EXPLOITING BANK LOCALITY IN MULTI-BANK MEMORIES

U. Sezer
Department of Electrical & Computer Engineering
University of Wisconsin-Madison
sezer@ece.wisc.edu

G. Chen, M. Kandemir, H. Saputra, M. J. Irwin
Department of Computer Science & Engineering
The Pennsylvania State University
{guilchen,kandemir,saputra,mji}@cse.psu.edu

ABSTRACT
Bank locality can be defined as localizing the number of load/store accesses to a small set of memory banks at a given time. An optimizing compiler can modify a given input code to improve its bank locality. There are several practical advantages of enhancing bank locality, the most important of which is reduced memory energy consumption. Recent trends indicate that energy consumption is fast becoming a first-order design parameter as processor-based systems continue to become more complex and multi-functional. Off-chip memory energy consumption in particular can be a limiting factor in many embedded system designs. This paper presents a novel compiler-based strategy for maximizing the benefits of low-power operating modes available in some recent DRAM-based multi-bank memory systems. In this strategy, the compiler uses linear algebra to represent and optimize bank locality in a mathematical framework. We discuss that exploiting bank locality can be cast as loop (iteration space) and array layout (data space) transformations. We also present experimental data showing the effectiveness of our optimization strategy. Our results show that exploiting bank locality can result in large energy savings.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *Compilers.*

General Terms
Algorithms, Experimentation

Keywords
Bank Locality, Energy Consumption, Embedded Systems, Affine Program, Optimizing Compilers

1. INTRODUCTION AND MOTIVATION
Compared with application-specific custom circuitry, many processor-based systems can exhibit a factor of 100-1000 worse energy-delay product. Consequently, many research groups are trying to reduce this gap by re-examining the hardware-software interface, and designing hardware and software level solutions for reducing energy consumption. The role of power and energy consumption in real-time and embedded systems is well recognized and becoming an increasingly important area of research.

In many system designs, large off-chip main memories can be a major energy consumer [6, 2]. There are two main reasons for high dynamic energy consumption of off-chip data memories. First, off-chip data memories present a large capacitive load to the rest of the system. Second, most data-intensive embedded applications exhibit poor data reuse (cache locality), resulting in frequent visits to the main memory. Consequently, hardware and software techniques that target at reducing memory energy are critical in many embedded designs.

One approach to attack this problem from the hardware perspective is to design a banked memory architecture, where the main memory is partitioned into multiple banks [17]. Since this effectively reduces the size of the memory accessed during a reference, it also helps to reduce per access dynamic energy consumption. In addition, many new banked memory architectures provide multiple low-power operating modes. Typically, an idle bank (i.e., a bank not used by the current computation) can be placed into a low-power operating mode, increasing energy savings further. An effective use of such low-power modes is critical for obtaining the best energy behavior, and has received some attention recently [12, 3, 10, 5].

In this paper, we attack this important problem using a mathematical framework embedded into an optimizing compiler. Specifically, we present a compiler-based strategy for maximizing the benefits of low-power operating modes available in DRAM architectures. In this strategy, an optimizing compiler employs loop and data transformations to increase bank inter-access times, thereby allowing a memory bank to remain in the idle state for longer periods of time. This is achieved by clustering accesses with temporal affinity in a small set of banks. In most cases, this has a great impact on memory energy consumption as a longer bank inter-access time means a more aggressive (i.e., more energy saving) low-power mode. In other words, unlike most existing compiler frameworks that use loop/data transformations for either cache locality and/or loop-level parallelism [21, 14, 22], we use these transformations for enhancing bank locality to improve energy behavior of banked memory architectures.

To test the effectiveness of the proposed compiler-based energy optimization strategy, we implemented it in a source-to-source translator [1] and evaluated its performance using five array-intensive embedded applications. Our experimental results

CASES '03, Oct. 30 – Nov. 2, 2003, San Jose, California, USA.

indicate that the proposed strategy is very successful in practice and improves bank locality significantly. As a result, it reduces off-chip memory energy consumption substantially over a strategy that uses low-power operating modes without our optimization. Our results also demonstrate that these energy improvements are consistent across different systems/software parameters (e.g., the number of banks, array layouts). In addition, the impact of our approach on execution cycles was found to be negligible. Based on these results, we strongly encourage compiler writers for embedded systems that employ banked memories to focus their loop/data optimizations on memory energy.

The rest of this paper is organized as follows. Section 2 describes low-power operating modes and makes an informal introduction to memory bank locality. Section 3 gives formal definition of array mappings to virtual banks. Section 4 presents intra-reference and inter-reference constraints for bank locality. Section 5 demonstrates how loop (iteration space) transformations can be used for improving bank locality. Section 6 discusses how data (memory layout) transformations can be used for implementing array decompositions across banks. Section 7 describes global optimizations, and virtual-to-physical bank mappings are discussed in Section 8. Section 9 presents our experimental results, and Section 10 concludes the paper by summarizing our major contributions and gives a brief introduction to our future work.

2. BANKED MEMORY ARCHITECTURE AND LOW-POWER OPERATING MODES

We focus on an RDRAM-like off-chip memory architecture [17], where off-chip memory is partitioned into several (equally-sized) banks. We also assume that each bank can be placed into a low-power operating mode (independently) when it is not in active use. In a low-power operating mode, a bank typically consumes much less energy than it would consume when it is in active (i.e., fully-operational) mode. However, when a bank in low-power mode is accessed, it takes some amount of time until it is available to service the request. This time delay is called re-synchronization penalty (or re-synchronization cost), and is the major factor that prevents us from using the most aggressive (i.e., the most energy-saving) low-power mode as soon as the bank becomes idle. Typically, given a set of low-power operating modes, there is a tradeoff between energy saving and re-synchronization latency. In other words, the most energy-saving low-power mode has also the highest re-synchronization penalty of all. Therefore, one should be careful in selecting the suitable low-power mode when a memory bank becomes idle. Figure 1 lists energy consumptions and re-synchronization costs for the operating modes used in this study. The energy values shown in this figure have been obtained from the measured current values associated with memory banks documented in memory data sheets (for 3.3V, 2.5 nsec cycle time, 8MB memory) [17]. The re-synchronization latencies have also been obtained from the same data sheets. Based on the trends gleaned from data sheets, the energy values are increased by 30% when bank size is doubled. Note, however, that our approach is general enough in that it can accommodate any number of low-power operating modes with different energy consumptions and re-synchronization costs.

	Energy Consumption (nJ)	**Resynchronization Cost (cycles)**
active	3.570	0
napping	0.320	30
power-down	0.005	9,000

Figure 1. Bank operating modes used in this work

Prior research shows that compiler-based [3, 10], OS-based [12], and pure hardware-based schemes [3] can be adopted to decide the most suitable low-power mode to use when a memory bank is detected to be idle. Since in this work we focus on array-intensive embedded applications, we opted to use a compiler-based approach, where an optimizing compiler (taking into account loop access patterns and array-to-bank mappings – that is, the layout of data in banked memory) decides which operating mode to use. Note that (where applicable) compiler-based strategy has an important advantage over OS-based and hardware-based techniques. The compiler-based strategy (unlike OS or pure hardware-based strategies) does not rely on history information; that is, the compiler can predict (quite accurately for array-based embedded codes) the future access patterns (and also, the future idle times), and select the most appropriate operating mode to use when idleness is predicted. In addition, the compiler can also predict when an idle bank will be requested, and can pre-activate the bank in question in an attempt to eliminate re-synchronization latency. The details of the compiler-based low-power mode detection strategy are beyond the scope of this paper and can be found in [3, 10]. [11] evaluates the impact of classical loop optimizations on energy consumption of banked memories. [4] studies the impact of array regrouping, a data optimization, on energy-efficiency of banked memory systems. In contrast, the work presented in this paper is oriented toward increasing the benefits of low-power modes by loop transformations and data distributions across memory banks. [7] shows how a sleep mode can be exploited for memory partitions. [19] and [18] discuss techniques for exploiting dual banks for ASIPs and DSPs, respectively. [16] addresses the problem of incorporating the application-specific customization of memory bank configuration into behavioral synthesis. In comparison, we study how compiler-directed data decomposition and loop optimizations can improve energy behavior of a multi-banked system.

