (requests) to a given memory bank. Obviously, the larger the BIT, a more aggressive low-power mode can be exploited. Then, the problem of effective power mode utilization can be defined as one of accurately estimating the BIT and using this information to select the most suitable low-power mode. This estimation can be done by software using the compiler [3, 2] or OS support [7], by hardware using a prediction mechanism attached to the memory controller [3], or by a combination of both. While the compiler-based techniques have the advantage of predicting BIT accurately for a specific class of applications, runtime and hardware based techniques are able to capture runtime variations in access patterns (e.g., those due to cache hits/misses) better.

In this paper, we employ a hardware-based BIT prediction mechanism. This prediction mechanism is similar to the mechanisms used in current memory controllers. Specifically, after 10 cycles of idleness, the corresponding bank is put in the standby mode. Subsequently, if the bank is not referenced for another 100 cycles, it is transitioned into the napping mode. Finally, if the bank is not referenced for a further 1,000,000 cycles, it is put into the power-down mode. Whenever the bank is referenced, it is brought back into the active mode, incurring the corresponding reactivation overhead (based on what mode it was in). It needs to be mentioned however that our bank-aware cache miss clustering scheme works with software based BIT prediction schemes as well. We focus on a single program environment, and do not consider the existence of a virtual memory system. Exploring the energy impact of our approach in the presence of a virtual address translation is part of our planned further research.

3. RELATED WORK

While there exists a large body of work on memory performance evaluation [18, 15], banked memory based studies for energy reduction are relatively new. It has been observed [5, 19, 1] that memory system is a dominant consumer of the overall system energy, making this a ripe candidate for software and hardware optimizations, thus serving as a strong motivation for the research presented in this paper. Lebeck et al [7] have presented OS based strategies for optimizing energy consumption of a banked memory architecture. The idea is to perform page allocation in an energy-efficient manner by taking into account the banked structure of the main memory. Delaluz et al [3] have discussed architectural and compiler techniques for low-power operating mode management. They have also discussed the bank pre-activation strategy used in this study. Fan et al [4] have shown how to set memory controller policies for the best memory energy behavior. Delaluz et al [2] have presented an experimental evaluation of some well-known classical high-level compiler optimizations on memory energy. *In contrast to these previous efforts, the work described in this paper takes into account data cache explicitly, and restructures the application code to cluster data cache misses.* In this respect, it is different from the previous studies on banked memory optimization. However, the proposed approach can co-exist with many of the previously-proposed memory energy optimization techniques. In fact, our point is that, since our approach increases bank idleness, it will increase the effectiveness of any hardware or software based scheme that makes use of low-power operating modes. The only cache miss clustering related studies that we are aware of are [11] and [12]. The first of these studies uses a variant of cache miss clustering as a means of improving memory parallelism in high-performance computing. and the second one compares it against software prefetching, a well-known latency hiding mechanism. Our cache miss clustering scheme is very different from these prior papers. First, our approach is bank sensitive and consequently it is preceded by a code transformation that isolates the loop iterations that access a given set of memory banks. Clustering misses without considering how the arrays are mapped to the banks in the memory system would not be very useful in our context. Second, our approach is oriented towards reducing energy consumption rather than improving memory level parallelism in high-end machines. As a result, our code transformation approach is entirely different from the one in [11, 12], which is based on loop strip-mining, a loop-level code restructuring technique.

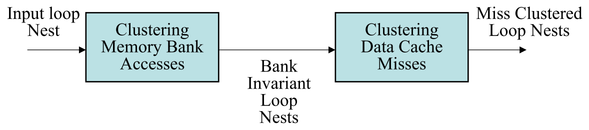

Input loop Nest
Clustering Memory Bank Accesses
Bank Invariant Loop Nests
Clustering Data Cache Misses
Miss Clustered Loop Nests

Figure 1: Overview of our approach.

4. COMPILER APPROACH

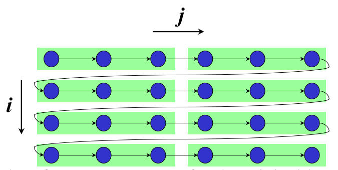
Figure 1 shows our two step approach to restructuring code to cluster data cache misses. The first part, which is explained in Section 4.3, clusters array references (load/store operations) based on the memory banks they access. The idea can be summarized as decomposing a loop nest into multiple smaller loop nests such that each of the resulting smaller nests accesses the same set of banks throughout its execution. The second part, which is explained in Section 4.4, handles these small loop nests generated by the first part one by one, and it transforms the loop nest such that the data cache misses are clustered. Our approach operates under the assumption of cache line alignment, which means each array starts at the beginning of a new cache line. Many compilers have directives that enforce this type of cache line alignments. Also, we assume that before our approach is invoked, array-to-bank assignment has already been performed. Our approach takes the size of a cache bank as parameter. Changing the size of cache bank requires recompilation of the application in order to maximize the benefit that can be achieved by cache miss clustering. This may be a problem for an application the needs to run without modification on multiple systems with different configurations. However, for embedded systems where the software are usually co-designed with the hardware, this is not a serious drawback.

