Influence of Loop Optimizations on Energy Consumption of Multi-bank Memory Systems

Mahmut Kandemir[1], Ibrahim Kolcu[2], and Ismail Kadayif[1]

[1] Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802, USA
kandemir@cse.psu.edu
[2] Computation Department, UMIST
Manchester, M60 1QD, UK
i.kolcu@stud.umist.ac.uk

Abstract. It is clear that automatic compiler support for energy optimization can lead to better embedded system implementations with reduced design time and cost. Efficient solutions to energy optimization problems are particularly important for array-dominated applications that spend a significant portion of their energy budget in executing memory-related operations. Recent interest in multi-bank memory architectures and low-power operating modes motivates us to investigate whether current locality-oriented loop-level transformations are suitable from an energy perspective in a multi-bank architecture, and if not, how these transformations can be tuned to take into account the banked nature of the memory structure and the existence of low-power modes. In this paper, we discuss the similarities and conflicts between two complementary objectives, namely, optimizing cache locality and reducing memory system energy, and try to see whether loop transformations developed for the former objective can also be used for the latter. To test our approach, we have implemented bank-conscious versions of three loop transformation techniques (loop fission/fusion, linear loop transformations, and loop tiling) using an experimental compiler infrastructure, and measured the energy benefits using nine array-dominated codes. Our results show that the modified (memory bank-aware) loop transformations result in large energy savings in both cacheless and cache-based systems, and that the execution times of the resulting codes are competitive with those obtained using pure locality-oriented techniques in a cache-based system.

1 Introduction

In programming for many embedded devices, one important aspect is to minimize the energy consumption. As off-chip main memories incur a significant energy and performance penalty when accessed, it is particularly important to perform user and/or compiler level optimizations to reduce energy consumption

R. N. Horspool (Ed.): CC 2002, LNCS 2304, pp. 276–292, 2002.

and improve cache locality (if a cache exists in the system). While the impact of loop-level compiler optimizations on performance is well understood (e.g., see [12] and the references therein), very few studies (e.g., [1]) have tried to address the effect of these transformations on energy consumption. Investigating the energy impact of loop optimizations is important, because this is the first step towards developing energy-oriented compiler optimizations.

Improving memory energy consumption is particularly important in embedded systems that execute image and video processing applications. These applications manipulate large arrays of signals using nested loops, and spend significant portions of their execution time in executing memory-related operations [1]. Large off-chip memories that hold the arrays manipulated by these codes exhibit high per access energy cost (due to long bitlines and wordlines). A recent trend in memory architecture design is to organize the memory as an array of *multiple banks* (e.g., [11]) instead of a more traditional monolithic single-bank architecture. Each bank contains a portion of the address space and can be optimized for energy using an appropriate mix of low-power operating modes. More specifically, a bank not used by the current computation can be placed into a low-power operating mode. Also, using smaller banks help reduce per access energy cost. Recent work has addressed how such low-power operating modes can be managed at software [3,6] and hardware [3] levels. The impact of array placement strategies and two loop optimizations (loop splitting and loop distribution) on a banked off-chip memory architecture has been presented in [2].

The focus of this paper is on reducing the energy consumption of a multi-bank memory system without sacrificing performance significantly. In particular, we focus on array-dominated applications that can be found in domains such as embedded image/video processing and scientific computing, and investigate several loop transformation techniques to see whether they are successful in reducing the memory system energy. We address the problem for both a cacheless system and a system with cache memory. In a cacheless system (which is used commonly in real-time embedded applications), we study the energy impact of classical locality-oriented loop-level techniques and show that slight modifications to them can bring large energy benefits. In a cache-based system, we attempt to modify the data locality-oriented techniques to take into account the banked nature of the off-chip memory. To test our approach, we have implemented bank-conscious versions of three loop transformation techniques (loop fission/fusion, linear loop transformations, and loop tiling) using the SUIF compiler infrastructure [5], and measured the energy benefits using nine array-dominated codes. Our results show that the modified loop transformations result in large energy savings, and that the execution times of the resulting codes are competitive with those obtained using pure locality-oriented techniques.

The rest of this paper is organized as follows. Section 2 introduces the memory architecture assumed, and revises the fundamental concepts related to low-power operating mode management. Section 3 discusses the relationship between cache locality and memory energy consumption. Section 4 discusses the impact of three different loop-level transformations (iteration space tiling, linear loop

transformations, and loop fusion and fission) on memory energy, and explains how these optimizations can be modified to take into account the banked nature of the memory system. Section 5 presents experimental results showing the energy benefits of loop transformations. Section 6 concludes the paper with a summary.