While a compiler-based low-power mode management strategy can lead to large energy savings in memory, it is possible to increase these savings even further by increasing the bank inter-access times (that is, the duration between two successive accesses to a given memory bank). One way of achieving this is to modify the program access pattern such that when a bank is accessed, it is used as much as possible before moving to another bank. Programs that exhibit such bank access patterns are said to have bank locality. We demonstrate in this paper that, for array-intensive embedded applications, loop (iteration space) transformations can be used for improving bank locality.

It should be stressed that all the results presented in this paper (whether optimized for bank locality or not) have been obtained using the same compiler-based low-power mode detection/ activation strategy. That is, the benefits associated with the optimized codes come solely from our optimizations, not from a specific low-power mode selection policy. We also need to mention that in this work we assume that no virtual memory support exists in the system under consideration. Consequently, the compiler can directly work with physical addresses; that is, it can layout data in physical memory and place banks into low-power modes based on the information it collected during program analysis. Note that there exist many embedded systems that work without a virtual memory support [8]. Work is in progress to extend the techniques discussed in this paper to environments with virtual memory (by enlisting help from OS).

3. AFFINE MAPPINGS OF ARRAYS TO BANKS

We assume an affine program, where both array references and loop bounds are affine functions of enclosing loop indices and loop-invariant unknowns. Many embedded image/video-processing codes fit into this description [2]. We represent each iteration of a given n-dimensional loop nest (i.e., a nest that contains n loops) using an n-entry vector (called iteration vector), where the i^{th} element from top refers to the value of the i^{th} loop index. Then, an iteration space, which is a collection of loop iterations, defines a polytope in an n-dimensional space. In a similar way, we can view data space (that is, the set of all possible array indices) of an m-dimensional array as a rectilinear polytope, where each point (called index vector) corresponds to an array element. An affine mapping from an iteration space to an array space corresponds to an array reference, and can be represented by the affine function $\varphi(I) = \theta I+\delta$, where I is the iteration vector (n-entry), θ is an m-by-n linear access matrix and δ is an offset (displacement) vector. For example, a given array reference U[i+1][j-2] in a two-dimensional nest (with loops i and j) has an identity access matrix and a displacement vector of $[1\ \ -2]^T$. It is to be noted that the iteration vector here is $[i\ \ j]^T$.

Based on these definitions, we can define an affine mapping of array elements to memory banks using an affine function $\Omega(d) = \Pi d+\pi$, where d is an array index vector, Π is a b-by-m linear mapping matrix (called bank decomposition matrix or array decomposition matrix), and π is a b-entry offset (displacement) vector. In this paper, we refer to Ω as the bank decomposition function. Based on this, the array-to-bank assignment problem can be cast as one of determining an affine mapping Ω for each array involved so as the bank locality is improved as much as possible.

An important issue here is to understand what an affine bank decomposition function represents. A bank decomposition function Ω indicates how each array element is mapped to a virtual bank architecture. A virtual bank architecture is a very fine-granular banked-memory architecture. Each bank in this architecture is represented using a b-entry vector (in other words, the bank space is b-dimensional). A bank decomposition function maps an array element into a virtual bank from the virtual bank architecture (also called virtual bank space). Two array elements are stored in the same virtual bank if and only if the corresponding index vectors result in the same output (i.e., the same virtual bank) when given to the bank decomposition function. In mathematical terms, let d1 and d2 refer to (the index vectors of) two array elements. They map to the same virtual bank (that is, they are co-located) if and only if:

$$\Omega(d_1) = \Omega(d_2).$$

Assuming, $\Omega(d) = \Pi d+\pi$, the above condition implies that $\Pi d_1+\pi= \Pi d_2+\pi$, which in turn indicates that $\Pi(d_1 - d_2) = 0$; that is, $(d_1 - d_2) \in \mathrm{Ker}\{\Pi\}$. In other words, these two array elements will be mapped to the same virtual bank if and only if the difference between them is in the kernel set (null set) of the bank decomposition matrix[1]. As an example, if $\Pi = [1\ 0]$, then $d_1 = [3\ \ 5]^T$ and $d_2 = [3\ \ 7]^T$ will be mapped onto the same virtual bank. However, $d_1 = [3\ \ 5]^T$ and $d_2 = [4\ \ 5]^T$ will not be mapped to the same bank under the same Π. A trivial bank decomposition matrix is a zero matrix that maps all elements to the same bank. However, such a decomposition matrix does not exploit any bank structure and, consequently, is hardly useful for our objective (which is optimizing memory energy consumption).

Before going into more technical discussion about bank decompositions and loop and data transformations, let us briefly mention three important issues. First, if two array elements of an array are accessed at similar times (during loop execution) – e.g., during the same or successive loop iterations – it is important that they map on the same bank. This is because (as discussed earlier) such a mapping allows us to take advantage of low-power operating modes better. As we will show in this paper (Section 5), this can be achieved, to a large extent, by employing loop transformations. Second, co-locating some array elements means (in some cases) that we need to separate some array elements (in physical memory) that are consecutive in logical memory (i.e., from the viewpoint of programmer/compiler). This can create problems in addressing such array elements. We show in Section 6 that data (array layout) transformations can be used to ensure proper addressing of array elements. Third, we intentionally keep our bank decomposition matrices very general. That is, Π can represent any matrix, meaning that we can perform mappings to multi-dimensional bank spaces (if desired). This flexibility allows us to explore the impact of compiler-level loop and data transformations on a wide variety of banked memory architectures. It also promotes modularity within the compiler in the following sense. When we determine loop and data transformations and bank decomposition functions, we can work with a multi-dimensional decomposition matrix (if we want to). Later, when we are to map our array data to a given banked architecture, we can use a folding function that eliminates some virtual banks (i.e., fold them into physical banks) in a systematic way. In other words, a folding function maps a virtual bank architecture (space) to a physical bank architecture (space). In doing so, some array elements that are not co-located in the

[1] A vector is said to be in the kernel set of a matrix if its dot product with every row of that matrix is 0.

virtual space can be co-located in the physical space. Section 8 discusses our folding functions in more detail.

4. Constraints for Bank Locality

Given a pair of references ($\varphi_1(I) = \theta_1 I+\delta_1$ and $\varphi_2(I) = \theta_2 I+\delta_2$) to the same array, in order to ensure accesses with spatial locality to the memory banks (i.e., bank locality), we need two types of constraints: intra-reference constrains and inter-reference constraints.

An intra-reference constraint originates from a single reference and indicates the condition that the reference in question should access the same memory bank in two consecutive loop iterations. For example, assuming zero displacement vectors (π), in order for $\varphi 1$ to exhibit bank locality,

$$\Pi(\theta_1 I+\delta_1) = \Pi(\theta_1 I^{+}+\delta 1)$$

should be satisfied. In this expression, I^{+} denotes the loop iteration that immediately follows I. For simplicity, we assume that when we move from I to I^{+}, no loop boundary (upper bound) is crossed. From this constraint, we can obtain $\Pi\theta_1(I^{+}-I) = 0$, which means that

$$\Pi\rho_1 = 0 \Rightarrow \Pi \in \mathrm{Ker}\{\rho_1\},$$

where ρ_1 is the last column of θ_1. Note that a similar condition must be satisfied for the second reference as well (that is, $\Pi \in \mathrm{Ker}\{\rho_2\}$).

