Improving the Energy Behavior of Block Buffering Using Compiler Optimizations

M. KANDEMIR
Pennsylvania State University
J. RAMANUJAM
Louisiana State University
and
U. SEZER
University of Wisconsin

On-chip caches consume a significant fraction of the energy in current microprocessors. As a result, architectural/circuit-level techniques such as block buffering and sub-banking have been proposed and shown to be very effective in reducing the energy consumption of on-chip caches. While there has been some work on evaluating the energy and performance impact of different block buffering schemes, we are not aware of software solutions to take advantage of on-chip cache block buffers.

This article presents a compiler-based approach that modifies code and variable layout to take better advantage of block buffering. The proposed technique is aimed at a class of embedded codes that make heavy use of scalar variables. Unlike previous work that uses only storage pattern optimization or only access pattern optimization, we propose an integrated approach that uses both code restructuring (which affects the access sequence) and storage pattern optimization (which determines the storage layout of variables). We use a graph-based formulation of the problem and present a solution for determining suitable variable placements and accompanying access pattern transformations. The proposed technique has been implemented using an experimental compiler and evaluated using a set of complete programs. The experimental results demonstrate that our

A preliminary version of this article appeared in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, ACM, New York, 2001. This article enhances the preliminary version by explaining how the proposed technique interacts with register allocation and by presenting more experimental data.
M. Kandemir is funded in part by National Science Foundation (NSF) grant CCR-0093082 and by the Pittsburgh Digital Greenhouse through a grant from the Commonwealth of Pennsylvania, Department of Community and Economic Development.
J. Ramanujam is supported in part by NSF grant CCR-007380 and NSF Young Investigator Award CCR-9457768.
Authors' addresses: M. Kandemir, Pennsylvania State University, Department of Computer Science and Engineering, University Park, PA 16802; email: kandemir@cse.psu.edu; J. Ramanujam, Louisiana State University, Department of Electrical and Computer Engineering Baton Rouge, LA 70803; email: jxr@ee.lsu.edu; U. Sezer, University of Wisconsin, Department of Electrical and Computer Engineering Madison, WI 53706; email: sezer@ece.wisc.edu.

approach leads to significant energy savings. Based on these results, we conclude that compiler support is complementary to architecture and circuit-based techniques to extract the best energy behavior from a cache subsystem that employs block buffering.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Caches memories*; D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*

General Terms: Design, Performance

Additional Key Words and Phrases: Energy optimizations, compiler transformations, block buffering, embedded systems, data cache

1. INTRODUCTION

On-chip caches are a major source of energy consumption in current microprocessors. For example, Edmondson et al. [1995] report that the on-chip cache in DEC Alpha 21264 consumes approximately 25% of the on-chip energy. Circuit and architectural techniques that implement alternative cache organizations (e.g., sub-banking, bitline segmentation, and block buffering [Kamble and Ghose 1997]) have been shown to be very effective in reducing the energy consumption of on-chip caches. A *block buffer* is a small line buffer inserted between the processor and the first-level on-chip cache to hold the most recently accessed cache line (block). If the cache line that resides in the block buffer is accessed again, this request can be satisfied from the block buffer, and the main on-chip cache (both data and tag arrays) can be disabled during this access to save energy. While previous research [Su and Despain 1995; Ghose and Kamble 1999; Kin et al. 1997; Esakkimuthu et al. 2000] investigated different block buffering schemes and evaluated their impact on energy and performance, to the best of our knowledge, no previous study considered compiler support for block buffering.

In this article, we present a compiler-based approach that modifies code and variable (data) layout to take better advantage of block buffering. The application domain that this technique targets includes a class of embedded codes that make heavy use of scalar variables. That is in contrast to a vast amount of locality-oriented work done for array-dominated applications in the optimizing compiler area (see, for example, Wolfe's book [Wolfe 1996] and the references therein). While previous work such as that of Panda et al. [1997] and Panda [1998] addresses optimizations for improving cache locality of scalar-dominated codes, their approach is limited to variable placement (called *storage pattern* or *storage sequence optimization* in this article). In comparison, the approach proposed in this paper employs both storage pattern and access pattern optimizations. Specifically, this article makes the following contributions:

—It presents an access pattern (access sequence) optimization technique to maximize the benefits of block buffering for data accesses in scalar-dominated embedded codes;

—It presents a unified (integrated) optimization strategy that employs both access pattern and storage pattern (variable placement) optimizations for multi-basic block codes; and

— It quantifies the benefits from the proposed techniques over a straightforward block buffering scheme that does not make use of any compiler support using four complete programs.

Our experimental results indicate that the proposed techniques improve data cache energy consumption by 45.8% on average, and by as much as 49.7% in some cases when no register allocation is performed. In the existence of an aggressive register allocation, the percentage energy benefits range from 17.0% to 43.9% for data cache, and from 6.8% to 18.8% for the entire data memory system. Based on these results, we believe that compiler support is complementary to architecture and circuit-based techniques to extract the best energy behavior from the cache subsystem equipped with a block buffer.

The remainder of this article is organized as follows. Section 2 gives background material on block buffering and compiler optimizations. Section 3 describes the problem and Section 4 formulates it on a graph-based structure. Section 5 gives a solution strategy for a single basic block (a straight line of code without branching/conditionals) case, and Section 6 presents a solution for multiple basic block case (i.e., whole procedure). Section 7 discusses extensions to our basic scheme and Section 8 presents experimental data showing the effectiveness of the proposed technique. Section 9 concludes with a summary and an outline of the planned future work.

2. BACKGROUND

2.1 Block Buffering

Block buffering is an extension of conventional cache architectures that uses small (one-block wide) line buffers. The key idea is to keep the most recently accessed cache line (block) in the block buffer so that the following request can access the data from the block buffer if it targets the same cache line (this clearly depends on the block-level data locality exhibited by the application). As noted by Ghose and Kamble [1999], this not only saves accesses to data arrays of the on-chip cache but, at the same time, also saves the access to the tag arrays. Su and Despain [1995] propose a single block buffer structure; Ghose and Kamble [1999] extend this structure to multiple buffers (for set-associative caches), and report as much as 75% savings in power dissipation as compared to a conventional cache architecture when block buffering is combined with other energy-saving hardware optimizations. Bellas et al. [1998] propose the use of a cache that sits between the CPU and the instruction cache to improve performance and energy consumption.

A simplified block buffering scheme is depicted in Figure 1. This is similar to the architecture proposed by Ghose and Kamble [1999]. An access to the block buffer-augmented cache is performed in two cycles but at the rate of one access per cycle using a two-phase clock. In the first cycle, the last set number is compared to the corresponding field of the address issued by CPU to determine if the current access is to the same set as the previous one. If it is, the tag and data array sensing are disabled. Otherwise, the selected set is latched in and the set number for the current access is moved into the latch that holds the

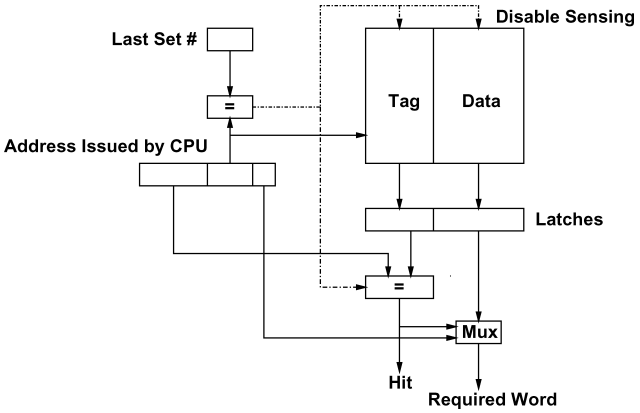

Fig. 1. Operation of block buffering.

set number for the last access. In the second cycle, normal tag comparison is done and if it succeeds, word multiplexing is performed as in the case of normal caches.