2 Memory Architecture

In this work, we focus on an RDRAM-like off-chip memory architecture [11] 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 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 active (normal operation) mode. However, when the bank is asked to service a memory request, it will take some time for the bank to come alive. The time it takes to switch to active mode (from a low-power mode) is called resynchronization overhead (or reactivation cost). Typically, there is a trade-off between energy saving and resynchronization overhead. That is, a more energy-saving low-power operating mode has also a higher resynchronization overhead. Thus, it is important to select the most appropriate low-power mode to switch to when the bank becomes idle. Note that different banks can be in different low-power modes at a given time.

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 [11] 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 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 resynchronization overheads for these operating modes are given in Figure 1. The energy values shown in this figure have been obtained from the measured current values associated with memory modules documented in memory data sheets (for a 3.3 V, 2.5 nsec cycle time, 8 MB memory) [10]. The resynchronization 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 bank. Obviously, the larger the BIT, the 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 [6], 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

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

Fig. 1. Energy consumptions (per access) and resynchronization times for different operating modes. These are the values used in our experiments

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 whose details are explained in [3]. The prediction mechanism is similar to the mechanisms used in current memory controllers. Specifically, after 10 cycles of idleness, the corresponding bank is put in 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 power-down mode. Whenever the bank is referenced, it is brought back into the active mode incurring the corresponding resynchronization overhead (based on what mode it was in). We focus on a single program environment, and do not consider the existence of a virtual memory system. Exploring the (memory) energy impact of loop transformations in the presence of a virtual address translation is part of our future planned research.

3 Cache Locality vs. Off-Chip Memory Energy

Many optimizing compilers from industry and academia use a suite of techniques for enhancing data locality. Loop transformation techniques [12] are particularly important as there is a well-defined data dependence and loop re-writing (code re-structuring) theory behind them and several efficient implementations exist. Almost all of compiler-based locality-enhancing techniques take some cache specific parameters (e.g., size and associativity) into account and introduce some extra loop overhead and might cause some degradation in the instruction cache performance (as they typically increase code size and reduce instruction reuse). If there exists no cache in the memory hierarchy, it might not be advisable to employ locality-oriented loop transformations as they do not bring any benefit; instead, they increase loop execution overhead. However, if the memory system is partitioned into banks, applying loop transformations still makes sense (i.e., even if there is no cache) as we can cluster loop iterations (through loop transformations) such that the memory accesses in a given time period are localized into a small set of banks. This obviously allows the system to place more banks into low-power operating modes. One of the questions that we try to address in this

paper is to see whether the classical cache locality oriented techniques are also suitable for optimizing off-chip memory energy in a *cacheless* multi-bank memory architecture; and if so, how they can be modified to extract the maximum energy benefits from the memory system.

The existence of a cache memory can, on the other hand, have an important impact on the energy consumption of a banked memory architecture. The cache memory can filter out many memory references and increase the bank inter-access times. This has two major consequences. First, the off-chip memory is accessed less frequently, and therefore consumes less energy. Second, more memory banks can be put in low-power modes and (in some cases) more aggressive low-power modes can be utilized.

If the banked-memory system has a cache memory, selecting a suitable combination and versions of loop-level transformations to apply becomes a much more challenging problem. This is because, two objectives, namely, optimizing cache locality and minimizing off-chip memory energy can sometimes conflict with each other (that is, they may demand different loop transformations and/or different parameters–e.g., tile size and unrolling factor–for the same set of transformations). In this case, one approach would be to optimize cache locality only and not to perform any banked-memory specific transformation. This strategy works fine as long as the cache is able to capture the data access pattern successfully; that is, the vast majority of data references are satisfied from the cache and do not go to off-chip memory. However, if this is not the case, then we need to take care of off-chip references as well. We address this problem by modifying the cache locality optimization strategy to take into account the fact that, for the best off-chip energy behavior, the data accesses should be clustered into a small set of memory banks. More specifically, we modify each type of loop transformation so that it becomes *bank-conscious* (*bank-aware*) as explained in the next section. One way of achieving this is to make sure that the transformed code accesses fewer banks than the original (unoptimized) code (even if all accesses miss the cache) and that the accesses are more clustered than the original code. If this is not possible, then we try not to increase the number of banks that need to be activated (as compared to the original code). In addition to evaluating the impact of loop transformations on the energy behavior of a cacheless memory architecture, this paper also experimentally evaluates two alternative schemes for optimizing energy and locality for a banked memory architecture *with cache*. The first scheme optimizes only for cache locality, and the second scheme tries to strike a balance between enhancing cache locality and reducing off-chip memory energy as explained above.