An inter-reference constraint, on the other hand, implies that the data elements accessed by a given iteration through two references should be on the same bank. In mathematical terms, one should have

$$\Pi(\theta_1 I+\delta_1) = \Pi(\theta_2 I+\delta_2).$$

From this constraint, we can derive that $\Pi(\theta_1-\theta_2)I = \Pi(\delta_1-\delta_2)$. In this case, if $\theta_1=\theta_2$ then we have

$$\Pi (\delta_1-\delta_2) = 0 \Rightarrow \Pi \in \mathrm{Ker}\{\delta_1-\delta_2\}.$$

On the other hand, if $\theta_1=\theta_2$ does not hold true, it is not possible to simplify the expression above further.

It should be noted that even if we assume that $\theta_1=\theta_2$, requiring that $\Pi \in \mathrm{Ker}\{\rho_1\}$, $\Pi \in \mathrm{Ker}\{\rho_2\}$, and $\Pi \in \mathrm{Ker}\{\delta_1-\delta_2\}$ are satisfied at the same time implies that ρ_1, ρ_2, and $\delta_1-\delta_2$ should be linearly dependent to each other, which may or may not be true. Of course, since in general $\theta_1=\theta_2$ may not be hold at all, finding a suitable Π that satisfies both intra-reference constraint and inter-reference constraint becomes much harder. In the next section, we show how loop transformations can help in finding a suitable bank assignment matrix such that both intra-reference and inter-reference bank locality constraints are satisfied.

5. Loop Transformations for Bank Locality

Previous research used many loop transformations for improving cache locality and parallelism (e.g., see [22] and the references therein). In this section, we demonstrate how loop transformations can also be used for enhancing bank locality (and thus for increasing bank inter-access times). The idea behind this approach is to select a loop transformation matrix such that the chances for satisfying both intra-reference and inter-reference constraints (after the transformation) are increased.

A linear loop transformation, also called iteration space mapping, for a given n-dimensional nest (i.e., a nest that contains n loops) can be represented using an n-by-n linear matrix Λ. This linear matrix maps each iteration vector in the original nest to a new vector in the transformed nest [22]. For example, if $I = [i \;\; j]^T$ is the original iteration vector, the loop transformation matrix

$$\Lambda = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

results in the new (transformed) iteration vector $I' = [\, i' \;\; j']^T = \Lambda [i \;\; j]^T = [j \;\; i]^T$. In other words, such a loop transformation maps an original loop iteration $[i \;\; j]^T$ to a new loop iteration $[j \;\; i]^T$. The issues related to code generation after a loop transformation (that is, re-writing loop body as well as re-computing loop bounds) have been studied extensively by prior research and can be found elsewhere [14, 22].

Let us first focus on a single nest (which might contain multiple loops) and a single array. In Section 7, we generalize our approach to multiple arrays and multiple nests. Suppose that we apply a loop transformation represented by Λ to a nest that contains two references φ_1 and φ_2 to the same array. Let us use I' to denote a loop iteration after the transformation (i.e., $I' = \Lambda I$ and $I = \Lambda^{-1} I'$). We can now write the intra-reference constraint as

$$\Pi(\theta_1\Lambda^{-1}I'+\delta_1) = \Pi(\theta_1\Lambda^{-1}I'^{+}+\delta_1).$$

Consequently, after the transformation, we obtain $\Pi\theta_1\Lambda^{-1} (I'^{+} - I') = 0$, which means that $\Pi\theta_1 q = 0 \Rightarrow \Pi \in \mathrm{Ker}\{\theta_1 q\}$, where q is the last column of the inverse of the loop transformation matrix (that is, the last column of Λ^{-1}). What this expression means is that by changing q (i.e., by selecting different q columns), we can obtain different bank decomposition matrices.

We can also re-write the inter-reference constraints after the loop transformation. Specifically, assuming $I' = \Lambda I$, we have

$$\Pi(\theta_1\Lambda^{-1}I'+\delta_1) = \Pi(\theta_2\Lambda^{-1}I'+\delta_2).$$

From this last constraint, we can derive that $\Pi(\theta_1-\theta_2)\Lambda\text{-}1I' = \Pi(\delta_2-\delta_1)$. So, if $\theta_1=\theta_2$ then we have $\Pi (\delta_2-\delta_1) = 0 \Rightarrow \Pi \in \mathrm{Ker}\{\delta_2-\delta_1\}$. However, now even if $\theta_1=\theta_2$ does not hold, if we can select a Λ^{-1} from the kernel set of $(\theta_1-\theta_2)$, we make sure that we can satisfy the inter-reference constraint by ensuring $\Pi \in \mathrm{Ker}\{\delta_2-\delta_1\}$. In other words, by selecting a suitable loop transformation matrix, we can increase the chances that our selection of bank decomposition matrix is constrained only by $\delta 1$ and δ_2. Consequently, our approach to determining a loop transformation matrix (Λ) and a bank decomposition matrix (Π) for satisfying both intra-reference and inter-reference constraints for a given loop nest performs the following two tasks:

- Select a loop transformation matrix Λ from the kernel set of $(\theta_1-\theta_2)$.
- Select a bank transformation matrix Π that satisfies $\Pi \in \mathrm{Ker}\{\theta_1 q\}$, $\Pi \in \mathrm{Ker}\{\theta_2 q\}$, and $\Pi \in \mathrm{Ker}\{\delta_2-\delta_1\}$.

The most important point to note here is that the chances of satisfying the second constraint here are higher than the case without loop transformations. This is because, in this case, we might be able to select a suitable q such that $\{\theta_1 q\}$, $\{\theta_2 q\}$, and $\{\delta_2\text{-}\delta_1\}$ become linear-dependent on each other. In the following discussion, we study this issue in more detail.

We first consider a special case where $\theta_1=\theta_2$. This case frequently occurs in many array-intensive applications from embedded image/video processing (especially in those that perform stencil-types of computations) [2]. In this case, we are left with two constraints that need to be satisfied: $\Pi\theta_1 q = 0$ and $\Pi(\delta_2\text{-}\delta_1) = 0$. Therefore, one can first determine Π from the second equation and then use it in solving the first equation for q. After determining q, it can be completed to a full Λ using, for example, the dependence-aware matrix completion strategy by [14]. This discussion clearly indicates that when $\theta_1=\theta_2$ holds, the loop transformation enables us satisfy both intra-reference and inter-reference constraints.

If $\theta_1=\theta_2$ does not hold true, this means that we need to satisfy these three constraints: $\Pi\theta_1 q = 0$, $\Pi\theta_2 q = 0$, and $\Pi(\theta_1\text{-}\theta_2)\Lambda^{-1}\Gamma' = \Pi(\delta_2\text{-}\delta_1)$. In this case, we first try to find a Λ^{-1} such that $(\theta_1\text{-}\theta_2)\Lambda^{-1} = 0$. If such a Λ^{-1} can be found, then we try to select a bank decomposition matrix Π such that, these three constraints are satisfied: $\Pi\theta_1 q = 0$, $\Pi\theta_2 q = 0$, and $\Pi(\delta_2\text{-}\delta_1) = 0$. If, on the other hand, we cannot find a Λ^{-1} that satisfies $(\theta_1\text{-}\theta_2)\Lambda^{-1} = 0$, then there is no point in trying to satisfy $\Pi(\delta_2\text{-}\delta_1)=0$, and so, we try to select a Π such that only $\Pi\theta_1 q = 0$ and $\Pi\theta_2 q = 0$ are satisfied. This means that if we cannot find a Λ^{-1} that satisfies $(\theta_1\text{-}\theta_2)\Lambda^{-1} = 0$, we sacrifice the inter-reference locality (i.e., we do not try to satisfy the inter-reference constraint).