4.1 Motivational Example

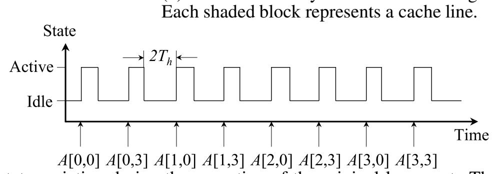
Figure 2 illustrates an example to motivate cache miss clustering. For ease of illustration, we consider a single bank in this example. In (a), we give the original loop nest we want to optimize. Part (b) shows the order of memory accesses. Each circle indicates an array element and the arrows capture the traversal order of array elements. We also highlight the cache line boundaries to show how the execution transitions from one cache line to another, assuming a cache line holds three elements. Part (c) of the figure illustrates the idle and active periods caused by the access pattern in part (b). We note that we incur a single cache miss in every three accesses. Assuming a cache hit latency of T_h cycles, the length of each idle period is $2T_h$ cycles. Our cache miss clustering strategy (to be explained shortly) transforms the original loop nest in (a) to the one shown in part (d). Part (e) illustrates the resulting data access pattern and part (f) shows the active and idle periods for the transformed loop nest. The important point here is that the new access pattern clusters the two cache misses together and this in turn increases the length of each idle period from $2T_h$ cycles to $4T_h$ cycles. This new pattern is clearly better than the original one from the perspective of energy

```
for $i = 0$ to 3
   for $j = 0$ to 5
      $\ldots A[i, j] \ldots$
```

(a) Original loop nest.



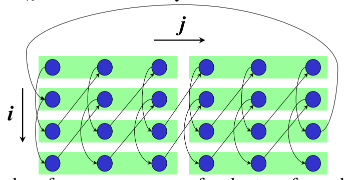
(b) Order of memory accesses for the original loop nest. Each shaded block represents a cache line.



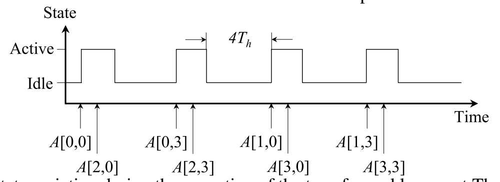
(c) Bank state variation during the execution of the original loop nest. The length of each idle period is $2T_h$ cycles, where T_h is cache hit latency.

```
for $i = 0$ to 1
   for $j = 0$ to 5
      $\ldots A[i, j] \ldots$
      $\ldots A[i+2, j] \ldots$
```

(d) Transformed loop nest.



(e) Order of memory accesses for the transformed loop nest. Each shaded block represents a cache line.



(f) Bank state variation during the execution of the transformed loop nest.The length of each idle period is $4T_h$ cycles, where T_h is cache hit latency.

Figure 2: An example motivating cache miss clustering.

saving. This is because we now have longer idle periods and can potentially use a more aggressive low-power operating mode for this bank.

4.2 Notation

For ease of discussion, let us first define some notations we use in the rest of this paper. Note that this paper focuses on array-based, loop-intensive programs, e.g., embedded multi-media applications. Such a program is typically composed of a set of loop nests; each loop nest accesses a set of arrays. We use the abstract form below to represent an array based loop-intensive program $\mathcal{P}$ that consists of l loop nests:

$$\mathcal{P} = \langle \mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_l \rangle,$$

where $\mathcal{N}_i$ $(i = 1, 2, ..., l)$ is the i^{th} loop nest of program $\mathcal{P}$. Further, we use $\mathcal{X}_i$ to represent the set of arrays accessed by loop nest $\mathcal{N}_i$, and $\mathcal{X}$ to represent the set of arrays accessed by program $\mathcal{P}$. Based on these definitions, we have

$$\mathcal{X} = \bigcup_{i=1}^{l} \mathcal{X}_i.$$

A loop nest $\mathcal{N}_i$ of the following form:

```
$\mathcal{N}_i$: for $j_1 = l_1$ to $u_1$ step $d_1$
       for $j_2 = l_2$ to $u_2$ step $d_2$
       ......
          for $j_k = l_k$ to $u_k$ step $d_k$
             Body
```

can be represented in an abstract form as:

$$\mathcal{N}_i : \text{for } \vec{I} \in [\vec{L}_i, \vec{U}_i] \text{ step } \vec{d} \ \langle s_1(\vec{I}), s_2(\vec{I}), ..., s_m(\vec{I}) \rangle,$$

where $\vec{I}$ is the iteration vector (i.e., a vector that contains the loop iterators in the nest from top to bottom), $\vec{L} = (l_1, l_2, ..., l_k)^{\text{T}}$ and $\vec{U} = (u_1, u_2, ..., u_k)^{\text{T}}$ are the lower and upper bound vectors, $\vec{d} = (d_1, d_2, ..., d_k)^{\text{T}}$ is the loop step vector (step $\vec{d}$ can be omitted if all its elements are 1), and $s_j(\vec{I})$ $(j = 1, 2, ..., m)$ is the j^{th} array reference in the body of loop nest $\mathcal{N}_i$ (for ease of discussion, we omit non-array references). When this loop nest is executed, $\vec{I}$ takes values from $\vec{L}$ to $\vec{U}$ in the lexicographic order. The array element accessed by $s_j(\vec{I})$ can be represented as $X[f_j(\vec{I})]$, where X $(X \in \mathcal{X}_i)$ is the name of the array. Also, function f_j maps iteration vector $\vec{I}$ to a vector of subscripts for array X. In this paper, we assume that f_j is an affine function of $\vec{I}$.