2.2 Compiler Optimizations for Data Locality

A vast majority of research in compiler optimizations has targeted array-dominated codes using source-level transformations to make better use of caches. Among the techniques investigated are loop restructurings (e.g., loop permutations and tiling [Wolfe 1996]), data transformations (e.g., dimension reindexings [Kandemir and Ramanujam 2000], integrated data and loop transformations [Kandemir et al. 1998, 1999; Kulkarni et al. 2000, 2001], dynamic memory layout modifications [Mellor-Crummey et al. 1999]), and locality enhancing transformations for pointer-intensive codes [Chilimbi et al. 2000]. An important characteristic of the techniques in the first two groups mentioned is that they focus on multilevel nested loops and exploit the regularity due to affine expressions in array subscript functions. Consequently, it is possible to model the problem using linear algebraic representations [Wolfe 1996] and/or polyhedral tools [Kelly et al. 1995], and solve it using linear transformations of iteration and/or data spaces [Catthoor et al. 1998]. The codes that are targeted by the approach presented in this paper, however, are quite different from those as they involve sequences of scalar assignments and conditional flow of control. Thus, they do not lend themselves to polyhedral formulations; instead, a graphical representation might be preferable.

A unique aspect of our approach is that the unified strategy that we discuss can optimize an entire procedure that might consist of multiple basic blocks. It operates on a control flow graph (CFG) representation of the procedure, and propagates information (obtained through compiler analysis as explained later) between basic blocks to obtain a globally (procedure-wide) acceptable solution. Our experimental results show that such a global approach is superior to the techniques that optimize each basic block in isolation.

3. PROBLEM DESCRIPTION

In this section, we describe the problem addressed in this article. Until Section 7, we assume that no register allocation is performed and all variables accessed from memory (through cache). We also assume the the architecture or the back-end compiler does not modify the order of load/store operations. In Section 7, we discuss in detail how our approach interacts with register allocation.

The success of a block buffering scheme is strongly dependent on the variable access pattern. For example, consider the following code fragment that consists of two assignment statements:

```
a = c + a + b
b = c + b + a
```

Assuming that the addition operation (+) is left-associative, the variable access pattern (access sequence) imposed by these two statements is `c,a,b,a, c,b,a,b`. If we have a single block buffer of one scalar element (word) wide, such an access pattern will not take advantage of block buffering as no variable in the sequence is accessed twice in a row. Note, however, that using commutativity and associativity properties of the addition operation, this program fragment can be rewritten as:

```
a = c + b + a
b = a + c + b
```

The new access sequence is `c,b,a,a,a,c,b,b`. While this transformed code fragment is semantically equivalent to the original one, it exploits the mentioned block buffer much better as it has a total of three repetitions: two for variable `a` (in subsequence `a,a,a`) and one for variable `b` (in subsequence `b,b`). Therefore, we can expect three block buffer hits (i.e., data accesses satisfied from the block buffer) from this new sequence. Now, consider the following code fragment:

```
a = 1
c = c - 1
d = a + b
```

The original access sequence is `a,c,c,a,b,d` and an optimization scheme that is limited to intrastatement transformations (e.g., those exploit commutativity and associativity) cannot generate a better sequence. However, if we apply an interstatement transformation that interchanges the order of the second and the third assignments, we obtain the following fragment:

```
a = 1
d = a + b
c = c - 1
```

This new fragment leads to an access sequence of `a,a,b,d,c,c` which is better than the original one as the second access to variable `a` in this sequence results in a buffer hit. These two examples show that access sequence optimizations can generate improvements (over original codes); that is, they increase the number of hits in the block buffer.

Let us now assume that our block buffer can hold k (> 1) variables (i.e., a line size of k elements). In this case, given a code fragment, we can increase the block buffer hit rate (which is defined as the ratio between the number of block buffer hits and the total number of data accesses) by using both access sequence and storage sequence (variable layout) optimizations. Consider the following code fragment, assuming that $k = 2$ and that the storage order of the variables is `a,b,c,d`, the first variable being at the head of a cache line, that is, aligned to the cache line boundary.

```
a = b + d
c = c + d + a
d = b + a
d = a + c + d
```

The original access sequence is `b, d, a, c, d, a, c, b, a, d, a, c, d, d`. For the given storage sequence `a, b, c, d`, the block buffer hit rate is 4/14; the hits are due to the subsequences `c, d, b, a, c, d` and `d, d`, which map onto the same cache line, and therefore, the block buffer.

Now, we will illustrate the effects of (i) optimizing only the storage sequence; (ii) optimizing only the access pattern (e.g., changing the access sequence by exploiting properties of the computation such as operator commutativity and associativity); and (iii) combining (integrating) access and storage sequence optimizations.

Optimizing only the Storage Sequence. Without making any changes to the given access sequence, if we change the storage sequence to `a, d, b, c` (derived using the approach due to Panda et al. [1997]), we get the best (among those that change only the storage sequence) block buffer hit rate of 6/14.

Optimizing only the Access Sequence. Given the original storage sequence of `a, b, c, d`, consider the following (equivalent) transformed code:

```
a = d + b
c = a + c + d
d = b + a
d = a + c + d
```

The right-hand side expressions in the first two statements have been reordered to improve the block buffer hit rate. The transformed (changed) access sequence is `d, b, a, a, c, d, c, b, a, d, a, c, d, d`. The best hit rate possible for this new access sequence is 7/14, which for example, can be realized using the storage sequence `a, b, c, d`.

Optimizing both the Access Sequence and Storage Sequence. Starting from the original code, consider the use of the storage sequence `a, d, b, c` together with the following (equivalent) transformed code:

```
a = b + d
c = a + d + c
d = b + a
d = a + c + d
```

The transformed (changed) access sequence is `b, d, a, a, d, c, c, b, a, d, a, c, d, d`. With the storage sequence `a, d, b, c`, we get a hit rate of 8/14. Note that, this hit rate is higher than the best achievable for the given code segment using either only storage sequence or only access sequence optimizations. Thus, in general, an integrated framework that uses both access and storage sequence optimizations can be very useful in practice.

We can conclude from these examples that in order to increase the block buffer hit rate, we need to maximize the number of consecutive accesses to a given line. This can be done by modifying the order of variable access (access sequence optimization), by modifying the storage order of variables in memory (storage pattern optimization), or by a combination of these (unified optimization). Note that, when k is one, access sequence optimization is the only option.

4. REPRESENTATION

It should be emphasized that the access pattern optimization problem is important for two major reasons. First, it is the only available optimization strategy when the cache line is single element wide or when storage (variable layout) optimizations are not applicable. Second, the access pattern optimization strategy can be used as a part of a global optimization framework that uses both storage pattern and access pattern transformations. Section 6 discusses such an optimization framework that can handle an entire procedure.

We represent the problem of access pattern optimization using a graph-based structure and solve it using a longest path algorithm. We define two data elements as *neighbors* if they map on the same cache line and the distance between them (in memory) is less than k (the cache line size) elements. Note that the neighboring elements are brought into cache (when they are accessed) and hence into the block buffer at the same time. We also define *virtual lines* in memory, each of which holding a group of neighboring elements.