Example*:* Consider the following loop nest that contains two loops and two references to the same two-dimensional array (U):

```
for i = 1, N
  for j = 1, N-1
    …U[i][j] …U[i-1][j+1] …
```

Let us first assume that we will not use any loop transformation, and check whether it is possible to come up with a data distribution (decomposition) strategy that satisfies both intra-reference and inter-reference bank locality constraints. In this example, we have $\theta_1=\theta_2$ and $\rho_1 = \rho_2 = [0 \;\; 1]^T$. In addition, $\delta_1 = [0 \;\; 0]^T$ and $\delta_2 = [-1 \;\; 1]^T$. Since we need to come up with a Π to satisfy both $\Pi\rho_1 = 0$ and $\Pi(\delta_1\text{-}\ \delta_2) =0$, we should have both

$$\Pi\begin{bmatrix}0\\1\end{bmatrix}=0 \qquad \text{and} \qquad \Pi\begin{bmatrix}-1\\1\end{bmatrix}=0$$

at the same time. Since it is not possible to select a non-zero Π to satisfy both of these equations (since δ_1 and δ_2 are linearly independent), we can conclude that it is not possible to exploit intra-reference and inter-reference bank locality at the same time.

On the other hand, if we use a loop transformation Λ, we need to satisfy $\Pi\theta_1 q = 0$ and $\Pi(\delta_1\text{-}\ \delta_2) =0$. From the second constraint, we can determine $\Pi = [1 \quad 1]$. By substituting this in the first constraint, we have

$$[1 \quad 1]\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}q = 0,$$

which means $q = [1 \quad -1]^T$. By completing this to a full matrix, we obtain

$$\Lambda^{-1} = \begin{bmatrix}1 & 1\\0 & -1\end{bmatrix}.$$

Consequently, assuming that there are no data dependences in the loop nest, the transformed array references are U[i'+j'][-j'] and U[i'+j'-1][1-j']. Now, let us check the intra-reference and inter-reference bank locality constraints. Since

$$[1 \quad 1]\begin{bmatrix}i'+j'\\-j'\end{bmatrix} = [1 \quad 1]\begin{bmatrix}i'+j'-1\\1-j'\end{bmatrix} = i',$$

the inter-reference constraint is satisfied. Similarly, since

$$[1 \quad 1]\begin{bmatrix}i'+j'\\-j'\end{bmatrix} = [1 \quad 1]\begin{bmatrix}i'+(j'+1)\\-(j'+1)\end{bmatrix} = i' \quad \text{and}$$

$$[1 \quad 1]\begin{bmatrix}i'+j'-1\\1-j'\end{bmatrix} = [1 \quad 1]\begin{bmatrix}i'+(j'+1)-1\\1-(j'+1)\end{bmatrix} = i',$$

the intra-reference constraints (one per array reference) are also satisfied. Therefore, we can conclude that using loop transformation (Λ) can enable the compiler to exploit memory bank locality fully (which was not possible without loop transformation).

The results presented above can be extended to cases where we have more than two references to the same array in a given loop nest. Specifically, let us assume that we have v references to the same array, namely, φ_1, φ_2, …., φ_v. In this case, if $\theta_1=\theta_2=\ldots=\theta_v$ holds, we need to satisfy the following constraints: $\Pi\theta 1q = 0$ and $\Pi(\delta_j\text{-}\delta_i) = 0$, where $1 \le i < j \le v$. This second expression represents a group of constraints (in fact, each pair of references contributes one constraint to the system of constraints). In this case, we first try to find a bank decomposition matrix from $\Pi(\delta_j\text{-}\delta_i) = 0$, and then, we determine a suitable q from $\Pi\theta_1 q = 0$. If, however, the said equality does not hold, we first try to find a Λ^{-1} such that the equality $(\theta_i\text{-}\theta_j)\Lambda^{-1} = 0$ will be true. If such a Λ^{-1} can be found, then we try to select a Π such that, the following constraints are satisfied: $\Pi\theta_k q = 0$ and $\Pi(\delta_j\text{-}\delta_i)=0$, where $1 \le k \le v$ and $1 \le i < j \le v$. If, on the other hand, we cannot find a Λ^{-1} that satisfies $(\theta_i\text{-}\theta_j)\Lambda^{-1} = 0$, then it is not meaningful to satisfy $\Pi(\delta_j\text{-}\delta_i) = 0$, and we simply try to select a Π such that only the constraints $\Pi\theta_k q = 0$ are satisfied.

It should be noted that during this solution process we might not be able to satisfy all constraints. That is, we may not be able to select Π and Λ^{-1} such that all our constraints are satisfied (especially when we have too many references to the array). In such cases, to find a solution, we need to drop some constraints from consideration. It should be observed that our constraints come from either a single reference (e.g., $\Pi\theta_1 q = 0$) or from two references (e.g., $\Pi(\delta_j-\delta_i) = 0$). Therefore, we can select the constraint(s) to be dropped by deciding relative importance for array references with respect to each other. For example, a reference that is accessed conditionally (i.e., due to an enclosing if-statement) would be less important than a reference that will definitely be accessed (unconditionally) once the loop is entered. In this case, it makes sense to drop the former reference from the consideration (instead of the latter) if required. More specifically, we can associate weights with references, and in deciding which constraint(s) to drop, we can use these weights. Our current implementation uses the number of compiler-estimated accesses to a reference as its weight. In other words, a more frequently accessed reference will have a larger weight than a less frequently accessed reference. Based on this, the weight of an intra-reference constraint is set to the weight of the reference it contains; and the weight of an inter-reference constraint is set to sum of the weights of the references involved. As an example, suppose that within a nested loop we have three references: U[i][j], U[i-1][j+1], and U[i-2][j+2]. Assume further that the first reference occurs three times, the second one two times, and third one only once. In this case, if the compiler cannot satisfy all the bank locality constraints, it drops those belonging to the reference U[i-2][j+2], and tries to satisfy the remaining ones. If it is still not successful, it drops the ones belonging to U[i-1][j+1], and tries to satisfy the remaining ones and so on. This process continues until the compiler satisfies the remaining constraints, or it cannot satisfy any of them (in which case we do not optimize the array in question).

6. Implementing Array Decompositions

The discussion so far helps us determine suitable loop transformations and bank decompositions for enhancing bank locality. However, it does not tell us anything about how such bank decompositions can actually be implemented. In this section, we demonstrate how data transformations can be used for implementing such decompositions. In particular, we study answers to the questions of the type "what does it mean to distribute a row-major array across memory banks in a column-by-column fashion?"

We recall from data transformation theory [15] that applying a data transformation represented by a linear matrix M to an array transforms a reference $\theta I+\delta$ (to that array) to $M(\theta I+\delta)$. In fact, using such linear transformation matrices has been the most convenient way for implementing memory layout transformations in several high-level languages such as C and FORTRAN [13]. For example, using an inverse identity matrix as M converts the memory layout of a multi-dimensional array from column-major to row-major, or vice versa. We refer the reader to [15] and [13] for a comprehensive discussion of data transformation theory and low-level code generation issues after a data transformation. While the main goal of prior work on data transformations has been enhancing data locality in cache memory based architectures, the main objective in our work is to implement array decompositions across banks of a multi-bank architecture.

In our context, however, we use data transformations for a different purpose. Suppose, for example, that our approach discussed above resulted in a bank decomposition matrix $\Pi = [1\ \ {-1}]^T$. What this means is that two array elements such as d1 and d2 are to be mapped into the same memory bank if and only if

$$\Omega(d_1) = \Omega(d_2) \Rightarrow [1\ \ {-1}]\, d_1 = [1\ \ {-1}]\, d_2.$$

In other words, two array elements such as $[5\ \ 3]^T$ and $[6\ \ 4]^T$ need to be co-located. But, we know that C language stores arrays using a row-major layout; that is, $[5\ \ 3]^T$ and $[6\ \ 4]^T$ are not consecutive in memory (as far as the compiler's view of layout is concerned). In order to reconcile our array decomposition with row-major storage, we need to reshuffle array elements (in memory) such that these two elements are consecutive in memory. This can be done by selecting a suitable data transformation matrix and applying it to the array in question. Our data transformation matrix should be such that after the transformation array elements such as $[5\ \ 3]^T$ and $[6\ \ 4]^T$ should be stored one after another.