4.3 Clustering Memory Bank Accesses

In this section, we discuss a method for isolating memory references (loads and stores) to a specific set of banks at a given time. We start by making an important definition, namely, the *bank-invariant loop nest*. Given a loop nest $\mathcal{N}^*$ of the form:

$$\mathcal{N}^* : \text{for } \vec{I} \in [\vec{L}, \vec{U}] \ \langle s_1(\vec{I}), s_2(\vec{I}), ..., s_m(\vec{I}) \rangle,$$

we say that it is a bank-invariant loop nest if the following constraint is satisfied for all $1 \le j \le m$:

$$\forall \vec{I}_1, \vec{I}_2 \in [\vec{L}, \vec{U}] : b(X_j[f_j(\vec{I}_1)]) = b(X_j[f_j(\vec{I}_2)]), \qquad (1)$$

where X_j is the array accessed by array reference s_j, f_j is the mapping function that maps an iteration vector of the loop nest to a subscript vector of array X_j, and function $b(x)$ gives the ID of the memory bank that stores array element x. In other words, the memory bank accessed by a given array reference (load/store operation) s_j does not change throughout the execution of $\mathcal{N}^*$.

We cluster memory bank accesses by splitting a given loop nest $\mathcal{N}$, which may not be bank-invariant, into a set of bank-invariant loop nests $\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_m$ such that $[\vec{L}_i, \vec{U}_i] \subseteq [\vec{L}, \vec{U}]$ ($[\vec{L}_i, \vec{U}_i]$ is the iteration space of nest $\mathcal{N}_i$) and that the result of executing these bank-invariant loop nests is equivalent to that of executing the original loop nest $\mathcal{N}$. Let us assume that the bank size is B and loop nest $\mathcal{N}$ uses a arrays, the capacity of data cache is C, the data cache uses a W-way associative mapping and LRU replacement policy, and the size of each cache line is equal to that of each array element. One can show that if $B \le C/W$, $a \le W$, clustering memory bank accesses does not increase the number of cache misses.

Our compiler algorithm for clustering memory bank accesses is given in Figure 3. Figure 4 illustrates the idea behind forming bank-invariant loop nests using an example. Part (a) of this figure gives the code of a loop nest $\mathcal{N}$ that accesses arrays X and Y. The bank layout of arrays X and Y are given in part (b). We can see that, due to the limited capacity of a bank, each array has to be stored in two banks: banks B_1 and B_2 for array X, and banks B_3 and B_4 for array Y. Consequently, during the execution of loop nest $\mathcal{N}$, depending on the current value of iteration vector $(i, j)^{\text{T}}$, the array reference s_1 may access either bank B_1 or bank B_2, and s_2 may access either bank B_3 or bank B_4. Applying our bank memory access clustering algorithm given in Figure 3 to loop nest $\mathcal{N}$, we obtain four bank invariant loop nests $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$, and $\mathcal{N}_4$, as shown in Figure 4(c). We note that the banks accessed by array references s_1 and s_2 do not change in a bank-invariant loop nest. One of the side benefits of clustering bank memory accesses is that it can be expected to improve cache locality as well since, at any given time frame, the memory accesses are localized in a small set of memory banks. We will return this issue in our discussion of experimental results.

4.4 Clustering Data Cache Misses

Our compiler algorithm for clustering data cache misses is given in Figure 5. This algorithm takes a bank invariant loop nest and cache miss *clustering factor* c as input, generates a new loop nest in which data cache misses are clustered. Clustering factor determines the number of cache misses that we want to cluster together.

```
Input:
  loop nest $\mathcal{N}$;
Output:
  bank-invariant loop nest set $S = \{\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_m\}$;

$S = \{\mathcal{N}\}$;
for $i = 1$ to $m$ {
  // assume $s_i(\vec{I})$ accesses array element $X[f(\vec{I})]$
  if(array $X$ is stored in $k$ banks and $k > 1$) {
    for each loop nest $c\mathcal{N}_i \in S$ {
      split $\mathcal{N}_i$ into $k$ loop nests $\mathcal{N}_{i,1}, \mathcal{N}_{i,2}, ..., \mathcal{N}_{i,k}$
        such that $s_i$ accesses only one bank within $\mathcal{N}_{i,j}$;
      $S' = S' \cup \{\mathcal{N}_{i,1}, \mathcal{N}_{i,2}, ..., \mathcal{N}_{i,k}\}$;
    }
  }
  $S = S'$;
}
```

Figure 3: Algorithm for splitting loop nest $\mathcal{N}$ into m bank-invariant loop nests ($\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_m$).

```
$\mathcal{N}$: for $i = 0$ to 99
        for $j = 0$ to 99
            $s_1$: ...$X[i, j]$...
              // access $B_1$, $B_2$
            $s_2$: ...$Y[j, i]$...
              // access $B_3$, $B_4$
```

(a) Original loop nest $\mathcal{N}$.