Given a basic block (i.e., a block of sequential assignment statements without a branch except maybe at the end of the sequence [Aho et al. 1986]), we use a *layout transition graph* (LTG) to show the connections between elements that are mapped on the same cache line. Specifically, a layout transition graph of a basic block is a directed graph $LTG(V, E)$ where each node (vertex) $v_i \in V$ represents the occurrence of a variable in the basic block, and a bi-directional edge $e = (v_i, v_j) \in E$ from a node v_i to a node v_j indicates that the variables represented by v_i and v_j are neighbors (i.e., they are in the same virtual line). An LTG also contains an edge between v_i to v_j if these two nodes represent the occurrences of the same variable.

For ease of exposition, we divide a given LTG into *layers,* each of which corresponding to an assignment statement in the basic block. If the basic block contains K statements, each variable v_i in the jth statement from top (denoted s_j where $1 \leq j \leq K$) is assumed to belong to the variable set of s_j; we express this concept using the notation $v_i \in s_j$. When there is no confusion, we will abuse the notation s_j to denote both the statement and its variable set.

A given variable set s_i can also be divided into two logical subsets: one that contains the variable on the left-hand side (LHS), and one that contains

the variables on the right-hand side (RHS). For a variable set s_i, the first set is denoted by s_{iL} and the second set is denoted by s_{iR}. As will be discussed later in this section, the edges of the LTG can be used to traverse the nodes in the graph, which, in turn, corresponds to intrastatement and interstatement transformations.

We define a *traversal* of (the variables in) a given LTG as a set of paths that collectively visit each and every node only once without mixing accesses to the variables from different statements. While a given LTG shows the storage connections (relations) between the variables in the basic block, it does not dictate any traversal. Note that a traversal of the LTG corresponds to an ordered sequence of variable accesses (i.e., access pattern). Consequently, different traversals correspond to different access patterns. It is well known from data dependence theory [Wolfe 1996] that there are some traversals (of variables during execution) that are not valid (i.e., semantically correct). To eliminate some of the invalid traversals from the LTG, we constrain it by eliminating any edge $e = (v_i, v_j) \in E$ if going from v_i to v_j during the program execution (i.e., touching the variables represented by v_i and v_j immediately one after another) does not preserve the original semantics of the code. This pruned LTG is called *constrained* LTG (CLTG) in this article and is the main data structure which the optimizations we employ operate on.

To illustrate how a LTG and a CLTG are constructed, let us consider the following code fragment (basic block):

```
a = i  + b  + 3j
e = g  + h  + k
b = 2d + 4a - 5
k = l  + f
```

Let us assume that the variables are stored in memory in the order of `a,b,c,d,e,f,g, h,i,j,k,l` and that $k = 4$. Consequently, we have three virtual lines: {`a,b,c,d`}, {`e,f,g,h`}, and {`i,j,k,l`} (We assume perfect alignment). Figure 2(i) shows the four layers corresponding to four statements in this fragment. Each layer is delimited using dashed lines and corresponds to an individual statement. For example, labeling the first statement as s_1, we have $s_{1L} = \{\texttt{a}\}$ and $s_{1R} = \{\texttt{i,b,j}\}$. Given the storage sequence (virtual line mapping) above, Figure 2(v) shows the LTG for this code fragment. Note that there is a bidirectional edge between two nodes whenever the corresponding variables are neighbors (i.e., they reside in the same virtual line). Figures 2(ii), (iii), and (iv), on the other hand, show the contributions (that can be called *sub-LTGs*) coming from the three virtual lines (group of neighbors) mentioned above. It should be noted that a different alignment in memory (virtual line mapping) would generate a totally different LTG.

We see that the LTG shown in Figure 2(v) is very dense. However, assuming that the statements in the fragment will not be broken into sub-statements and that the variable accesses from different statements are not mixed, a simple analysis of the code fragment reveals that many of the edges in this LTG cannot be traversed by a legal access pattern. For example, there is no way that the edge from `i` to `k` be taken by any given schedule (access pattern) as a LHS variable needs to be touched between these two variables. Data dependence constraints

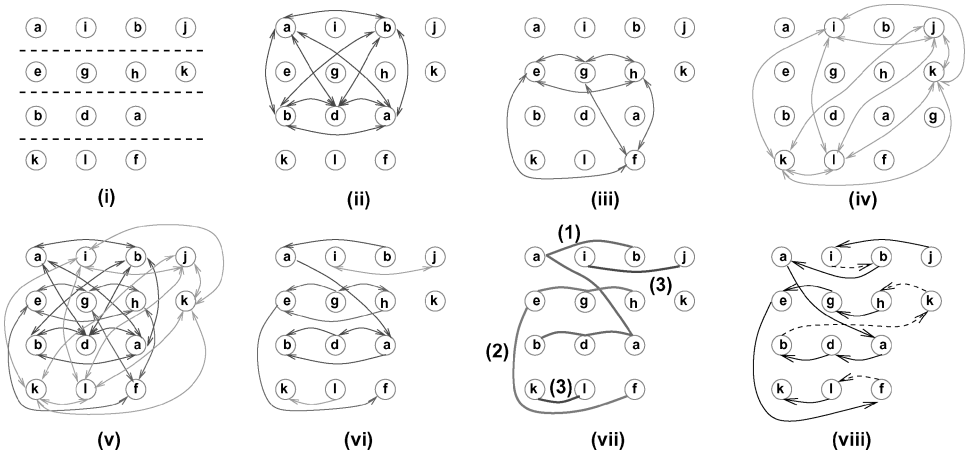

Fig. 2. (i) Four layers (corresponding to four statements) for a basic block. (ii–iv) sub-LTGs induced by different virtual lines. (v) the overall LTG. (vi) the corresponding CLTG. (vii) four representative paths in the CLTG. (viii) the final traversal of nodes.

might also help one to eliminate a number of edges (e.g., the one from a in the first statement to d in the third statement). Eliminating all these illegal (unacceptable) edges (transitions) gives us the CLTG shown in Figure 2(vi). Note that, the graph in Figure 2(vi) contains very few edges, compared to Figure 2(v).

The optimization process described in the next section operates on the CLTG. Before moving to the optimization phase, let us first formalize the constraints that allow us derive a CLTG from a given LTG. A *constrained layout transition graph,* written $CLTG(V', E')$, is a subgraph of the $LTG(V, E)$ such that $V' = V$ and E' contains all the edges in E except those that can lead to an incorrect or infeasible code transformation (transition). Note that the construction of the CLTG subsumes both the intrastatement constraints (i.e., evaluation rules that need to be obeyed when processing the RHS expression) and the interstatement constraints (i.e., data dependence and other constraints between different statements). In mathematical terms, to build the CLTG, the following edges of the LTG should be dropped:

—Any edge $(v_i, v_j) \in E$ such that $v_i \in s_{kR}$, $v_j \in s_{k'R}$ with $k \neq k'$
—Any edge $(v_i, v_j) \in E$ such that $v_i \in s_{kR}$, $v_j \in s_{k'L}$ with $k \neq k'$
—Any edge $(v_i, v_j) \in E$ such that $v_i \in s_{kL}$ and $v_j \in s_{k'L}$, where $k \neq k'$ and $s_{k'R} \neq \emptyset$
—Any edge $(v_i, v_j) \in E$ such that traversing this edge would break expression evaluation rules or data dependence.