In mathematical terms, let us assume that Π is the desired decomposition, and d_1 and d_2 are the array elements that need to be brought together, the selected data transformation matrix M should satisfy the following equality (focusing on a one-dimensional bank layout and a two-dimensional array): $\Pi M = [1\ \ 0]$. This is because, under the mentioned assumptions, $[1\ \ 0]$ represents the distribution of array elements across virtual banks in a row-by-row fashion. By post-multiplying Π by M, we make sure that the elements that are required to be co-located by the decomposition matrix should be treated as if they are consecutive in memory.

7. Global Optimizations

So far, we have discussed how loop transformations can be used for improving memory bank locality, and how data transformations can be used for implementing the desired decomposition of arrays across memory banks. However, our discussion focused on a single nest (which might contain multiple loops) and a single array. In reality, most large-scale embedded applications have multiple nests, each manipulating a subset of the arrays declared in the code. In this section, we extend our approach to multiple arrays and multiple nests.

7.1 Single Nest, Multiple Arrays

Let us first focus on a case where we have a single nest (which may contain multiple loops) that manipulates multiple arrays. For clarity of presentation, we will focus on a special case with two arrays only; however, the results we obtain can easily be generalized to larger number of arrays as well.

Let Π_u and Π_v be the bank decomposition matrices for arrays U and V, respectively. Also, let $\theta_1 I+\delta_1$ and $\theta_2 I+\delta_2$ (resp. $\theta_3 I+\delta_3$ and $\theta_4 I+\delta_4$) be two references to array U (resp. V) in the nest. Then, assuming a loop transformation matrix Λ, for the best bank locality, the following constraints should be satisfied:

- $\Pi_u\theta_1 q = 0$
- $\Pi_u\theta_2 q = 0$
- $\Pi_u(\theta_1-\theta_2)\, \Lambda^{-1}\Gamma' = \Pi_u(\delta_2-\delta_1)$

- $\Pi_v\theta_3 q = 0$
- $\Pi_v\theta_3 q = 0$
- $\Pi_v(\theta_3\text{-}\theta_4)\ \Lambda^{-1}I' = \Pi_u(\delta_4\text{-}\delta_3)$

As before, in these expressions, q denotes the last column of the inverse of loop transformation matrix, and I' is the transformed iteration vector.

Let us first focus on the case where $\theta_1=\theta_2$ and $\theta_3=\theta_4$. In this case, we find a Π_u from $\Pi_u(\delta_2\text{-}\delta_1)=0$, and a Π_v from $\Pi_v(\delta_4\text{-}\delta_3)=0$. Then, using these Π_u and Π_v, we determine a suitable q by simultaneously solving $\Pi_u\theta_1 q = 0$ and $\Pi_v\theta_3 q = 0$. Of course, when such a q is not feasible, we need to drop of one or more equations. The concept of reference weights discussed earlier (in Section 5) can be used for choosing the victim.

If $\theta_1=\theta_2$ and $\theta_3=\theta_4$ do not hold, then, we proceed as follows. First, we check whether it is possible to find a Λ^{-1} such that both $(\theta_1\text{-}\theta_2)\ \Lambda^{-1}=0$ and $(\theta_3\text{-}\theta_4)\ \Lambda^{-1}=0$ will be satisfied. If it is, then, we can solve $\Pi_u\theta_1 q = 0$, $\Pi_u\theta_2 q = 0$, and $\Pi_u(\delta_2\text{-}\delta_1) = 0$ for Π_u; and similarly, we can solve $\Pi_v\theta_3 q = 0$, $\Pi_v\theta_4 q = 0$, and $\Pi_v(\delta_4\text{-}\delta_3) = 0$ for Π_v. As before, a conflict resolution strategy is used when necessary. If, on the other hand, it is not possible to find a Λ^{-1} such that both $(\theta_1\text{-}\theta_2)\ \Lambda^{-1}=0$ and $(\theta_3\text{-}\theta_4)\ \Lambda^{-1}=0$ are satisfied, then the third and sixth constraints given above can be dropped from consideration (i.e., inter-reference locality is sacrificed). Consequently, we first select a q, and then using $\Pi_u\theta_1 q = 0$, $\Pi_u\theta_2 q = 0$, $\Pi_v\theta_3 q = 0$, and $\Pi_v\theta_4 q = 0$, we determine Π_u and Π_v. Note that the case where $\theta_1=\theta_2$ holds but $\theta_3=\theta_4$ does not (or vice versa) can be handled by modifying the strategy explained above slightly. Details are omitted due to lack of space.

7.2 Multiple Nests, Single Array

We now concentrate our attention on another interesting case, where we have multiple nests manipulating one array. Again, for ease of presentation, we focus on a special case where we have only two nests. It should be straightforward to extend these results to multiple nests case. Assuming that Λ_1 and Λ_2 are the loop transformation matrices (to be determined) for the first and second nest, and that q_1 and q_2 are the last columns of Λ_1^{-1} and Λ_2^{-1}, respectively, we need to satisfy the following constraints:

- $\Pi\theta_1 q_1 = 0$
- $\Pi\theta_2 q_1 = 0$
- $\Pi(\theta_1\text{-}\theta_2)\ \Lambda_1^{-1}I' = \Pi(\delta_2\text{-}\delta_1)$
- $\Pi\theta_3 q_2 = 0$
- $\Pi\theta_3 q_2 = 0$
- $\Pi(\theta_3\text{-}\theta_4)\ \Lambda_2^{-1}J' = \Pi(\delta_4\text{-}\delta_3)$

In these expressions, I' (which is $\Lambda_1 I$) and J' (which is $\Lambda_2 J$) are the iteration vectors for the first and second nests, respectively, after the loop transformations. Also, $\theta_1 I+\delta_1$ and $\theta_2 I+\delta_2$ (resp. $\theta_3 J+\delta_3$ and $\theta_4 J+\delta_4$) are the references to the array in the first (resp. second) nest.

As before, let us first consider the case where $\theta_1=\theta_2$ and $\theta_3=\theta_4$. In this case, we can find a Π by solving $\Pi(\delta_2\text{-}\delta_1)=0$ and $\Pi(\delta_4\text{-}\delta_3)=0$ together. Then, using this Π, one can determine a q_1 by solving $\Pi\theta_1 q_1 = 0$, and a q_2 by solving $\Pi\theta_3 q_2 = 0$.

1. *track* ← false
2. Build $\Pi_j\theta_{i,j,k}q_i = 0$
   for $1 \le i \le F$, $1 \le j \le G$, and $1 \le k \le H(i,j)$
3. Build $\Pi_j(\theta_{i,j,k}\text{-}\theta_{i,j,k'})\ \Lambda_i^{-1}I' = \Pi_j(\delta_{i,j,k}\text{-}\delta_{i,j,k'})$
   for $1 \le i \le F$, $1 \le j \le G$, $1 \le k, k' \le H(i,j)$, and $k \ne k'$
4. For each i and j:
   If $(\theta_{i,j,k}=\theta_{i,j,k'})$ for any $k \ne k'$ then
   track ← true
   Solve Π_j from $\Pi_j(\delta_{i,j,k}\text{-}\delta_{i,j,k'})=0$ trying to satisfy as many such k and k' as possible
5. If *track* is true, then for all j such that $\theta_{i,j,k}=\theta_{i,j,k'}$ does not hold for any i, k, k', select arbitrary Π_j
6. If *track* is true, then
   determine q_i from $\Pi_j\theta_{i,j,k}q_i = 0$
7. If *track* is not true, determine Π_j and q_i from $\Pi_j\theta_{i,j,k}q_i = 0$
8. Complete q_i columns to Λ_i^{-1} matrices taking care of data dependences