4 Energy Impact of Loop Transformations

In this section, we discuss how classical loop-based techniques developed for optimizing cache locality affect off-chip memory energy consumption. The conclusions we make here will be supported by experimental evaluation given in Section 5. As mentioned earlier in the paper, the optimizations considered in

this work include loop fusion/fission, iteration space tiling (loop blocking), and linear loop transformations.

4.1 Loop Fusion and Fission

Combining two loops into a single loop is called loop fusion. It is traditionally used to bring array references to the same elements close together [12]. Consider the following example written using a C-like notation, which consists of two separate loops that access the same array `a`. It is easy to see that if the loop limit is sufficiently large that the array does not fit in cache, this code will stream the array `a` from memory through the cache twice (once for each loop).

```
for(i=0;i<L;i++)
 a[i] = i*i + c;
for(i=0;i<L;i++)
 k = k + a[i]*a[i];
```

If this fragment is transformed into the form below, on the other hand, the array needs to be streamed through the cache only once since its contribution to the second assignment can be calculated, while the cache line holding `a[i]` is still cache resident from its use in the first assignment statement. This simple example illustrates that loop fusion can improve cache locality by bringing accesses to the same array closer.

```
for(i=0;i<L;i++)
 {
  a[i] = i*i + c;
  k = k + a[i]*a[i];
 }
```

Unfortunately, the impact of loop fusion on off-chip memory energy is not as clear. If the loop nests to be fused contain extra arrays (i.e., arrays that are not targeted by fusion), these arrays might lead to accesses to a large number of memory banks (some of which would not be accessed if we have not fused the loops). Therefore, in a multi-bank memory architecture, loop fusion should be applied with care. One criterion in applying this optimization is to check whether fusing loops would lead to activation of more banks than individual nests demand.

Loop fission (also known as loop distribution [12]) is the reverse of loop fusion, and places the statements in a given loop into separate loops, each with its own iteration space. One can expect this transformation to be useful from a memory energy viewpoint, in particular, in cases where it separates the references to different arrays, thereby minimizing the number of banks that need to be activated for a given loop.

It is important to note the conflicting objectives of optimizing cache locality and optimizing memory energy when these transformations are employed. In general, when one wants to optimize data cache locality, loop fusion is preferable

whereas loop distribution is generally used to enhance iteration-level parallelism by placing the sinks and sources of data dependences into separate loops. As far as memory energy optimization is concerned, however, loop fission is, in general, preferable as it has the capability of isolating accesses to small set of banks. For example, suppose that a loop nest accesses two different arrays `a` and `b`. Further assume that each array is accessed in a separate statement (in the loop body) and resides in a separate memory bank. If we do not perform loop fission, each iteration of the loop will access both the banks and the BIT (for each bank) will be very small to take any advantage of. If, on the other hand, the loop fission is applied (provided that it is legal), each loop accesses a single bank. Since in this case the BIT for each bank is large, this may present more opportunities for placing banks into low-power modes.

Based on the discussion above, we propose the following strategy for applying loop fusion and fission in a banked-memory environment. If there is no cache in the memory hierarchy, then we do not apply loop fusion; we apply loop fission in such a way that the arrays that share the same set of banks reside within the same loop after fission. If there exists a cache, we do not modify our loop fissioning strategy except that we do not separate statements that contain references to the same array (in an attempt to preserve cache locality). Delaluz et al. [2] present a loop distribution strategy for optimizing off-chip memory energy. As compared to that algorithm, the approach presented here is not based on trying a subset of all possible fissioning alternatives (that is, it finds the solution in one shot), it is integrated with loop fusion, tiling, and loop permutation, and it tries to optimize cache locality and off-chip memory energy consumption in concert.