Bank B_1 X[0..49, 0,99]	Bank B_3 Y[0..49, 0,99]
Bank B_2 X[50..99, 0,99]	Bank B_4 Y[50..99, 0,99]

(b) Bank layout of arrays X and Y, assuming that each bank can hold up to 5000 array elements.

```
$\mathcal{N}_1$: for $i = 0$ to 49                          $\mathcal{N}_2$: for $i = 0$ to 49
      for $j = 0$ to 49                                 for $j = 50$ to 99
        $s_1$: ...$X[i, j]$... // access $B_1$           $s_1$: ...$X[i, j]$... // access $B_1$
        $s_2$: ...$Y[j, i]$... // access $B_3$           $s_2$: ...$Y[j, i]$... // access $B_4$

$\mathcal{N}_3$: for $i = 50$ to 99                         $\mathcal{N}_4$: for $i = 50$ to 99
      for $j = 0$ to 49                                 for $j = 50$ to 99
        $s_1$: ...$X[i, j]$... // access $B_2$           $s_1$: ...$X[i, j]$... // access $B_2$
        $s_2$: ...$Y[j, i]$... // access $B_3$           $s_2$: ...$Y[j, i]$... // access $B_4$
```

(c) Bank-invariant loop nests obtained by applying bank-based memory access clustering to loop nest $\mathcal{N}$.

Figure 4: Splitting loop nest $\mathcal{N}$ into a set of bank-invariant loop nests: $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$, and $\mathcal{N}_4$.

We can cluster more cache misses by using a larger clustering factor. However, increasing the clustering factor also increases the size of program code. Further, a very large clustering factor can also increase the number of cache misses. Figure 6 illustrates how our cache miss clustering algorithm works using an example where we cluster cache misses in the loop:

$$\mathcal{N} : \text{for } i = 0 \text{ to } 59 \ \langle s_1 : X[f_1(i)], s_2 : Y[f_2(i)]\rangle$$

with a clustering factor of $c = 3$. In this figure, we can observe that, we split the iteration space of loop $\mathcal{N}$ into three sections: $[0, 19]$, $[20, 39]$, and $[40, 59]$. Since the values of the iteration vectors from the different sections are far from each other, and the array element accessed by each array reference is determined by the value of the iteration vector, the addresses of the data elements accessed by the iterations that belong to the different sections tend to be far from each other. Therefore, data reuses are not likely to happen between two loop iterations from the different sections. As an example, array reference $s_1(i)$ executed in the iteration space section $[20, 39]$ is not likely to reuse any data brought to the cache by $s_1(i)$ executed in the iteration space section $[0, 19]$. Our cache miss clustering algorithm is based on this observation. As shown in Figure 6, we achieve our goal by clustering the array references that are originally executed in the different sections of the iteration space since these array references are likely to incur cache misses together. And, this clustering generates long active periods and, consequently, longer idle periods for memory banks.

An important issue regarding our implementation is about data dependences. Note that both bank-based clustering and cache miss clustering modify the original execution order of loop iterations. Therefore, their legality depends on the data dependences in the loop nest. In our current implementation, we do not apply them to a loop nest if they violate any data dependence relationship in the nest.

```
Input:
  bank-invariant loop nest $\mathcal{N}$:
    $\mathcal{N}$: for $\vec{I} \in [\vec{L}, \vec{U}]$ $\langle s_1(\vec{I}), ..., s_m(\vec{I})\rangle$
      where $\vec{L} = (l_1, l_2, ..., l_n)^{\mathrm{T}}$ and $\vec{U} = (u_1, u_2, ..., u_n)^{\mathrm{T}}$
  $c$: cache miss clustering factor.
Output:
  the code for transformed loop nest.

$r = (u_1 - l_1)$ mod $c$;
if($r \neq 0$) {
  // peeling the first $r$ iterations from the outer most loop.
  $\vec{\delta} = (r - 1, 0, 0, ..., 0)^{\mathrm{T}}$;
  // generate code for the first $r$ iterations.
  emit "for $\vec{I} \in [\vec{L}, \vec{L} + \vec{\delta}]$ $\langle s_1(\vec{I}), ..., s_m(\vec{I})\rangle$";
}
$\vec{\delta} = ((u_1 - l_1 - r)/c, 0, 0, ..., 0)^{\mathrm{T}}$;
// compute the new loop bounds
$\vec{L}' = (l_1 + r, l_2, l_3, ..., l_n)^{\mathrm{T}}$;
$\vec{U}' = (l_1 + r + \lfloor (u_1 - l_1)/c \rfloor - 1, u_2, u_3, ..., u_n)^{\mathrm{T}}$;
// generate the loop control structure for the transformed loop nest
emit "for $\vec{I} \in [\vec{L}', \vec{U}']$ $\langle$";
// generate the body for the transformed loop nest
for $i = 1$ to $m$
  emit "$s_i(\vec{I}), s_i(\vec{I} + \vec{\delta}), s_i(\vec{I} + 2\vec{\delta}), ..., s_i(\vec{I} + (c - 1)\vec{\delta})$";
emit "$\rangle$"
```

Figure 5: Cache miss clustering algorithm.

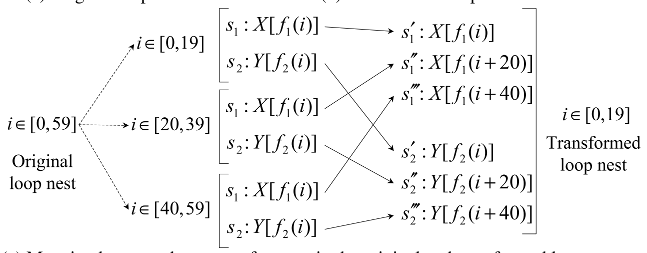
```
$\mathcal{N}$: for $i = 0$ to 59 $\langle$
      $s_1$ : $X[f_1(i)]$;
      $s_2$ : $Y[f_2(i)]$;
   $\rangle$
```

(a) Original loop nest.