Figure 2. Compiler algorithm for bank locality

If the above equalities do not hold true, however, we can proceed as follows. First, we check whether it is possible to find a Λ_1^{-1} such that $(\theta_1\text{-}\theta_2)\ \Lambda_1^{-1}=0$ and a Λ_2^{-1} such that $(\theta_3\text{-}\theta_4)\ \Lambda_2^{-1}=0$ will be satisfied. If such Λ_1 and Λ_2 are possible, then, we can solve $\Pi\theta_1 q_1 = 0$, $\Pi\theta_2 q_1 = 0$, $\Pi\theta_3 q_2 = 0$, $\Pi\theta_3 q_2 = 0$, $\Pi(\delta_2\text{-}\delta_1)=0$, and $\Pi(\delta_4\text{-}\delta_3)=0$ for Π. If necessary, a conflict resolution strategy can be used to drop some constraint(s). If, however, it is not possible to find such Λ_1^{-1} and Λ_2^{-1} such that both $(\theta_1\text{-}\theta_2)\ \Lambda_1^{-1}=0$ and $(\theta_3\text{-}\theta_4)\ \Lambda_2^{-1}=0$ are satisfied, then the third and sixth constraints above can be dropped from consideration. In this case, we can select an arbitrary Π, and then using $\Pi\theta_1 q_1 = 0$, $\Pi\theta_2 q_1 = 0$, $\Pi\theta_3 q_2 = 0$, and $\Pi\theta_4 q_2 = 0$, determine q_1 and q_2.

Example*:* As an example, let us assume that a program has two separate nests that access a two-dimensional array U. Assume further that the references in the first nest are U[i][j] and U[i+1][j-1], and those in the second nest are U[j][3i] and U[j+2][3i-2]. First, from $\Pi(\delta_2\text{-}\delta_1)=0$ and $\Pi(\delta_4\text{-}\delta_3)=0$, we need to determine a Π. In this case, we find that $\Pi = [1\ \ 1]$. Then, using this decomposition matrix, we need to determine a q_1 (for the first nest) from $\Pi\theta_1 q_1 = 0$, and a q_2 (for the second nest) from $\Pi\theta_3 q_2 = 0$. In this example, we find that $q_1 = [1\ \ -1]^T$ and $q_2 = [1\ \ -3]^T$. After that, using Li's method [14], these vectors (last columns) can be completed to Λ_1^{-1} and Λ_2^{-1}.

7.3 Multiple Nests, Multiple Arrays

This is the most general case where multiple nests operate on multiple arrays. A solution to this case can be developed by combining the solutions presented in the last two sections ([single nest, multiple arrays] and [multiple nests, single array]). Therefore, instead of giving the details (of the solution procedure) here, we just present the overall solution in an algorithmic form in Figure 2.

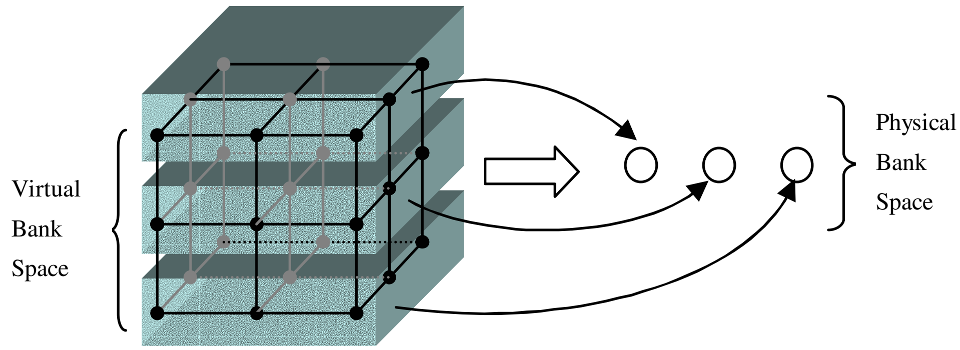

Figure 3. An example folding function

In this figure, $\theta_{i,j,k}$ represents the access matrix to the k^{th} reference to array j in nest i. $\delta_{i,j,k}$ is the corresponding offset vector. Π_j, Λ_i, and q_i denote, respectively, the bank decomposition matrix for array j, the loop transformation matrix for nest i, and the last column of Λ_i^{-1}. Also, F, G, and H(i,j) denote, respectively, the number of nests, the number of arrays, and the number of references to array j in nest i. In lines 2 and 3, we build intra-reference and inter-reference constraints (for all arrays, references, and nests in the code). The variable *trace* indicates whether inter-reference constraints should be taken into account. If the loop in line 4 determines that for some arrays, inter-reference constraints can be solved, it determines the bank decomposition matrices taking into account those constraints as well (in addition to intra-reference constraints). For the arrays that it is not possible, only intra-reference constraints are accounted for. In lines 4, 6, and 7, we use the weight concept defined earlier for resolving conflicts (i.e., for dropping constraints). Finally, some remarks on line 7 follow. In order to solve such a system, we use an incremental approach. For example, we can first fix a Π_j and determine (using this Π_j) a q_i. This q_i in turn can be used to find a $\Pi_{j'}$ (where j' ≠ j), which can be used, in turn, for determining a $q_{i'}$ (where i' ≠ i), and so on.

7.4 Discussion

It should be stressed that so far our presentation has centered on intra-reference and inter-reference constraints. Both of these constraints are in fact what we can call *intra-array constraints.* That is, they are defined with respect to a given array. One can also define *inter-array constraints*. The idea would be clustering array elements (that belong to different arrays) that are accessed by the same loop iteration ([3] tries to achieve this by employing clustering heuristics). The displacement vectors (that we have not exploit so far) in bank decomposition function can be used for this purpose (along with bank decomposition matrices). Handling inter-array constraints can enable aggressive bank locality optimizations such as array interleaving and might result in a very complex set of equations (to be satisfied). Due to lack of space, we postpone the detailed discussion of inter-array constraints to a future study. However, it should be mentioned that intra-array constraints and inter-array constraints are complementary; that is, an optimizing compiler should try to satisfy both of them to the extent possible.

8. Folding Functions

A folding function specifies a mapping of virtual banks to physical banks (i.e., the banks that actually exist in the system). In doing so, it can reduce the dimensionality and/or extents (dimension sizes) of the virtual bank space. A folding function Δ is specified by giving a decomposition style for each dimension of the virtual bank space along with the physical bank size in each dimension (of the physical bank space). For example, a folding function such as

$$\Delta:\ (b_1,b_2,b_3) \rightarrow (b_1/N,*,*)$$

indicates that a three-dimensional virtual bank space is mapped to a one-dimensional physical bank space. The * notation here indicates that the corresponding virtual bank dimension is not distributed across physical banks. This indicates, for the example above, that the second and the third dimensions are not distributed; instead, they are folded. The notation b_1/N (where / denotes integer division) in the first dimension, on the other hand, reveals that this dimension (of the virtual bank space) is distributed across N physical banks. So, assuming N is 8, under such a folding function, the virtual bank (16,i,j) is mapped to physical bank 2 for all values i and j can take.