While a CLTG shows only the legal edges, this does not necessarily mean that traversing the shown edges will always result in correct codes. For example, accessing two nodes, v_i and v_j consecutively, itself may not break any dependence; however, after this access sequence, it may not be possible to generate legal code due to the a new restriction (in the access order) resulting from the said transition between v_i and v_j. Note also that this may not be detectable

during the construction of the CLTG since we do not know at that point what the access sequence (traversal order) will be (e.g., whether the edge (v_i, v_j) will actually be visited by the traversal).

5. SOLUTION

We formulate the problem of modifying a given basic block code for effective use of the available block buffer of k elements as one of determining a path cover [Cormen et al. 1990] and a traversal order on the CLTG. To generate correct code (i.e., to preserve the semantics of the basic block), we impose the following conditions on the traversal order:

—Each node in the CLTG (i.e., a variable occurrence in the basic block) should be visited.
—For a given layer in the CLTG corresponding to the statement s_k, all nodes in s_{kR} should be visited before the node in s_{kL}.
—Once the traversal reaches a layer corresponding to the statement s_k, it should finish all variables in that layer (i.e., the set $s_{kL} \cup s_{kR}$) before moving to another layer.
—All data dependences and other restrictions such as latency constraints or expression evaluation constraints should be preserved.

Based on these observations, our approach determines a traversal order and during the traversal it also transforms the underlying code fragment (basic block). *The objective of the traversal* (*and that of transformation*) *is to minimize the cost of traversal.* In our context, the *traversal cost* is defined as the number of transitions (i.e., successive variable accesses) that do not have a corresponding edge in the CLTG. This is because each such transition accesses two variables (one after another) that reside in different virtual lines, and consequently, the second access (in the transition) cannot exploit the data currently residing in the block buffer (i.e., results in a block buffer miss). Recall that we assume a single block buffer that can hold k consecutive elements. Therefore, one way of minimizing the cost is to traverse as many edges from the CLTG as possible.

In the following, we present the description of an algorithm that takes as input a CLTG and generates as output a traversal (an access sequence) and all the necessary (interstatement and intrastatement) transformations to obtain this access sequence. Given a CLTG, the algorithm first detects the *longest directed path* (i.e., the path that contains the maximum number of edges in the same direction). It then transforms the portion of the CLTG (which contains a subset of the statements in the original basic block) in accordance with this longest path. Finding the longest path in a given directed graph is straightforward, and takes $O(N^3)$ time where N is the number of nodes in the graph [Cormen et al. 1990]. Transforming the program code in accordance with the longest path is more challenging as explained next.

Let us now discuss the transformations imposed by a path. Figure 3 shows the situations that may demand code (access sequence) transformations during the optimization process. The first situation corresponds to the case where

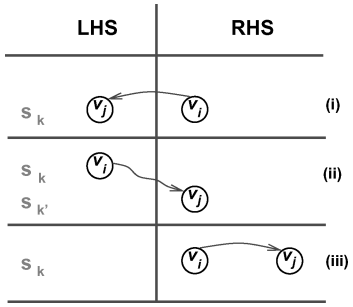

Fig. 3. Example cases that might require code transformations.

there is an edge from a node v_i in s_{kR} to the node v_j in s_{kL}. In this case, the RHS node in question should be made the last node accessed on the RHS (if it is not already the last node accessed on the RHS). The second situation is the one in which we have an edge (in the CLTG) from the LHS node v_i of a statement s_k to a node v_j in $s_{k'R}$ of statement $s_{k'}$ where $k \neq k'$. In this case, we may need two types of transformations: first, if $s_{k'}$ and s_k are not consecutive, that should be made consecutive (an inter-statement transformation); second, if v_j is not the first variable accessed in the $s_{k'R}$, it should be made so (an intrastatement transformation). The third situation corresponds to the case in which we have an edge from a node v_i in s_{kR} to another node v_j in s_{kR}. In this case, we need an intra-statement transformation that should bring these two nodes together. In cases where a variable v_j is needed to be both the first variable accessed in $s_{k'}$ (due to the edge from s_k to $s_{k'}$) and be the last variable accessed (due to the intra-statement edge in $s_{k'}$), we need a conflict resolution scheme.

We illustrate the operation of this transformation mechanism using the example in Figure 2. Recall that in this example we have three virtual lines: {a,b,c,d}, {e,f,g,h}, and {i,j,k,l}. The longest path in the CLTG in Figure 2(vi) is marked (1) in Figure 2(vii). It contains two nodes (b and a) from the first statement and three nodes (a, d, and b) from the third statement. In order to realize this path (i.e., to touch the variables at runtime in the order indicated by this path), variable b in the first statement should be the variable that is accessed last on the RHS (just before the LHS variable a), the third statement should be moved to just below the first statement (an inter-statement transformation) to ensure successive accesses to variable a, the variables a and d (in the original third statement) should be accessed one after another (already satisfied as there are only two variables on the RHS), and d should be the last variable accessed on the RHS. Note that after performing these transformations the access orders for variables i and j (in the first statement) are also fixed. The approach next moves to the second longest path (marked (2) in Figure 2(vii)) and performs the transformations indicated by it. It is important to stress that the transformations that would be performed based on the second (longest) path cannot override (nullify) those performed based on the longest (previously optimized) path. Note that after processing path (2), the access order of all the variables in the code is fixed. The

approach verifies this by checking the remaining paths (which are marked (3) in the figure) and making sure that the nodes contained in them have all been processed.

Figure 2(viii) shows the final access sequence (traversal). The dashed edges (arrows) correspond to transitions that contribute the cost of the traversal as they do not have corresponding edges in the CLTG. In this particular case, the overall cost is 4. Note that the cost of the traversal directly corresponds to the number of block buffer misses. It is easy to verify that the traversal cost would be 11 had we not performed any transformation. Considering that the total number of variable accesses in this example is 14, we observe a significant impact of the code transformations used.

6. MULTIPLE BASIC BLOCKS

Up until now, we have focused on a single basic block and presented a technique that employs access sequence transformations built upon a graph-based representation. In reality, most embedded codes have multiple basic blocks connected to each other through control dependences such as loop structures and conditional branching. Below we present an approach that optimizes a given procedure that might contain multiple basic blocks using a mix of data (variable) layout (storage sequence) and access sequence transformations.

It should be noted that if we restrict ourselves to access sequence transformations, it is relatively straightforward to extend the technique presented in the previous section to multiple basic blocks as follows. Assume that the procedure being optimized is represented using a control flow graph (CFG) where each node represents a basic block and each directed edge represents a possible transition (transfer of control) between two basic blocks. The proposed approach starts with the most frequently executed basic block (which might be determined through profiling and static analysis where applicable) and optimizes it as explained in the previous section. It then visits each basic blocks in the order of decreasing execution frequency and uses the same algorithm to transform its access sequence.

A more interesting problem (and that is the one addressed in this section), however, is to optimize a given CFG using both access sequence and storage sequence transformations. In order to achieve this we need to employ a storage sequence optimization strategy. Panda et al. [1997] and Panda [1998] present a powerful strategy for this purpose. Their strategy first obtains an access sequence and then builds a *closeness graph* (CG). The CG has a node for every variable used in the code. The two nodes, v_i and v_j, in the CG are connected using an edge with a weight of w_{ij} if the distance between them in the access pattern is $\leq k$ (line size) and the corresponding edge (transition) is taken w_{ij} times. After constructing the CG, the next step is to group the variables into clusters of k elements. As explained in Panda et al. [1997], a higher edge weight in the CG between two variable represents a potential reduction in the number of memory accesses (i.e., increase in the cache hit rate) if the two variables are stored close to each other in memory. They propose a heuristic that determines the temporally correlated variables and stores them consecutively as much as

possible. Note that in our terminology, each cluster in Panda et al. [1997] corresponds to a virtual line.