$\Rightarrow$

```
$\mathcal{N}$]: for $i = 0$ to 19 $\langle$
      $s_1'$ : $X[f_1(i)]$;
      $s_1''$ : $X[f_1(i + 20)]$;
      $s_1'''$ : $X[f_1(i + 40)]$;
      $s_2'$ : $Y[f_2(i)]$;
      $s_2''$ : $Y[f_2(i + 20)]$;
      $s_2'''$ : $Y[f_2(i + 40)]$;
   $\rangle$
```

(b) Transformed loop nest.



(c) Mapping between the array references in the original and transformed loop nests.

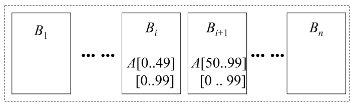
Figure 6: An example showing how our approach clusters cache misses (clustering factor $c = 3$).

4.5 A Complete Example

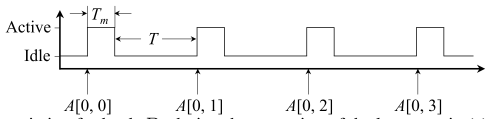
In this section, we present a complete example showing how the two steps of our approach explained in Sections 4.3 and 4.4 work together. Figure 7(a) shows the code for a loop nest $\mathcal{N}$ that accesses array $A[0.99][0.99]$. For the sake of explanation, let us assume that each memory bank can store up to 500 array elements. Therefore, array A can be stored in two banks B_i and B_{i+1} (see Figure 7(b)). We further assume that the size of a cache line is equal to that of an array element, the cache miss latency is T_m cycles, and per iteration execution time, excluding the delay due to cache misses, for loop nest $\mathcal{N}$ is T cycles. Figure 7(c) shows the state variations for bank B_i during the execution of loop nest $\mathcal{N}$. We can observe that loop nest $\mathcal{N}$ incurs a data cache miss every $T + T_m$ cycles, and the length of bank idle period between two successive cache misses is T cycles. By applying memory bank access clustering to loop nest $\mathcal{N}$, we obtain two bank-invariant loop nests $\mathcal{N}_1$ (accessing only bank B_i) and $\mathcal{N}_2$ (accessing only bank B_{i+1}), as shown in Figure 7(d). By applying cache miss clustering with a clustering factor of $c = 2$ to the loop nest $\mathcal{N}_1$ and $\mathcal{N}_2$, we obtain loop nests $\mathcal{N}_1'$ and $\mathcal{N}_2'$ in Figure 7(e). Figure 7(f) shows the state variations for bank B_i during the execution of $\mathcal{N}_1'$. In this figure, we observe that loop nests $\mathcal{N}_1'$ and $\mathcal{N}_2'$ incur two clustered cache misses every $2(T + T_m)$ cycles, and the bank idle period between two successive cache misses is $2T$ cycles. To sum up, clustering cache misses increases the length of idle period so that we have more opportunities to put memory

```
N: for i = 0 to 99
     for j = 0 to 99 {
       x = A[i, j];
       ... use x...
     }
```

(a) A loop nest accessing array A.



(b) Memory layout for array A.



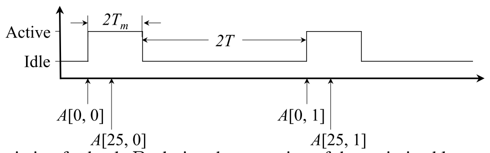
(c) State variation for bank B_i during the execution of the loop nest in (a).

```
N1: for i = 0 to 49
      for j = 0 to 99 {
        x = A[i][j];
        ... use x...
      }
N2: for i = 50 to 99
      for j = 0 to 99 {
        x = A[i][j];
        ... use x...
      }
```

(d) After clustering memory bank accesses.

```
N'1: for i = 0 to 24
       for j = 0 to 99 {
         x1 = A[i][j];
         x2 = A[i + 25][j];
         ... use x1, x2...
       }
N'2: for i = 50 to 74
       for j = 0 to 99 {
         x1 = A[i][j];
         x2 = A[i + 25][j];
         ... use x1, x2...
       }
```

(e) After clustering cache misses.



(f) State variation for bank B_i during the execution of the optimized loop nest in (e).

Figure 7: An example demonstrating the two steps of our approach.

banks into the low-power mode; it also reduces the number of times that a bank needs to be re-activated, and thus reduces the overall re-activation overheads.

5. EXPERIMENTAL EVALUATION

5.1 Setup

To evaluate our bank-aware cache miss clustering scheme, we automated our approach within an open-source compiler infrastructure called SUIF [6], and made a series of simulation-based experiments using SimpleScalar [17]. SUIF is built a series of compiler phases and our approach has been implemented as a separate phase. The additional increase in compilation times due to the addition of our phase was about 27% on average, as compared to the case without our phase. In our experiments, we assume the embedded system consists of a 400MHz processor, 8KB 4-way data cache, and eight main memory bank. The access latencies of the data cache and the main memory are 2 and 110 cycles, respectively. However, we also performed experiments with different values for off-chip memory access latency.