This folding function definition is quite general and encompasses very different types of virtual-to-physical bank mappings. For example, it can accommodate functions such as

$$\Delta:\ (b_1,b_2,b_3) \rightarrow (b_1/N,b_2/M,*),$$

which indicates that a three-dimensional virtual bank space is mapped to a two-dimensional physical bank space that contains NxM banks (this might be useful, for example, for some SDRAMs, where memory banks actually form a two-dimensional grid). It should also be noted that when multiple virtual banks are mapped to the same physical bank, the loop iterations that access those virtual banks are localized (that is, they exhibit bank locality as they now –after folding- access the same physical bank). Therefore, selection of a suitable folding function can be very important. In this paper, we set the dimensionality of the virtual bank space (for a given application) to the number of dimensions of the array (in that application) with the largest number of dimensions. Also, we restrict ourselves to one-dimensional physical bank spaces. Consequently, we use folding functions of the following type:

$$\Delta: (b_1,b_2,\ldots,b_r) \rightarrow (b_1/N,*,\ldots,*).$$

Such a folding function tries to take advantage of spatial locality between neighboring banks. In the future, we will study the impact of other types of folding functions on memory energy behavior. Figure 3 shows an example folding, where a three-dimensional virtual bank space is mapped to a one-dimensional physical bank space. We see that each two-dimensional slice of the virtual bank space is mapped to a single node of the physical bank space.

9. Experiments

Our objective in this section is to demonstrate how successful our approach is in practice. In Section 9.1, we introduce our benchmarks and experimental setup. In Section 9.2, we give detailed experimental results.

9.1 Benchmark Codes and Experimental Framework

Benchmark Name	Number of Lines	Input Size	Base Energy	Base Time
Bss	486	44MB	274.3mJ	8.11nsec
Flt	324	62MB	188.1mJ	5.20nsec
Spe	156	61MB	108.1mJ	2.14nsec
Seg	144	58MB	106.4mJ	2.08nsec
Vid	1,618	62MB	342.7mJ	9.65nsec

Figure 4. Benchmark codes used in our experiments and baseline values

Figure 4 lists the benchmark codes used in this study and their important characteristics. Our applications are array-intensive codes that are typical in many embedded environments that target image and video processing. Bss is a blind signal separation and deconvolution application. Flt is a particle filter algorithm for wireless communication. Spe is combined spectral envelope normalization and subtraction algorithm. Seg is an image segregation code based on pitch tracking and amplitude modulation. Finally, Vid is a video-enhancing algorithm that modifies transient locations. The reason that we selected these applications for our study is that they represent real-life embedded codes and were available to us. The third column in Figure 4 gives the total size of the data manipulated by each application. The last two columns, on the other hand, give the memory energy consumptions and execution times of our applications when an 8x8MB (total 64MB without any data cache) memory configuration is employed and when our low-power operating modes are used (whenever the banks are idle). In other words, even our base configuration takes advantage of low-power operating modes to the extent possible, and the benefits reported in Section 9.2 come solely from our optimization. The energy numbers reported in the next subsection are the values *normalized* with respect to the values given in the fourth column of Figure 4. Unless stated otherwise, 8x8MB (that is, 8 memory banks, each has a capacity of 8MB) is our default bank configuration. Note that if a bank is not accessed during the entire execution of an application, the said bank is never activated for both the original and the optimized code versions.

Our loop and data transformations are implemented using SUIF from Stanford University [1]. SUIF has independently developed compilation passes that work together by using a common intermediate format (IF) to represent programs. A typical compilation framework based on SUIF includes the following components: front end, data dependence analysis, and several optimization modules. Our framework is implemented as a separate optimization module within SUIF. We also use a powerful back-end compiler (when converting the C code to the executable) that performs instruction scheduling and graph coloring based global register allocation.

All energy numbers presented in this paper have been obtained using a custom memory energy simulator. This simulator takes as input a C program and a banked memory description (i.e., the number and sizes of memory banks as well as available low-power operating modes with their energy saving factors and re-synchronization penalties). As output, it gives the energy consumption in memory banks along with a detailed bank inter-access time profiles. By giving the original and the optimized programs to this simulator as input, one can measure the impact of loop and data optimizations on memory system energy of a banked architecture.

9.2 Results

We start by presenting the percentage energy improvements when our optimization strategy is applied to the codes in our experimental suite. The results illustrated in Figure 5 (the first bar for each application) indicate that our strategy improves the energy consumption of all benchmark codes in our experimental suite. The average percentage memory energy improvement (across all benchmarks) is 19.1%. Note that our approach (in addition to determining a loop transformation for each nest in the code) also determines a suitable data transformation for each array (to implement array decompositions as explained in Section 6). These data transformations are dependent on the initial layout of data (row-major in our case); however, the resulting memory layouts are independent of the initial (default) memory layout. That is, independent of the initial layout of the arrays, our strategy determines the most suitable memory layouts for them (depending on the loop transformations selected for the nests) from the viewpoint of bank locality. However, the base energy numbers given in Figure 4 are dependent on the initial layout. Therefore, it is important to see whether our optimizations are still effective (in improving memory energy behavior) when the original data layout is highly optimized. To analyze this issue, we also measured the percentage energy savings over an energy optimization strategy that does not apply any loop transformations but selects the most suitable bank layouts for each array in the code being optimized (this version is from [3]). The results given in Figure 5 (the second bar for each application) clearly show that our approach is still able to improve memory energy consumption for all benchmarks in our suite, achieving an average memory energy improvement of 13.8%. It should also be observed that the energy benefits due to our optimizations vary depending on the application under consideration. Specifically, we obtain the best results with Flt. This is due to two main reasons. First, there are not many data dependences in this application; consequently, the compiler finds a legal (semantic-preserving) loop transformation most of the time. Second, there are not many array references per loop, and so, the compiler does not need to drop some (bank locality) constraints from consideration frequently. In fact, in compiling this application, the compiler dropped only five constraints (the average value across all benchmarks was thirteen). In comparison, our savings with Spe are relatively low due to large number of data dependences in the code (that is, although most of the time the compiler was able to find a loop transformation matrix Λ, the resulting transformation was found to be illegal – not semantic-preserving).

In order to evaluate the impact of our approach with different bank configurations, we performed another set of experiments. More specifically, keeping the total memory size fixed at 64MB, we conducted experiments with 2, 4, 8, 16, 32, and 64 banks. The energy results are given in Figure 6 (again as percentage improvements). One can observe from these results that working with larger number of banks (i.e., with smaller bank sizes) in general increases the benefits due to our optimization strategy. This is because smaller bank sizes give our strategy more opportunities for energy-managing even smaller portions of main memory. Such a finer-grain management, in turn, increases the energy benefits. However, one can also see from the same graph that, in two applications (Bss and Flt), increasing the number of memory banks beyond 16 does not bring further energy benefits. This is because in these two applications increasing the number of banks beyond 16, say to 32, increases the average number of banks accessed per loop iteration. This in turn offsets the potential benefits of working with a larger number of banks. We faced the same problem with the remaining three applications only when we went beyond 64 banks (the results of which are not shown here).

So far, our experiments involved no cache memory. Including a data cache in the architecture can impact memory system energy consumption in at least two ways. First, the cache filters out some memory references that would otherwise go to off-chip memory. Consequently, we can expect an increase in bank inter-access times. Second, the cache reduces the overall energy consumption due to memory system as typically cache accesses consume much less energy than off-chip memory accesses. To calculate the data cache energy consumption, our simulator employs the cache energy model proposed in [9]. Figure 7 gives the percentage memory system energy improvements with different on-chip data cache configurations over both original layouts and optimized layouts. Note that memory system energy with the cache versions includes the energy consumption in the cache as well as that in the off-chip memory. All cache configurations used are two-way set-associative with a block size

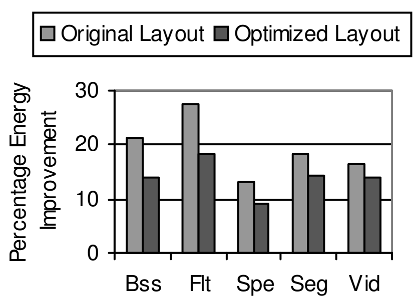

Figure 5. Percentage energy savings over original (unoptimized) and optimized layouts