In principle, an approach that integrates access sequence optimizations and storage sequence optimizations in a unified framework should generate better results than pure access sequence optimizations and pure storage sequence optimizations.

Before discussing how the approach in Panda et al. [1997] can be combined (integrated) with the access transformation technique presented in this paper, let us address a subproblem that will be needed in the integration process. The problem that we address below is to optimize the access and storage pattern of a basic block assuming that only some (not all) of the variables have fixed memory locations and the remaining variables do not have fixed locations (i.e., they are yet to be assigned to storage locations). Let us call these two groups of variables $\mathcal{F}_A$ and $\mathcal{F}_N$, respectively. The objective of the optimization process is then to find an access sequence (to come up with an access sequence transformation) and to find a storage sequence for the variables in $\mathcal{F}_N$. We propose the following three-step approach to solve this subproblem:

—Use access sequence transformations to obtain an access sequence (for the variables in $\mathcal{F}_A$) compatible with the storage sequence of the variables in $\mathcal{F}_A$. These transformations will typically modify the access sequence of some of the variables in $\mathcal{F}_N$ as well. Let us call the set of these variables $\mathcal{F}_{N'}$ ($\subseteq \mathcal{F}_N$).

—Use storage sequence optimizations for the variables in $\mathcal{F}_{N'}$ to obtain a storage sequence (variable layout) which is compatible with the access sequence determined in the previous step.

—If the set $\mathcal{F}_N - \mathcal{F}_{N'}$ is not empty, then determine a suitable storage sequence for the variables in this set. In most of the cases encountered in practice, this set is empty.

Note that the first step uses our approach explained in the previous section whereas the second step use Panda et al.'s [1997] approach.

We are now ready to present our unified approach to procedure-wide (global) optimization for effective utilization of block buffering. Our approach operates on the CFG representation of the procedure and starts by ranking the basic blocks according to decreasing execution frequencies (which might be obtained through static analysis or profiling as mentioned earlier). It then starts the optimization process with the most frequently executed basic block and optimizes this basic block using only storage sequence optimizations proposed by Panda et al. [1997]. After optimizing this block, the storage order of the variables accessed by this block is known. The approach then moves to the second most frequently executed basic block and optimizes this basic block using the three-step approach discussed in the previous paragraph. After optimizing this block, the set of variables whose storage locations are determined is updated and the approach moves to optimize the next basic block, and so on. Therefore, two distinguishing characteristics of the approach are:

—during the optimization process, it gives priority in optimization to the most frequently executed basic blocks; and

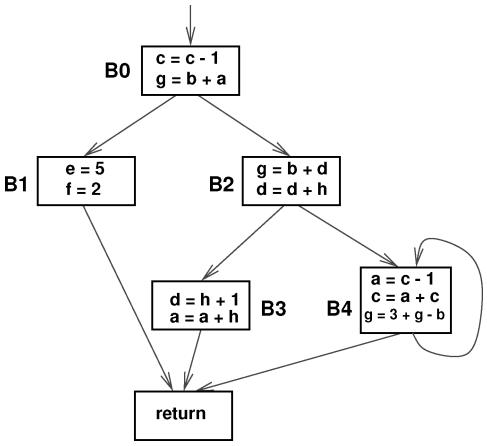

Fig. 4. An example control flow graph (CFG).

—it propagates variable layouts between basic blocks to reach a globally (procedure-wide) acceptable solution.

It should be noted that once a memory location has been determined for a variable (during optimization of a basic block), it is never changed later (during the optimization of a less frequently executed basic block); that is, the approach does not backtrack.

To illustrate the operation of this approach, we consider the CFG shown in Figure 4. Let us assume without loss of generality that the basic blocks are ordered according to decreasing execution frequency as `B4`, `B0`, `B2`, `B3`, and `B1` and that $k = 2$. Note that a total of eight variables, namely `a`, `b`, `c`, `d`, `e`, `f`, `g`, and `h`, are referenced in this CFG. Our approach starts with `B4` and optimizes it using Panda et al.'s scheme which indicates that variables `a` and `c`, and similarly variables `g` and `b` should be stored consecutively; that is, `a` and `c` should form a virtual line and `g` and `b` should form a virtual line. Having fixed the storage order (sequence) for four variables, we move to `B0` (the second most frequently executed basic block). Since `c` and `a` (and similarly `b` and `g`) have already been decided to be stored one after another (in the same virtual line), we apply a commutativity transformation and transform the statement `g = b + a` to `g = a + b`. The next block to be optimized is `B2`. In optimizing this block, our approach first applies commutativity transformation to the first statement to makes accesses to variables `b` and `g` successive. It then captures the temporal proximity between `d` and `h` and stores them consecutively (in a single virtual line). After that, it moves to `B3` and changes the access order of variables in the second statement (`a = a + h`) to bring accesses to `d` (in the first statement) and `h` (in the second statement) together. Finally, the approach handles `B1` by storing `e` and `f`, consecutively (in the same virtual line). A possible storage order satisfying these transformations is `a,c,g,b,d,h,e,f` in which we have four virtual lines: {`a,c`}, {`g,b`}, {`d,h`}, and {`e,f`}.

7. EXTENSIONS TO THE BASIC APPROACH

7.1 Multiple Blocks

So far, we have considered only a single block buffer. The proposed technique can be extended to work with multiple block buffers. In this case, after obtaining the storage and access sequences, we need to place the virtual lines in memory so as to minimize the number of conflict misses (in the block buffer) between the virtual lines. We can achieve this by employing the cluster optimization proposed by Panda et al. [1997]. The idea is to build a *cluster* (*virtual line*) *interference graph* in which each node represents a virtual line (determined by our approach in the previous section), and the weight of an edge between two nodes indicates the number of potential conflict misses in the block buffer (cache) due to these two clusters. Since this cluster (virtual line) assignment problem can be shown to be NP-hard, a heuristic can be used so that the nodes (clusters) with high number of potential conflict misses (i.e., high edge weights) are assigned to consecutive memory locations to minimize the chances for conflict misses.

7.2 Impact of Register Allocation

Until now, we have assumed no register allocation; that is, all variables are accessed from memory (through the cache plus block buffer system). A register allocation mechanism can have two impacts on the energy consumption of a memory system. First, it filters out a number of memory references that would otherwise end up in the cache and (maybe) off-chip memory. This filtering, in general, leads to a reduction on the overall memory system energy consumption. Second, it can distort the regularity in data accesses to cache memory. This second impact can be expected to lower the percentage benefits due to our strategy. On one hand, there would be fewer references to the block buffer, hence less scope for optimization. On the other hand, those references that end up in the block buffer exhibit less regularity. This obviously makes a high-level compiler optimization that targets block buffering less predictable.

There are two ways of handling the interaction between our approach and register allocation. The first approach is to do nothing. In other words, we can continue to optimize the code as if all data references would go to cache, and expect that the references that actually go to cache at runtime would, hopefully, still exhibit a regularity so that our optimization would bring some benefit. The second approach is more involved as explained below.