Note that our fusioning/fissioning strategy tries to strike a balance between two objectives. When applying loop fusion in a cache-based environment, on the other hand, we take cache considerations into account but never fuse two loops if doing so increases the number of banks accessed in a single iteration. For example, suppose that there are three one-dimensional fussable loops in the code, each with one statement within it: `k1 += a[i]+b[i]` in the first loop; `k2 += a[i+1]*b[i-1]` in the second loop; and `k3 += c[i]-b[i]` in the third loop. Also, assume that each array is stored in a separate bank. In this case, while a pure cache locality-oriented approach would fuse all three loops (in conjunction with array padding), our bank-conscious approach would fuse only the first two loops. Note that as in the case of loop fission, this loop fusion scheme also tries to find a balance between conflicting objectives. To sum up, in a cache-based environment, we use cache constraints to restrict loop fission and banked-memory constraints (e.g., minimizing the number of active banks) to restrict loop fusion.

4.2 Loop Tiling

A widely-used technique for improving cache locality is loop tiling [12]. Here, data structures that are too big to fit in the cache are broken up into smaller pieces that will fit in the cache. Consider the following matrix-multiply example.

If the arrays accessed in this nest do not fit in the cache, the cache performance might be poor.

```
for(i=0;i<L;i++)
 for(j=0;j<L;j++)
  for(k=0;k<L;k++)
   c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

If, however, this nest is tiled (blocked) as shown below (assuming that `T` divides `L` evenly, where `T` denotes the tile size), a square-block of array `c` is computed by taking the product of a row-block of `a` with a column-block of `b`. Note that this product consists of a series of sub-matrix multiplies. If these three blocks, one from each matrix, all fit in cache simultaneously, their elements only need to be read in from memory once for each sub-matrix multiply. Thus, the array `a` will now only need to be touched once for each column-block of `c`, and `b` will only need to be touched once for each row-block of `a`. As a result, the memory traffic will be reduced by the size of the blocks.

```
for(ii=0;ii<L;ii=ii+T)
 for(jj=0;jj<L;jj=jj+T)
  for(i=ii;i<ii+T;i++)
   for(j=jj;j<jj+T;j++)
    for(k=0;k<L;k++)
     c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

While this transformation enhances temporal locality across multiple loop levels, it also modifies the array access pattern dramatically. For instance, after the transformation, at a given time, a column-block of array `b` is active. It should be observed that depending on the tile size parameter, a majority of these elements are not consecutive in memory (assuming a row-major memory layout). Consequently, all the banks that hold these elements need to be active during a given short period of time. This is, of course, assuming that the references to these elements will go to off-chip memory and that the array is large enough. If there is a cache memory that captures these references successfully, then the impact of tiling on memory energy is expected to be positive (as it increases the bank inter-access times).

Our bank-aware tiling strategy works as follows. It first determines the loops that carry some form of data reuse as tiling a loop which does not carry any reuse does not improve cache performance but increases loop overhead. We achieve this using the reuse-oriented tiling strategy. Then, among these loops (with data reuse), it selects a subset such that the resulting access pattern does not generate a data tile (i.e., data footprint) on the array space which is *orthogonal* to the storage direction of the array. This is because, under the assumption that elements of a given array are stored consecutively in memory (from the first element to the last element), a data tile orthogonal to the storage direction (of the array) leads to a maximum number of bank activation. For example, in a two-dimensional row-major array case, the bank-aware tiling strategy never selects an iteration space tile shape if it leads to a column-block data tile on the

array space. If possible, it works with only row-block and square tiles. Note that, in the ideal case, one would want to work with only row-block data tiles; but, in many cases, due to data dependences and array access patterns, it may not be possible to obtain only row-block tiles. But, our experience and experiments show that many nested loops can be tiled using only row-block and square tiles. To achieve this, when necessary, linear loop optimizations such as loop permutation can be used prior to tiling. To sum up, our strategy first determines the loops with reuse, filters out the ones with orthogonal footprints (with respect to the storage order), and tile the resulting nest. Our current implementation also tries all permutations of outer nests[1] to obtain row-block and square tiles (i.e., eliminate column-block tiles).

4.3 Linear Loop Transformation

Linear loop transformations that aim at improving cache locality generally try to achieve either of two objectives for each array reference: optimizing temporal locality in the innermost loop or optimizing spatial locality in the innermost loop [7]. Optimizing temporal locality in the innermost loops allows the back-end compiler to place the reference in question into a register (provided that no alias exists). Note that this eliminates accesses to the cache and memory, thereby increasing the memory idle time and creating more opportunities for the employment of low-power operating modes. Optimizing spatial locality (unit stride accesses) is also beneficial from an energy perspective as it allows all the accesses to a given bank to be completed before moving to another bank (provided that the array elements are stored sequentially).