The important characteristics of the benchmark codes that we used to measure the energy benefits of our cache miss clustering approach are given in Table 2. `fourier` and `flt` are Fourier transform and digital filtering routines, respectively. `adi` and `cholesky` are ADI and cholesky decomposition codes; `hydro2d` is an array-dominated code from the Spec Benchmark Suite; and `tis` and `tsf` are from the Perfect Club Benchmarks. The third column in Table 2 gives the total dataset size manipulated by the corresponding code. Finally, the last two columns give the energy consumption and execution cycles with the baseline scheme (explained below). While our code restructuring increased the executable sizes by about 14% on the average, the increase in instruction cache misses was very small (less than 0.4%). In any case, in these benchmark codes, the data memory behavior is much more dominant than the instruction memory behavior. The data cache misses of these benchmark codes varied between 8.8% and 21.4%. The default clustering factor (see Section 4.4) used in our experiments is 8. The energy consumption results presented below capture all the energy consumption due to data accesses. This includes the energy consumed in the data cache, the off-chip memory and the interconnects.

Benchmark Name	Number of Lines	Input Size	Baseline Energy	Baseline Cycles
`adi`	56	78MB	19.3mJ	3.92M
`cholesky`	34	61MB	61.1mJ	7.10M
`hydro2d`	52	44MB	76.3mJ	6.59M
`flt`	85	51MB	328.1mJ	11.57M
`fourier`	167	57MB	411.7mJ	8.90M
`tis`	485	56MB	511.0mJ	12.04M
`tsf`	1986	60MB	620.4mJ	16.71M

Table 2: Our benchmarks and their important characteristics.

We conducted experiments with five different schemes:

Baseline: This scheme does not perform any energy and performance optimizations, and used as a base case against which the other schemes can be normalized and compared. Its memory energy or performance numbers are given in the last two columns of Table 2.

LOOP: This represents a conventional loop restructuring scheme for optimizing cache locality. It basically employs two types of optimizations: loop permutation (i.e., interchanging the order of two loops to ensure units stride access with the innermost loop after the transformation) and loop tiling (i.e., generating the blocked version of a loop for improving cache locality). While this scheme does not specifically target energy, the performance improvements it brings can translate to energy savings since the number of accesses to the main memory is reduced. The specific locality optimization techniques used in this scheme are adapted from [9, 8, 10].

CMC: This is the scheme proposed and discussed in this paper. As mentioned earlier, it energy benefits come from both increased bank idleness (as a result of clustering) and improved data locality.

PRI: This is a previously-proposed scheme [16] to energy optimization for banked memories. It tries to cluster memory accesses to a small set of banks at a given time and this improves what is called bank locality. It needs to be noted that this approach works in a cache independent fashion. That is, it does not take cache behavior explicitly into account.

The important point to emphasize here is that all the schemes listed above, including the baseline scheme (which does not employ any code/ data optimization) make use of the same underlying hardware-based low-power management scheme explained in Section 2. Therefore, the differences between the different schemes can be attributed to the duration of the bank idle periods (BIT values) they generate. It is also important to note that we compare our scheme (CMC) to two previously proposed schemes: a data locality oriented one (LOOP, adapted from [9, 8, 10]) and a memory energy oriented one (PRI, adapted from [16]).

5.2 Results

We first present in Figure 8 the CDF (cumulative distribution function) for the bank inter-access times (BIT) for original applications when all eight banks are considered together. Specifically, an (x,y) point on any curve in this plot indicates that x% of the total bank idle periods are between y cycles or less. We see from this plot that there are many short idle periods, which are potential candidates for optimization.

The graph in Figure 9 gives the normalized energy consumptions for our benchmarks with four different schemes described above. Our first observation is that the LOOP scheme generates some energy savings (16.1% on an average). This is a direct result of (1) optimizing data cache behavior and (2) increasing memory idleness. When we look at the result of CMC (our approach) and PRI, we see that they are competitive. While in some benchmarks PRI outperforms CMC, in others the latter generates better energy savings than the former. The average energy savings brought by PRI and CMC are 25.4% and 27.3%, respectively. These results clearly indicate

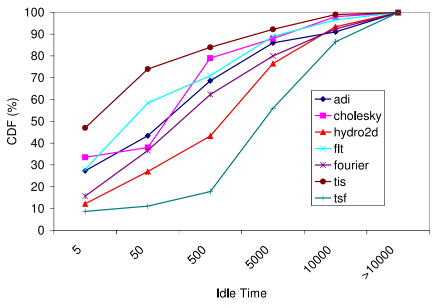


Figure 8: CDF for memory bank inter-access times (accumulated over all banks).

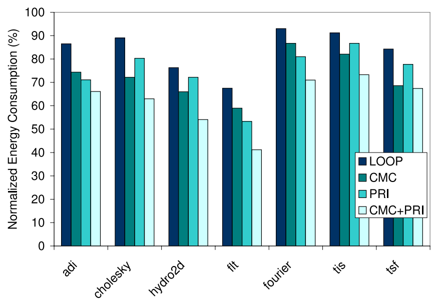


Figure 9: Normalized energy consumptions with respect to the baseline scheme.

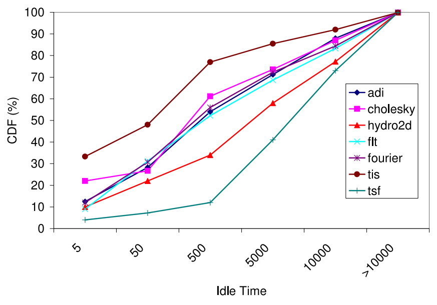


Figure 10: CDF for bank inter-access idle times (accumulated over all banks) when our approach is used.