Note that our strategy as explained so far, has been applied at the source-level. This obviously makes it difficult to take into account the interactions with low-level optimizations performed by the back-end compiler. However, it is also possible to apply our approach at the low level, possibly even after register allocation and instruction scheduling. Such a strategy will bring two main benefits. First, we can clearly see (at the low level) which references are register allocated (therefore, they do not need to be considered by our approach). Second, we can have a better idea about the orders of loads and stores. Consequently, our modified approach proceeds as follows. The CLTG is constructed after register allocation has been performed and all register-allocated variables are omitted

from consideration. In the multiple basic blocks case, once the storage locations of all variables that are not register-allocated have been determined, the remaining variables are assigned locations from a separate region of memory; that is, they do not interfere with the former group of variables.

It should be stressed, however, that modifying the access pattern at low level is more challenging than doing it at high level. For example, going from an evaluation order `a + b + c` to `c + a + b` in the low-level representation of the code not only changes the order of loads but may also affect the register usage. However, this potential problem does not create an excessively negative impact with our current implementation as we do not move load/store instructions beyond basic block boundaries.

7.3 Impact of Load/Store Re-ordering

As mentioned earlier in the article, if the back-end optimizer changes the relative order of loads/stores, the effectiveness of our strategy may reduce as such re-orderings can make access pattern assumptions at the high level invalid. Implementing our optimization strategy at the low level solves this load/store problem partially (i.e., as far as the compiler's view of the order of loads/stores is concerned, if we run our strategy after instruction scheduling). This has drawbacks in that reordering the instructions (that were ordered by the instruction scheduler to reduce stalls) may have an adverse effect by increasing stalls. A detailed modeling of the potential interaction between instruction scheduling and our strategy is beyond the scope of this article. We are currently exploring the problem of integrating instruction scheduling with our strategy. This problem can be much worse for superscalar processors that perform out-of-order (OOO) execution through which the order of instructions can be modified at runtime in a manner that may not be predictable at compile time.

It should also be mentioned that many embedded systems (in particular, those aiming at real-time environments) do not perform aggressive load/store scheduling at runtime as such optimizations make the execution time prediction highly difficult, in general [Wolf 2001].

8. EXPERIMENTAL EVALUATION

This section provides results of the experiments we performed to evaluate the proposed optimization scheme. Our scheme has been implemented within the SUIF compilation framework [Amarasinghe et al. 1996] and has been evaluated using four codes: `int_mxm`, an integer matrix multiply program (that contains one initialization and one multiplication nest); `full_search`, a motion estimation code; `fft`, a discrete Fourier analysis code; and `flt`, a filtering routine. Since our strategy targets codes that make frequent use of scalar variables, a pre-pass in the compiler unrolls the loops in the code completely, and replaces array references with their scalar counterparts. For example, array references such as `a[1][2]` and `b[5][3]` are converted to scalar variables a_{12} and b_{53}, respectively. After this optimization, each element can be treated independently from others.

The data set sizes (respectively the number of basic blocks) for int_mxm, full_search, fft, and flt are 196K (respectively, 2), 71K (respectively, 11), 224K (respectively, 8), and 128K (respectively, 12). For each code, four different versions have been evaluated: the original code, a version that uses only storage layout optimizations (denoted s_opt) [Panda et al. 1997], a version that uses only access sequence optimizations (denoted a_opt), and a version that uses both storage layout and access sequence optimizations (denoted s+a_opt).

Our first set of experiments measure the energy benefits of our approach when no register allocation is performed. The input to our implementation is a code written in C. The SUIF pass implemented first applies loop unrolling and converts array references to scalars as mentioned above. After this step, it applies our strategy explained in Section 6, which transforms access patterns and determines storage locations for variables. Then, this optimized output code in C and the original code are compared with each other. To eliminate register allocation, all the versions are compiled using a custom Sparc-based back-end where no scalar variable is permanently register allocated. The block buffer miss rates are obtained through the use of an in-house simulator built upon the Shade infrastructure [Cmelik and Keppel 1994]. To calculate energy consumptions, we employ the on-chip energy formulations in Shiue and Chakrabarti [1999]. All the graphs shown here are for an 8K, direct-mapped, write-back data cache with a single block buffer. Experiments with 2-way and 4-way associative caches exhibited similar trends; so, they are not presented here. Also, when we increase the number of block buffers (to 4 and 8), we obtained larger energy savings (though slightly sub-linear in terms of the number of block buffers). Beyond 8 buffers, we noticed that the buffers themselves consume a large amount of energy.

Figure 5 gives the percentage improvements (reductions) in data cache energy consumption with respect to the original version (unoptimized code) for two different values of k (1 and 4). Note that, the unoptimized code also takes advantage of block buffering depending on the amount of block-level data locality in the code. We can make two major observations from these graphs. First, there is no clear choice between s_opt and a_opt as none of them dominates the other. This means that both access pattern optimizations and storage pattern optimizations need to be considered by the compiler. Second, the unified strategy (s+a_opt) outperforms the other two optimization schemes over all codes and buffer sizes. When $k = 1$, the average percentage improvements brought about by s_opt, a_opt, and s+a_opt are 26.1%, 26.7, and 48.2%, respectively.

Figure 6 presents reductions in the overall data memory system energy (main memory energy plus data cache energy, including the block buffer). We see that the average energy reductions (over all codes and all buffer sizes) due to s_opt, a_opt, and s+a_opt are 12.6%, 12.2%, and 21.4%, respectively. These results clearly demonstrate that the unified strategy generates much better results than pure storage pattern or pure access pattern optimizations.

Next, we performed a set of experiments in which the impact of register allocation on the effectiveness of our approach was gauged. Figure 7 gives the percentage reductions in data cache energy consumption with respect to the original version. To perform these experiments, we first coded a simple back-end,

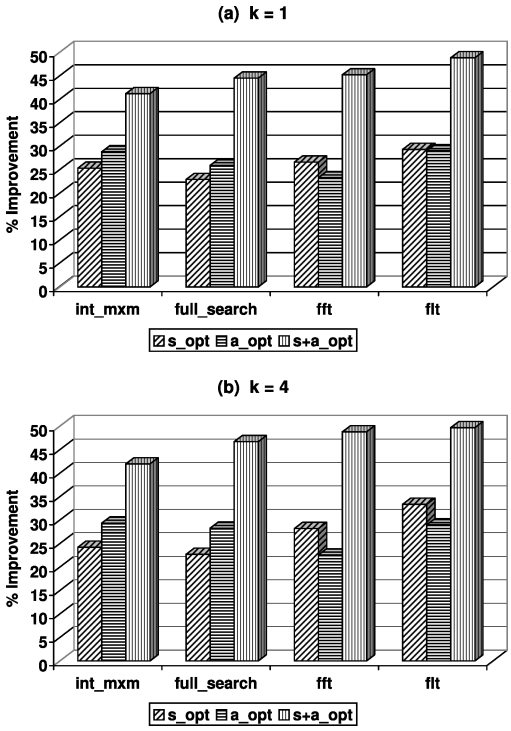

Fig. 5. Data cache energy percentage improvement over original codes (without register allocation).

which takes the SUIF output (of the unoptimized code) and generates Sparc code. Then, our back-end implementation, after register allocation, applied the optimization technique discussed in this article. The register allocator employed here closely follows the Chaitin's graph-coloring-based approach [Chaitin 1982]. We see from Figure 7 that, when $k = 1$, the average percentage data cache energy improvement due to `s+a_opt` is 35.7%, which is 12.5% worse than when no register allocation is employed. Similarly, the results illustrated in Figure 8 show that (when register allocation is used) our approach improves overall data memory energy by 16.4%. These results indicate that, even in the existence of an aggressive register allocation, large energy benefits are possible by using compiler-directed storage pattern and access pattern optimizations in concert.