We note that there are cases where linear transformations might be desirable from one objective's angle and not desirable from the other's angle. Consider the following nested loop which accesses a two-dimensional row-major array:

```
for(i=0;i<L;i++)
 for(j=0;j<L;j++)
  a[j][i] = a[j][i]*a[j][i] - 1;
```

Since the column-wise access pattern exhibited by the inner loop here is not suitable from a cache locality perspective, a solution is to interchange the order of the loops. Such an optimization makes the accesses in the inner loop consecutive in memory, and consequently improves data locality. Assuming that array `a` spans multiple banks, the loop interchange here is beneficial from an energy perspective as well (with or without cache). This is because, after the interchange, the array is accessed sequentially; that is, array accesses to a bank are completed before moving to the next bank. However, if we assume that the entire array fits into a single bank, then an energy-oriented optimization strategy would not need to perform any transformation as no transformation would have an effect on the inter-access time of the bank (BIT) in question. However, if there

[1] The innermost loop is determined by linear loop transformations; changing the position of this loop during tiling may not be very beneficial.

is a cache in the system, from a cache locality point of view, it is still desirable to apply loop interchange.

From the discussion above, we can conclude that linear loop transformations might be beneficial even if there is no cache in the banked-memory system. Our bank-conscious linear loop transformation strategy works as follows. If there is no cache in the system, the compiler tries to optimize spatial and temporal locality aggressively. Specifically, it uses the loop transformation framework presented in [7]. However, it does *not* apply a transformation if the transformation will not reduce the number of active banks (at a time) or cluster array accesses (e.g., when the array fits in a single bank). If there is a cache in the system, it tries to optimize locality taking cache characteristics into account, and uses the fact that memory is banked only when it needs to distinguish between references with no cache locality. For example, suppose that a nested loop that manipulates three arrays (`a`, `b`, and `c`) can be optimized for locality in two alternate ways (using linear loop transformations). In the first alternative, arrays `a` and `b` have unit stride accesses, whereas array `c` has no cache locality. In the second alternative, arrays `a` and `c` have unit stride accesses but array `b` has no cache locality. Then, our strategy calculates how many different banks are accessed due to array `c` in the first alternative and due to array `b` in the second alternative. It selects the alternative with the minimum number of banks accessed. We have also experimented with an alternate strategy in which (when multiple optimization alternatives exist) the alternative that leads to the activation of the minimum number of banks (when all array accesses–optimized or unoptimized–are considered) is selected. Our experimental results indicate that for the codes in our experimental suite these two strategies generate very similar results. This is because, in general, the number of banks accessed is determined by the unoptimized array references.

4.4 Discussion

So far we have considered our optimizations in isolation. When we consider the interaction between these optimizations, the problem becomes much harder. In particular, it should be noted that the two objective functions, namely, improving data locality and reducing off-chip memory energy might demand different *combinations* of transformations. Consider the following nested loop which accesses four different arrays:

```
for(i=1;i<L;i++)
 for(j=1;j<L;j++)
  {
   a[i][j] = b[i][j] + 1;
   c[i][j] = d[i][j] - 1;
  }
```

Let us assume that arrays `a` and `b` are stored in one bank, whereas `c` and `d` reside in another bank. A data locality optimization scheme would normally not perform any transformation on this loop, as all the references exhibit high spatial locality and the loop body is not large enough to justify loop distribution (due to

instruction cache locality concerns). A memory energy optimization strategy, on the other hand, will apply loop distribution to isolate the accesses to individual banks so as to maximize the idle periods for each bank. Now, let us assume that all the subscript expressions in the last example above are `[j][i]` instead of `[i][j]` (under the same array placement scheme). In this case, a locality-oriented optimization strategy would apply loop interchange (i.e., changing the order of `i` and `j` loops) to obtain unit stride accesses in the inner loop position. A strategy that targets off-chip memory energy would, however, still use loop distribution. If the underlying architecture contains both a banked memory system and a cache, then it would be best to apply both loop interchange and loop distribution.