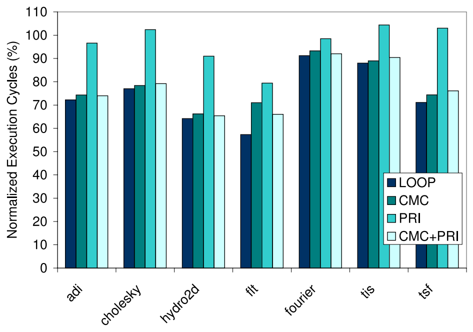


Figure 11: Normalized execution cycles with respected to the baseline scheme.

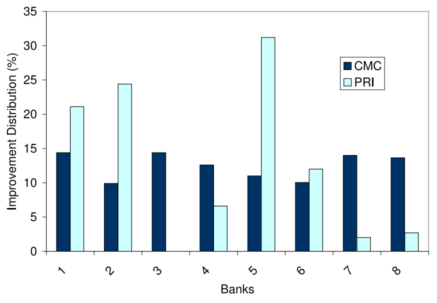
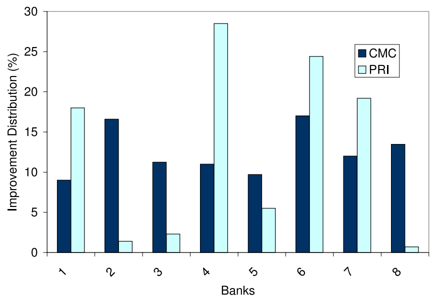



Figure 12: Distribution of energy savings across banks. Left: `cholesky` **Right:** `fourier`**.**

that CMC is very effective in reducing energy consumption. Finally, the last bar, for each benchmark code, in Figure 9 gives the energy consumption with the CMC+PRI scheme (when we combine CMC and PRI). One can see that this combined approach achieves about 37.7% when averaged over all the seven codes evaluated. These results also imply that by miss clustering one can achieve much better results than the conventional code optimizations. To explain where the energy benefits are coming from, we give in Figure 10 the CDF curves for the codes restructured using the CMC scheme. When we compare these curves to the corresponding ones given in Figure 8, we see that our approach increases the length of the bank idle times significantly. These in turn translate to memory energy savings.

It needs to be mentioned at this point that, while both PRI and CMC are very effective in saving energy consumption, they save energy in different ways, as has been discussed earlier. The PRI scheme tries to increase the idleness of each bank in the system independently. As a result, while it may get significant benefits with some benchmarks, it may be unsuccessful with the others. In contrast, our approach works at the data cache level. Therefore, its savings are expected to be distributed across the different banks uniformly, under the assumption that the banks are accessed uniformly. The graphs in Figure 12 give the distribution of energy savings across the eight memory banks for two of our benchmarks, namely, `cholesky` and `fourier`. One can see from these results that the savings with our scheme are uniform across the banks, whereas the variance with the PRI scheme can be very large in some cases. The remaining benchmarks also exhibit similar patterns.

We next look at the performance results given in Figure 11. As expected the LOOP scheme achieves best execution time improvements across all versions, since it is performance oriented. However, the results with the CMC scheme are also very good. Specifically, the execution cycle improvements brought by LOOP and CMC are 25.6% and 21.9%, respectively. This is because our miss clustering approach captures cache locality as well, in addition to bank idleness. That is, clustering data accesses first based on the set of banks they refer enhances data locality at the cache level. Consequently, the results it generates are not very far from those obtained through the LOOP scheme. Our last observation from these results is that the PRI version does *not* improve execution time significantly (3.6% improvement on an average, and increasing original execution times in some cases). The main reason for this poor performance behavior is the fact that this scheme does not consider data locality at all. Therefore, while its results are good from the energy saving perspective, they are not impressive as far as performance is considered. *Overall, when we consider the results given in Figures 9 and 11 together, we can say that considering bank idleness and locality together is a must, if we are to improve both energy and performance.*

We next present the results with varying data cache capacities (all are 4-way associative as in the default configuration). The default cache capacity used in our experiments so far was 16KB. The results with the different cache capacities are given in Figure 13 for the benchmark `cholesky`. The trends observed with the remaining benchmarks were similar; so, we do not present them. One can see from this graph that the PRI scheme is not very sensitive to the data cache capacity used. This is because it operates in a cache oblivious way. However, since a different cache capacity changes the hit/miss behavior, it changes the absolute savings obtained by PRI. In comparison, the effectiveness of CMC and LOOP is reduced with the increased cache capacity. This is expected because the results are given as normalized with respect to the baseline approach without any code optimization, and the behavior of this optimization improves significantly as we increase data cache capacity. In fact, in the extreme case, a very large cache capacity can make any data locality optimization useless. However, we also note that CMC is affected less than LOOP when we increase cache capacity. This is mainly because its energy savings do not come entirely from locality improvement but also from increasing bank idle times through miss clustering. We observed similar trends when we fix the cache capacity at its default value and change the cache associativity. That is, with increased associativity, we saw that the PRI scheme does not exhibit a significant variance. Also, the effectiveness of CMC and LOOP is reduced with the increased associativity. Our next set of experiments study the impact of the off-chip access latency on our results (obtained using the CMC scheme) and are presented in Fig-

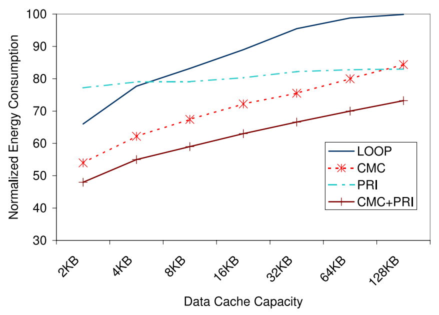


Figure 13: Normalized energy consumptions with varying data cache capacities (`cholesky`).