In the next set of experiments, we measured the impact of load/store reordering. For this purpose, we modified our back-end so that loads and stores are moved to exploit as much instruction level parallelism (ILP) as possible (without performing any load speculation). Since the observed trends are the same for all the codes in our experimental suite, we present only the results for `flt` when the register allocator is activated and the unified optimization strategy (`s+a_opt`) is used. The results presented in Figure 9 show that, with load/store re-ordering, the data cache energy and overall data memory energy brought by our approach is 24.7% and 9.1%, respectively. It should be noted that load/store speculation can even take these benefits to lower values.

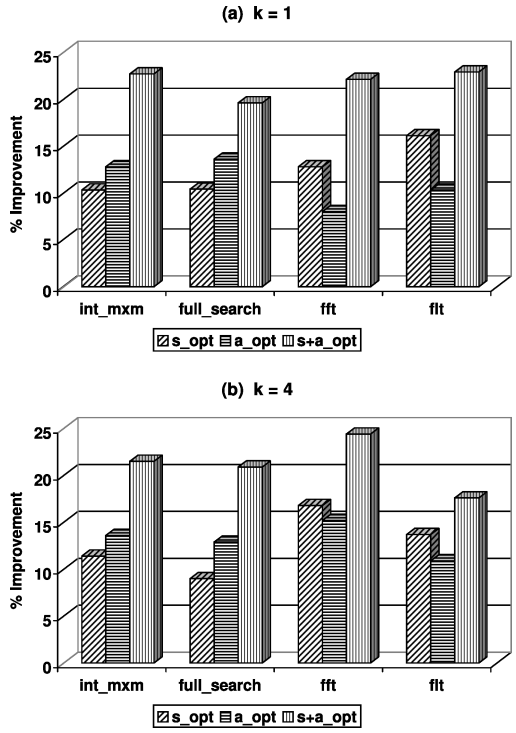

Fig. 6. Overall memory energy percentage improvement over original codes (without register allocation).

Before moving to our concluding remarks, we would like to discuss two important issues: performance impact of our approach and its impact on overall energy savings. The impact of our approach on performance (execution cycles) can be evaluated under two scenarios. In the first scenario, we assume that the block buffer access and the cache access are serialized. That is, we first access the block buffer and then to the cache if we miss in the block buffer. In this case, the impact of our approach on performance is directly dictated by the block buffer hit rate. Specifically, if the block buffer hit rate is high, we can expect performance benefits (in addition to energy benefits). If, on the other hand, the block buffer hit rate is low, this can affect the overall performance since each miss in the block buffer brings 1 cycle additional penalty. The second scenario, whose implementation we followed up to this point in the article, is based on the implementation strategy suggested by Ghose and Kamble [1999], in which the buffer and cache accesses are performed concurrently, using a two-phase clock. In this case, the cache access is terminated early if the block buffer access turns out to be a hit. In this case, the performance behavior of our scheme is less affected by the buffer hit rate, compared to the first scenario described above.

Figure 10 gives the performance behavior of our strategy (`s+a_opt`) under these two scenarios. Each bar in this graph gives the execution cycles, normalized with respect to the case without any optimization (i.e., the original version). Two trends can be observed from this graph. First, our approach

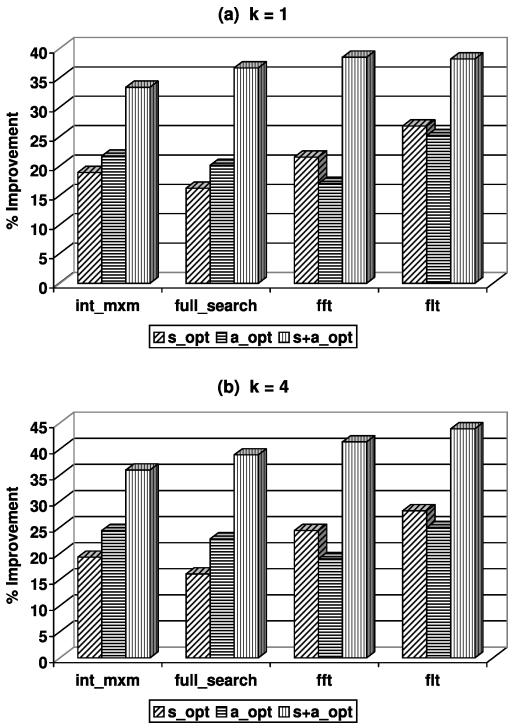

Fig. 7. Data cache energy percentage improvement over original codes (with register allocation).

improves overall performance under both the scenarios. The average execution cycle reductions for the first (serial access) and the second (parallel access) scenarios are 9.4% and 13.1%, respectively. Second, as expected, the execution cycle savings are lower with the first scenario. The reason that we still achieve execution cycle savings with this scenario is the fact that the block buffer hit rates are generally high, thereby removing the potential cache access in many circumstances. The experimental results so far demonstrate that our approach is beneficial from both energy and performance viewpoints.

Finally, it is also important to consider the overall energy savings achieved by our approach. We start by observing that since our approach does not increase execution cycles (under both the scenarios discussed above), we do not incur any extra leakage consumption (over the original codes). Our experimentation with these benchmark codes showed that, without any optimization, the data memory system energy (including the data cache and the data main memory) constitutes about 31% of the total energy consumption (for a simple five-stage pipelined embedded processor, excluding the input/output units), indicating that data memory system is an important target from the energy angle. Since the results in Figures 6 and 8 show that our approach is able to reduce overall data memory system energy by 16–21% when averaged over all benchmark codes in our experimental suite, one can expect around 5–6.5% savings, on the average, in total energy consumption.

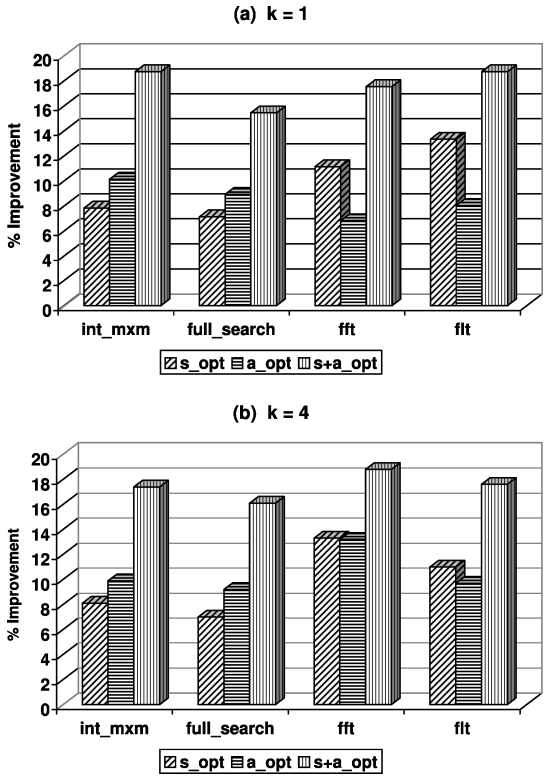

Fig. 8. Overall memory energy percentage improvement over original codes (with register allocation).