We can conclude from this example that the selection of loop transformations to apply depends strongly on the data locality characteristics of the code as well as the array allocation in off-chip memory (i.e., array-to-bank mappings). An important issue then is to combine our loop-based transformations in such a fashion that both the off-chip energy and the cache locality are optimized. However, combining loop-level transformations has not been easy in the past even if one focuses only on specific types of transformations and performance issues [12]. Our heuristic strategy to this problem is as follows. We first apply loop fission to isolate as many nested loops as possible. This will enable the compiler to turn off as many memory banks as possible. After that, we apply bank-conscious version of loop fusion to take advantage of cache memory (if there is one in the system). Then, we consider each of the resulting nests one-by-one, and optimize it using bank-conscious versions of loop permutation (linear transformation) and tiling. Figure 4 shows the overall optimization algorithm. Note that this algorithm calls the algorithms Bank-Conscious-Fusion(.) and Bank-Conscious-Fission(.) in Figures 2 and 3, respectively. Note also that the algorithm in Figure 2 is a greedy heuristic based on the depth of compatibility, similar to the performance-oriented fusioning strategy presented in [8]. It builds a DAG from candidate loops, where edges are dependences between the loops and the weight of each edge is the potential gain due to loop fusion. The nests are partitioned into sets of compatibility at the deepest loop levels possible. Note that the approach first fuses nests with the deepest compatibility and locality. Then, the DAG is updated and the fusion is applied at the next level until all compatible sets are considered. The algorithm in Figure 3, on the other hand, considers each nest one-by-one, and applies loop distribution while being careful in not distoring data locality. In both the algorithms, for a given loop l, Arrays(l) gives the set of arrays accessed by it and Banks(Arrays(l)) gives the set of bankes touched. After applying loop fission and fusion, within the outer for-loop (in Figure 4), each of the nests is optimized using loop permutation and tiling for off-chip memory energy and data locality.

5 Experimental Evaluation

Our loop nest optimizer attempts to improve cache locality and off-chip memory energy consumption by performing high-level transformations on loops. The

```
Bank-Conscious-Fusion(𝒩)
INPUT: 𝒩 = N_1, N_2, ···, N_s, nests that are fusion candidates
ALGORITHM:
 build ℳ = {M_1, ···, M_t} where:
  M_i = {m_i}, a set of compatible nests with depth(M_{i+1}) ≤ depth(M_i);
  build DAG ℋ with dependence edges and weights;
  for each M_i = {m_1, ···, m_p} do
   for k_1 = m_1 to m_p do
    for k_2 = m_2 to k_1 do
     if (no cache memory) then continue;
     else
      if ((there exists locality between k_1 and k_2)
      and (Banks(Arrays(k_1)) == Banks(Arrays(k_2)))
      and (it is legal to fuse k_1 and k_2)) then
       fuse k_1 and k_2 and update ℋ;
      endif
    endfor
   endfor
  endfor
```

Fig. 2. Bank-conscious loop fusion algorithm

```
Bank-Conscious-Fission(𝒩)
INPUT: 𝒩 = N_1, N_2, ···, N_s, nests that are fission candidates
ALGORITHM:
  for each N_i = {n_1, ···, n_k}, where n_js are individual loops in N_i do
   let p_1, ···, p_l be the statements in N_i;
   for each n_j ∈ N_i, j = 1, k
    if (no cache memory) then
     distribute n_j over n_{j+1}, ···, n_k, p_1, ···, p_l
      such that:
      if (Banks(Arrays(p_k)) == Banks(Arrays(p_j))) then
       p_k and p_j stay in the same loop after distribution;
      endif
    else
     apply classical (performance-oriented) loop distribution algorithm
      such that:
      if (Banks(Arrays(p_k)) == Banks(Arrays(p_j))) then
       p_k and p_j stay in the same loop after distribution;
      endif
    endif
   endfor
  endfor
```