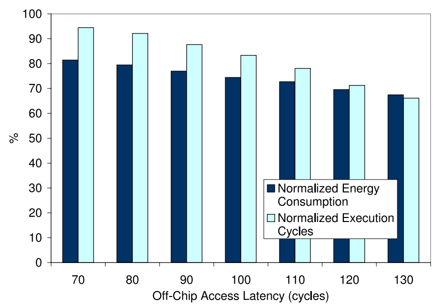


Figure 14: Normalized energy consumptions with varying off-chip latency values (`cholesky`).

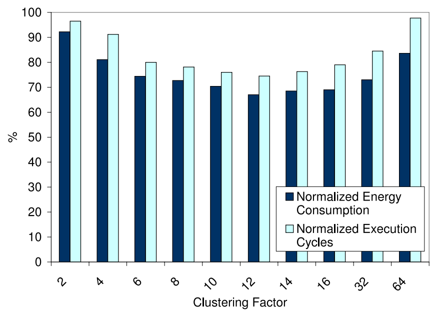


Figure 15: Influence of the clustering factor on energy consumption and execution cycles.

ure 14 for `cholesky`. The default value for off-chip memory access latency was 110 cycles. We see from Figure 14 that the energy results are not very sensitive to the variations in the off-chip access latency, though we abserve a declining trend in normalized energy consumption with increased latency values. However, as expected, the execution cycle savings achieved by our approach reduces more – as compared to the reduction in energy savings – when we reduce the off-chip access latency. We still observe however that, even with an off-chip access latency of 60 cycles, our approach improves execution cycles by 5.56% for this benchmark (on an average 7.27%, when considering all the benchmarks.

Recall that an important parameter in our approach is the clustering factor as defined in Section 4.4. As mentioned earlier, the default value for this parameter used in our experiments so far was 8. We also performed experiments with different values of this parameter. The results are given in Figure 15. In this plot, we give the normalized energy consumption and the normalized execution cycles when averaged over all the benchmarks in our experimental suite. We only present the average results since the results with the different benchmarks are very similar to each other. We observe an interesting trend from this results: both energy consumption and execution cycles are first decreasing as we increase the value of the clustering factor, but beyond a point, they are showing an increasing trend. This can be explained as follows. As mentioned earlier, increasing the value of the clustering factor is good from the viewpoint of clustering cache misses. In a sense, the larger its value, the better miss clustering we have. However, this increase also increases the gap (in terms of the intervening accesses) between the two successive accesses to a given cache line. As a result, when the clustering factor becomes large enough, this can cause extra cache misses, which increase both execution cycles and memory energy consumption.

6. CONCLUDING REMARKS

This paper proposes a cache conscious memory energy reduction scheme based on the concept of cache miss clustering. The idea is to cluster data cache misses and thus cluster banked memory accesses and idle periods. This increase in idle periods then allows better energy management via low-power operation modes and eventually leads to better energy savings. Our experimental analysis shows that our approach is effective in reducing both energy consumption and execution cycles, and, more effective than the previously-proposed schemes. Our results also indicate that the energy savings achieved by our approach are consistent across different cache sizes, cache associativities, and off-chip access latencies.

7. REFERENCES

[1] 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, June 1998.

[2] V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-oriented compiler optimizations for partitioned memory architectures. In Proc. *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems,* 2000.

[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] X. Fan, C. S. Ellis, and A. R. Lebeck. Memory controller policies for DRAM power management. In *Proc. the International Symposium on Low Power Electronics and Design,* 2001.

[5] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J.-A. M. Anderson. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proc. SIGMETRICS,* pages 252–263, 2000.

[6] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer,* December 1996.

[7] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Proc. Ninth International Conference on Architectural Support for Programming Languages and Operating Systems,* 2000.

[8] W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Computer Science Department, Cornell University, Ithaca, NY, 1993.

[9] K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems,* 1996.

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

[11] V. S. Pai and S. V. Adve. Code transformations to improve memory parallelism. In *Proc. the 32nd International Symposium on Microarchitecture,* 1999.

[12] V. S. Pai and S. V. Adve. Comparing and combining read miss clustering and software prefetching. In *Proc. the International Symposium on Parallel Architectures and Compilation Techniques,* 2001.

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

[14] Rambus Inc. http://www.rambus.com/.

[15] M. A. R. Saghir, P. Chow, and C. G. Lee. Exploiting dual data-memory banks in digital signal processors. In *Proc. International Conference on Architectural Support for Programming languages and Operating Systems,* 1996.

[16] U. Sezer, G. Chen, M. Kandemir, H. Saputra, and M. J. Irwin. Exploiting bank locality in multi-bank memories. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems,* 2003.

[17] SimpleScalar LLC. http://www.simplescalar.com/

[18] A. Sudarsanam and S. Malik. Simultaneous reference allocation in code generation for dual data memory banks ASIPs. *ACM Transactions on Design Automation of Electronic Systems* 5, 2000, pp. 242–264.

[19] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. the International Symposium on Computer Architecture,* 2000.