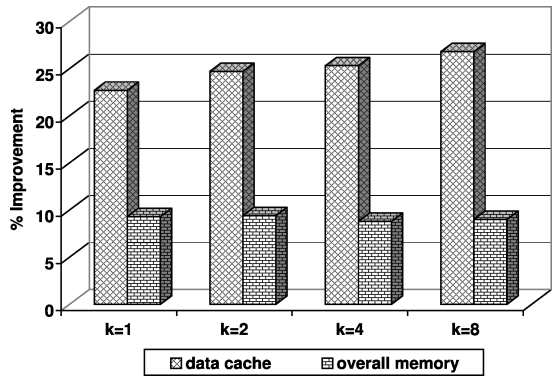

Fig. 9. Data cache energy and overall memory energy percentage improvements over original codes when load/store re-ordering is performed (with register allocation).

9. CONCLUSIONS AND FUTURE WORK

This article presented a compiler-based approach that modifies code and variable layout to take better advantage of block buffering. The proposed technique is aimed at a class of embedded codes that make heavy use of scalar variables.

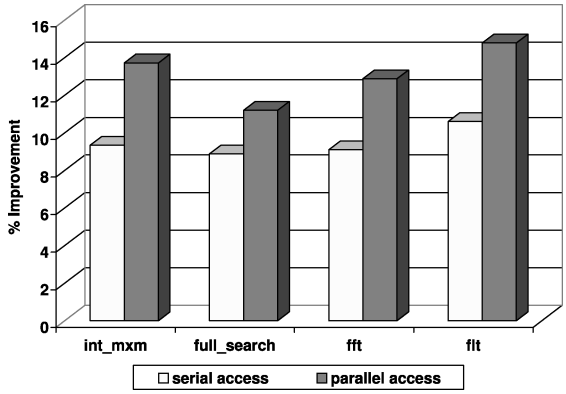

Fig. 10. Overall performance improvement over original codes (without register allocation and when $k = 1$.)

Unlike previous work that uses only storage pattern optimization, we use an integrated approach that employs both code restructuring and storage pattern optimizations. Our solution works for whole procedures, that is, multiple basic blocks. We have implemented our solution in an experimental compiler. Experiments with several codes demonstrate that our solution results in up to 24% savings in overall memory energy without register allocation and 18% with register allocation.

Work in progress includes the investigation of different ways of combining storage layout and code restructuring transformations, incorporating partitioning of variables for multiple block buffers, and studying the impact of static single assignment. We also plan to investigate how storage pattern decisions made for a given procedure can be propagated to other procedures in the same application.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

AMARASINGHE, S. P., ANDERSON, J. M., WILSON, C. S., LIAO, S.-W., MURPHY, B. R., FRENCH, R. S., LAM, M. S., AND HALL, M. W. 1996. Multiprocessors from a software perspective. *IEEE Micro*.

BELLAS, N., HAJJ, I., STAMOULIS, G., AND POLYCHRONOPOULOS, C. 1998. Architectural and compiler support for energy reduction in the memory hierarchy of high-performance microprocessors. In *Proceedings of the 1998 ACM/IEEE International Symposium on Low Power Electronics and Design*, ACM, New York, 70–75.

CATTHOOR, F., WUYTACK, S., GREEF, E. D., BALASA, F., NACHTERGAELE, L., AND VANDECAPPELLE, A. 1998. *Custom Memory Management Methodology—Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers.

CHAITIN, G. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, ACM, New York, 98–105.

CHILIMBI, T., LARUS, J., AND HILL, M. 2000. Making pointer-based data structures cache conscious. *IEEE Comput*.

CMELIK, B. AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ACM, New York, 128–137.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

EDMONDSON, J., FISCHER, T. C., JAIN, A. K., MEHTA, S., MEYER, J. E., PRESTON, R. P., RAJAGOPALAN, V., SOMANATHAN, C., TAYLOR, S. A., WOLRICH, G. M., RUBINFELD, P. I., BANNON, P. J., BENSCHNEIDER, B. J., BERNSTEIN, D., CASTELINO, R. W., COOPER, E. M., DEVER, D. E., AND DONCHIN, D. R. 1995. Internal organization of the Alpha 21164, a 300 MHz 64-bit quad-issue CMOS RISC microprocessor. *Digit. Tech. J. 7*, 1, 119–135.

ESAKKIMUTHU, G., VIJAYKRISHNAN, N., KANDEMIR, M., AND IRWIN, M. 2000. Memory system energy: Influence of hardware-software imizations. In *Proceedings of the ISLPED'00 ACM/IEEE International Symposium on Low Power Electronics and Design*. ACM, New York.

GHOSE, K. AND KAMBLE, M. 1999. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ACM, New York, 70–75.

KAMBLE, M. AND GHOSE, K. 1997. Energy-efficiency of vlsi caches: a comparative study. In *Proceedings of the International Conference on VLSI Design*, 261–267.

KANDEMIR, M., CHOUDHARY, A., RAMANUJAM, J., AND BANERJEE, P. 1998. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the International Symposium on Microarchitecture*.

KANDEMIR, M., CHOUDHARY, A., SHENOY, N., BANERJEE, P., AND RAMANUJAM, J. 1999. A linear algebra framework for automatic determination of imal data layouts. *IEEE Trans. Parall. Distrib. Syst. 10*, 2, 115–135.

KANDEMIR, M. AND RAMANUJAM, J. 2000. Data relation vectors: A new abstraction for data imizations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

KELLY, W., MASLOV, V., PUGH, W., ROSSER, E., SHPEISMAN, T., AND WONNACOTT, D. 1995. The Omega library interface guide. Technical Report CS–TR–3445, CS Dept., University of Maryland.

KIN, J., GUPTA, M., AND MANGIONE-SMITH, W. 1997. The filter cache: An energy-efficient memory structure. In *Proceedings of the MICRO-30*, 184–193.

KULKARNI, C., CATTHOOR, F., AND MAN, H. D. 2000. Advanced data layout imization for multimedia applications. In *Proceedings of the Workshop on Parallel and Distributed Computing in Image, Video, and Multimedia Processing, held in conjunction with IPDPS'00*.

KULKARNI, C., MIRANDA, M., GHEZ, C., CATTHOOR, F., AND MAN, H. D. 2001. Cache conscious data layout organization for embedded multimedia applications. In *Proceedings of the 4th ACM/IEEE Design and Test in Europe Conference*. ACM, New York.

MELLOR-CRUMMEY, J., WHALLEY, D., AND KENNEDY, K. 1999. Improving memory hierarchy performance for irregular applications. In *Proceedings of the ACM International Conference on Supercomputing*. ACM, New York.

PANDA, P. 1998. Memory minimization and exploration for embedded systems. Ph.D. dissertation. University of California, Irvine, Irvine, CA.

PANDA, P., DUTT, N., AND NICOLAU, A. 1997. Memory data organization for improved cache performance in embedded processor applications. *ACM Trans. Des. Auto. Elec. Sys. 2*, 4.

SHIUE, W. AND CHAKRABARTI, C. 1999. Memory exploration for low power embedded systems. Tech. Rep., Arizona State Univ.

SU, C. AND DESPAIN, A. 1995. Cache design tradeoffs for power and performance imization: A case study. In *Proceedings of the ISLPED95, ACM/IEEE International Symposium on Low Power Electronics and Design*, ACM, New York, 63–68.

WOLF, W. 2001. *Computers as Components: Principles of Embedded Computing System Design*. Morgan-Kaufmann, San Francisco, CA.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, MA.

Received January 2003; revised March 2003; accepted March 2005