Fig. 3. Bank-conscious loop fission (loop distribution) algorithm

current implementation uses only three optimizations (loop permutation, loop fusion/fission, and iteration space tiling) discussed earlier in the paper. The important characteristics of the benchmark codes that we used to measure the energy benefits of loop optimizations are given in Figure 5. `fourier` and `flt` are Fourier transform and digital filtering routines, respectively. `adi` and `cholesky` are ADI and cholesky decomposition codes; `hydro2d` and `nasa7` are array-dominated codes from the Spec Benchmark Suite; and `tis` and `tsf` are from the Perfect Club Benchmarks. Finally, `nwchem` is a kernel routine from a large real-life application that performs computational chemistry-specific calculations.

```
Bank-Conscious-Optimization(𝒩)
INPUT: 𝒩 = N_1, N_2, ···, N_s, nests in the procedure
ALGORITHM:
Bank-Conscious-Fission(𝒩);
Bank-Conscious-Fusion(𝒩);
 for each N_i = {n_1, ···, n_k}, where n_js are individual loops in N_i do
  best-cost = ∞;
  best-permutation = none;
  determine permutations of n_1, ···, n_k with the best locality;
  let P_1, ···, P_f be such permutations;
   for each P_i, i = 1, f do
    current-cost = find the number of banks accessed by the arrays
     with no locality;
    if (current-cost < best-cost) then
     best-cost = current-cost;
     best-permutation = P_i;
    endif
   endfor
  determine the set S_i, the loops with reuse in P_i;
  if (there is a cache in the system) then
   tile each loop s_j ∈ S_i if its data footprint is not orthogonal
    to storage direction;
  endif
 endfor
```

Fig. 4. Bank-conscious energy optimization algorithm

The third column in Figure 5 gives the total dataset size manipulated by the corresponding code. BaseE- and BaseE+ correspond to *base* energy values (without any loop optimizations) for a cacheless system and for a system with a 32KB two-way set-associative cache (with a block size of 32 bytes), respectively. Note that these base energy values have been obtained using the original codes and exploiting low-power operating modes to save energy (as explained in the second section). In other words, our base version already takes advantage of the low-power operating modes. Also, these energy numbers include the energy consumed in off-chip memory (due to data accesses only) and the energy consumed in the data cache (when it exists). BaseT- and BaseT+ are the corresponding *base* execution times. The last three columns indicate whether a given benchmark is amenable to a specific optimization. All energy numbers given in Section 5.1 (resp. Section 5.2) are percentage improvements over the corresponding entry in the BaseE- (resp. BaseE+) column. All the energy numbers given in Figure 5 are in microjoules and have been obtained using a default memory bank configuration which contains eight 8MB banks (denoted 8×8MB). All performance numbers are in seconds.

5.1 Cacheless System

Figure 6 gives the percentage energy improvements for a cacheless system for four different versions. c-opt1, c-opt2, and c-opt3 denote the optimized versions assuming an *imaginary* cache architecture of 8KB, 16KB, and 32KB, respectively (All caches are two-way set-associative with a block size of 32 bytes). The objective in measuring the energy behavior of these versions is to see whether we can

Benchmark Name	Number of Lines	Input Size	Base Energy		Base Performance		Optimization Applicability		
			BaseE-	BaseE+	BaseT-	BaseT+	fusion+fission	tiling	linear
adi	56	78MB	28.9	19.3	5.76	3.92		√	√
cholesky	34	61MB	88.2	61.1	9.68	7.10	√		√
hydro2d	52	44MB	104.0	76.3	10.02	6.59		√	√
flt	85	51MB	723.3	328.1	16.81	11.57		√	√
fourier	167	57MB	634.0	411.7	11.96	8.90		√	√
nasa7	1,105	54MB	1,418.6	783.2	29.77	18.52	√	√	√
nwchem	370	44MB	780.5	408.9	13.95	8.16	√	√	√
tis	485	56MB	899.8	511.0	18.72	12.04	√	√	√
tsf	1,986	60MB	1,066.2	620.4	24.83	16.71	√	√	√

Fig. 5. Benchmark codes and their important characteristics

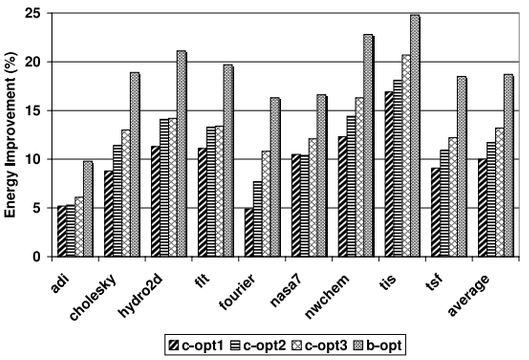

Fig. 6. Energy improvements in a cacheless system

use a cache locality-oriented scheme without modification for optimizing memory energy of a banked system *without* cache. The b-opt version, on the other hand, denotes a version that uses loop transformations solely for optimizing memory energy (i.e., the bank-aware version). We observe two important trends from these results. First, as the *assumed* cache size is increased, the energy benefits also increase. This is because with larger caches, the locality-oriented strategy becomes less aggressive, and performs fewer cache-specific optimizations. This, in turn, causes less side effects on the memory energy consumption. Second, in a cacheless system, customizing loop optimizations taking into account the banked nature of the memory makes sense as it improves energy 18.72% on average (as compared to 13.20% for c-opt2). We need to mention that increasing the assumed cache size further did *not* bring any additional improvement over c-opt3 (except for `tis`, where an assumed data cache size of 64KB reduced the memory energy by 2.8% over the c-opt3 version). Our experiments with different bank configurations also showed similar trends.