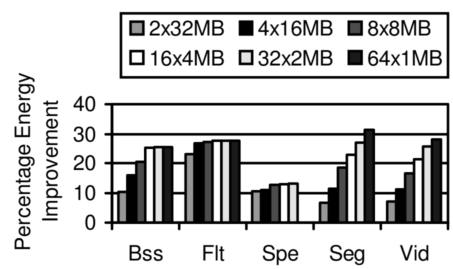

Figure 6. Percentage energy savings with different bank configurations (over original layouts)

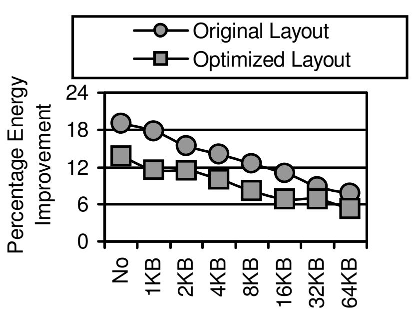

Figure 7. Percentage energy savings with different cache configurations (averaged over all benchmarks)

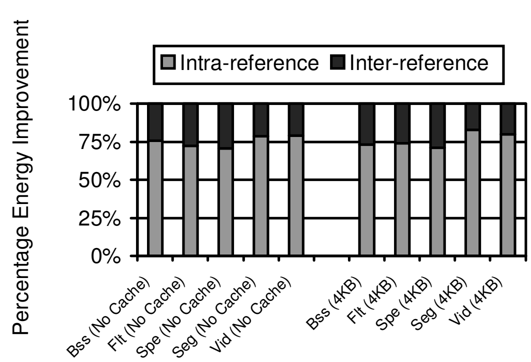

Figure 8. Contributions of intra-reference and inter-reference constraints to memory energy savings (averaged over all benchmarks)

of 32 bytes. One can clearly see from Figure 7 that, while increasing the cache size reduces the benefits of our optimization strategy, we still achieve reasonable energy savings. Therefore, even in a cache-based environment, applying bank-aware loop and data transformations can be useful. One should observe, however, that if the compiler can have an accurate hit/miss prediction strategy, it could potentially do a much better job in managing memory banks. Implementing such a strategy is in our future agenda.

The energy benefits shown in this section have been obtained by trying to satisfy both intra-reference and inter-reference constraints. We also wanted to see how these constraints contribute to these energy savings. In Figure 8, we illustrate how energy benefits are broken down between those coming from satisfying inter-reference constraints and those coming from satisfying intra-reference constraints (for both a cacheless memory system and for a memory system with 4KB data cache – both with our default bank configuration). These results reveal that most of the energy savings are coming from satisfying intra-reference constraints. There are two main reasons for this. First, the algorithm given Figure 2 does not try to satisfy inter-reference constraints if $\theta_{i,j,k}=\theta_{i,j,k'}$ checks fail. Second, in many cases, satisfying intra-reference constraints brings more benefits as such constraints capture access behavior across loop iterations. This is in contrast to inter-reference constraints whose existence is limited by the number of references to the same array in the loop body. Nevertheless, we still observe that the contribution of satisfying inter-reference constraints to overall memory energy savings is around 25% on the average, which indicates that it is important to take care of them as well.

Finally, we studied the performance impact of power-mode control with/without our loop and data transformations on performance. Since in this work we focus on array-intensive embedded applications (with compile-time analyzable data access patterns) and adopted a compiler-based operating mode control strategy, the compiler can also predict future bank re-activations and perform ahead-of-time bank re-activation (called pre-activation). We refer the reader to [3] for a discussion of memory bank pre-activation and its implementation details. Consequently, the performance penalty of power mode control itself was negligible (less than 2% on the average). When our optimizations are applied, we did not observe any significant performance variation in the cacheless memory architecture. In a cache-based memory architecture, on the other hand, we observed nearly 11% performance improvement on the average (mainly coming from the improved cache access patterns).

10. Concluding Remarks and Future Work

Energy consumption is becoming increasingly important for both embedded and high-performance computing systems. The dynamic energy spent in off-chip DRAMs can be a significant portion of the overall energy budget. In this paper, we have presented a compiler-based optimization framework for array-intensive embedded applications to increase energy savings that can be obtained from low-power operating modes. Our experiments with a prototype implementation and five array-intensive embedded applications show significant savings in memory system energy.

Our future work consists of two different directions:

- First, we would like to study the impact of data optimizations on the effectiveness of our optimization strategy. While in this work we employed data transformations for implementing array decompositions, they can also be used for improving bank locality. This can be particularly useful in applications such as Vid, where we have lots of data dependences preventing loop transformations from modifying the access pattern to enhance bank locality. It is also possible that combining loop and data transformations under an integrated framework can further increase energy savings.
- Second, we would like to apply our optimization to an on-chip multiprocessor architecture. In particular, we believe that studying how bank localities of individual processors interact with each other can reveal important data, which can be used to reduce memory energy consumption in such architectures.

11. Acknowledgements

This work was supported in part by NSF CAREER Award #0093082, and NSF Award #0103583.

12. References

[1] S. P. Amarasinghe, J. M. Anderson, C. S. Wilson, S.-W. Liao, B. R. Murphy, R. S. French, M. S. Lam, and M. W. Hall. Multiprocessors from a Software Perspective. IEEE Micro, June 1996, pages 52-61.

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

[3] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM Energy Management Using Software and Hardware Directed Power Mode Control. In Proc. the 7th Int'l Symposium on High Performance Computer Architecture, 2001.

[4] V. Delaluz and M. Kandemir. Array Regrouping and Its Use in Compiling Data-Intensive Embedded Applications. To appear in IEEE Transactions on Computers.

[5] X. Fan, C. S. Ellis, and A. R. Lebeck. Modeling of DRAM Power Control Policies Using Deterministic and Stochastic Petri Nets. In Proc. Workshop on Power-Aware Computer Systems, Springer-Verlag, February 2002.

[6] 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, pp. 252-263, 2000.

[7] A. Farrahi, G. Tellez, and M. Sarrafzadeh. Exploiting Sleep Mode for Memory Partitions and Other Applications. VLSI Design, Vol. 7, No. 3, pp. 271-287.

[8] W-M. W. Hwu. Embedded Microprocessor Comparison. http://www.crhc.uiuc.edu/IMPACT/ece412/public_html/Notes/412_lec1/ppframe.htm.

[9] M. B. Kamble and K. Ghose. Analytical Energy Dissipation Models for Low Power Caches. In Proc. International Symposium on Low-Power Electronics and Design, 1997.

[10] M. Kandemir, U. Sezer, and V. Delaluz. Improving Memory Energy Using Access Pattern Classification. In Proc. the International Conference on Computer Aided Design, San Jose, CA, November 4-8, 2001.

[11] M. Kandemir, I. Kolcu, and I. Kadayif. Influence of Loop Optimizations on Energy Consumption of Multi-Bank Memory Systems. In Proc. International Conference on Compiler Construction, Grenoble, France, April 6-14, 2002.

[12] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power-Aware Page Allocation. In Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[13] S. -T. Leung and J. Zahorjan. Optimizing Data Locality by Array Restructuring. Technical Report TR 95-09-01, Dept. of Computer Science and Engineering, University of Washington, September 1995.

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

[15] M. O'Boyle and P. Knijnenburg. Integrating Loop and Data Transformations for Global Optimization. In Proc. International Conference on Parallel Architectures and Compilation Techniques, October 1998, Paris, France.

[16] P. R. Panda. Memory Bank Customization and Assignment in Behavioral Synthesis. In Proc. ICCAD, 1999.

[17] 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., May 1999.

[18] 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, Cambridge, MA, 1996, pp. 234–243.

[19] A. Sudarsanam, S. Malik. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs." ACM Transactions on Design Automation of Electronic Systems 5, 2000, pp. 242–264.

[20] W. Wolf. Modern VLSI Design: System-on-Chip Design, Prentice Hall PTR, 2002.

[21] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In Proc. ACM Conference on Programming Language Design and Implementation, pp. 30-44, June 1991.

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