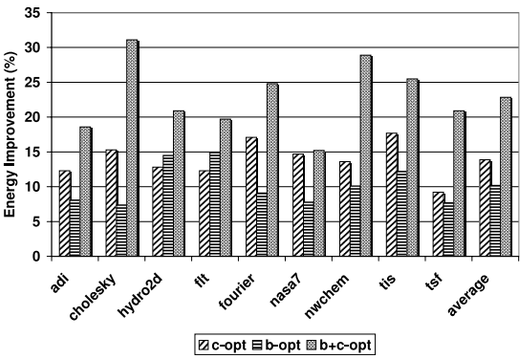

Fig. 7. Energy improvements for a memory system with cache

5.2 Memory System with Cache

Figure 7 presents the percentage energy improvements for three different versions for a banked memory system with a 32KB two-way set-associative cache memory. c-opt is the version that optimizes only for cache memory and b-opt optimizes only for memory energy. The b+c-opt version, on the other hand, tries to strike a balance between the two objectives (optimizing cache locality and reducing off-chip memory energy). We can observe from this figure that, in general, c-opt generates better results than b-opt. That is, if there is a cache in the banked-memory system, it is not a good idea to use optimizations that target only memory energy. Using pure locality-based optimizations results in better energy savings for most of the time. However, we also observe that the b+c-opt version generates the best result across all applications (averaging a 22.84% overall energy improvement). Although not presented here due to lack of space, we observed similar trends in experiments performed using different cache sizes and associativities.

5.3 Performance Gains

Figure 8 gives the performance benefits (over the values given under the column BaseT+ in Figure 5) of three different versions (b-opt, c-opt, and b+c-opt) for a banked memory system with a 32KB two-way set-associative cache memory. We observe that the b+c-opt version generates comparable results to the c-opt version (pure locality-oriented approach). The difference between them is only 1.80%. Therefore, we can conclude that the combined optimization strategy is almost as good as the pure cache locality-oriented approach in improving the performance, but it leads to significantly more (memory system) energy savings than a pure locality-oriented approach.

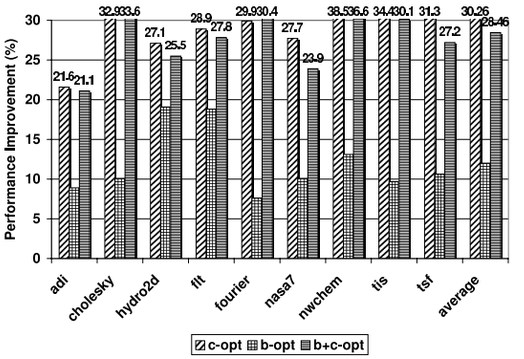

Fig. 8. Percentage performance gains in a cache based system

6 Conclusions

In this paper, we investigate the influence of three different types of loop transformation techniques on memory system energy assuming a multi-bank memory architecture. A multi-bank memory system allows unused banks to be transitioned to low-power operating modes. In a multi-bank memory system without cache, we have found that slightly modified versions of classical locality-oriented loop transformation techniques generate large energy savings. In a cache-based multi-bank system, our results show that the modified (bank-aware) loop transformations result in large energy savings, and that the execution times of the resulting codes are competitive with those obtained using pure locality-oriented techniques.

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. 277
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,* November 2000. 277, 278, 282
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. 277, 278, 279
4. DSPstone Benchmark Suite. http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html.
5. 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. 277

6. 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,* November 2000. 277, 278
7. W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Computer Science Department, Cornell University, Ithaca, NY, 1993. 284, 285
8. K. McKinley, S. Carr, and C. W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems,* 1996. 286
9. 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.
10. Rambus Inc. http://www.rambus.com/. 278
11. 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., May 1999. 277, 278
12. M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996. 277, 279, 281, 282, 286