Compiler Support for Block Buffering

Mahmut Kandemir* J. Ramanujam† Ugur Sezer‡

Abstract

On-chip caches consume a significant fraction of energy in current microprocessors. Hence, hardware techniques such as block buffering have been developed and shown to be effective in reducing on-chip cache energy consumption. We are not aware of any software solutions to exploit block buffering. This paper presents a compiler-based approach that modifies both code and variable layout to effectively exploit block buffering, and is aimed at the class of embedded codes that make heavy use of scalar variables. Unlike previous work that uses only storage pattern optimization, our solution integrates both code restructuring and storage pattern optimization. Experimental results on a set of complete programs demonstrate that our solution leads to significant energy savings.

1 Introduction

On-chip caches are a major source of energy consumption in current microprocessors. For example, Edmondson et al. [2] 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 [5]) 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). The key idea is to keep the most recently accessed cache line 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 depends on the block-level locality exhibited by the application). As noted by Ghose and Kamble [4], 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 [9] propose a single block buffer structure; Ghose and Kamble [4] 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.

A simplified block buffering scheme is depicted in Figure 1. This is similar to the architecture proposed by Ghose and Kamble [4]. 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 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 normal caches.

While previous research [9, 4, 6, 3] 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 paper, we present a compiler-based approach that modifies code and variable 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 [11] and the references therein). While previous work such as [7] 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 paper). In comparison, the integrated approach proposed in this paper employs *both* storage pattern and access pattern optimizations. Specifically, this paper 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.

Overall, we believe 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.

The remainder of this paper is organized as follows. Section 2 describes the problem. Section 3 formulates it on a graph-based structure, and presents our solution strategy for the single basic block (a straight line of code without branching/conditionals) case along with a brief discussion of how we handle multiple basic blocks. Section 4 presents experimental data showing the effectiveness of the proposed technique. Section 5 concludes with a summary and an outline of the planned future work.

2 Problem Description

The success of a block buffering scheme is strongly dependent on the variable access pattern. Let us 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 (i.e., aligned to the cache line boundary):

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

The original access sequence is `b,c,a,d,b,c` and does not take any advantage of the block buffer. If the code is transformed to

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

we obtain a new access sequence (`c,b,a,b,d,c`) with a 50% (3/6) block buffer hit rate. Note that it would also be possible to optimize the original code fragment above using storage (variable layout) optimizations by transforming the original storage pattern `a,b,c,d` into `b,c,a,d`.

*Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA. (kandemir@cse.psu.edu). Supported in part by NSF grant CCR-0093082 and by the Pittsburgh Digital Greenhouse.

†Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803, USA. (jxr@ee.lsu.edu). Supported by NSF grant CCR-0073800 and NSF Young Investigator Award CCR-9457768.

‡Department of Electrical and Computer Engineering, University of Wisconsin–Madison, Madison, WI 53706, USA. (sezer@ece.wisc.edu)

ISLPED'01, August 6-7, 2001, Huntington Beach, California, USA.

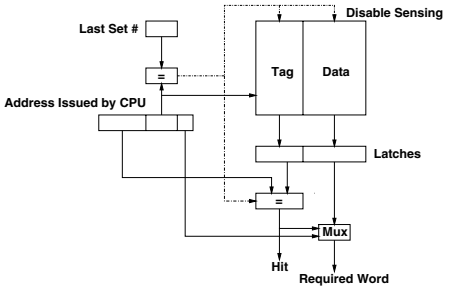

Figure 1: Operation of block buffering

We can conclude from this example 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 available option.

3 Representation and Solution

We represent the problem using a graph-based structure and solve it using a longest path algorithm. We define two data elements as *neighbors* if they map onto 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), 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 (that is, 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 variable(s) 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 intra-statement and inter-statement 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 [11] 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 paper and is the main data structure which the optimizations we employ operate on.

To illustrate how an 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} = {`a`} and s_{1R} = {`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 bi-directional 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 variables. Data dependence constraints might also help one to eliminate a number of edges (e.g., the one from `b` in the third statement to `b` in the first statement). Eliminating all these illegal (unacceptable) edges (transitions) gives us the CLTG shown in Figure 2(vi). Note that as compared to Figure 2(v), the graph in Figure 2(vi) contains very few edges.

The optimization process described next 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 $LTG(V, E)$ with the same nodes 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 intra-statement constraints (i.e., evaluation rules that need to be obeyed when processing the RHS expression) and the inter-statement 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 (**Note**: s.t. means 'such that')

- Any edge $(v_i, v_j) \in E$ s. t. $v_i \in s_{kR}$, $v_j \in s_{k'R}$ with $k \neq k'$
- Any edge $(v_i, v_j) \in E$ s. t. $v_i \in s_{kR}$, $v_j \in s_{k'L}$ with $k \neq k'$
- Any edge $(v_i, v_j) \in E$ s. t. $v_i \in s_{kL}$ 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

3.1 Single Basic Block

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 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 traversal order:

- Each node in the CLTG (i.e., a variable occurrence in the basic block) should be visited.

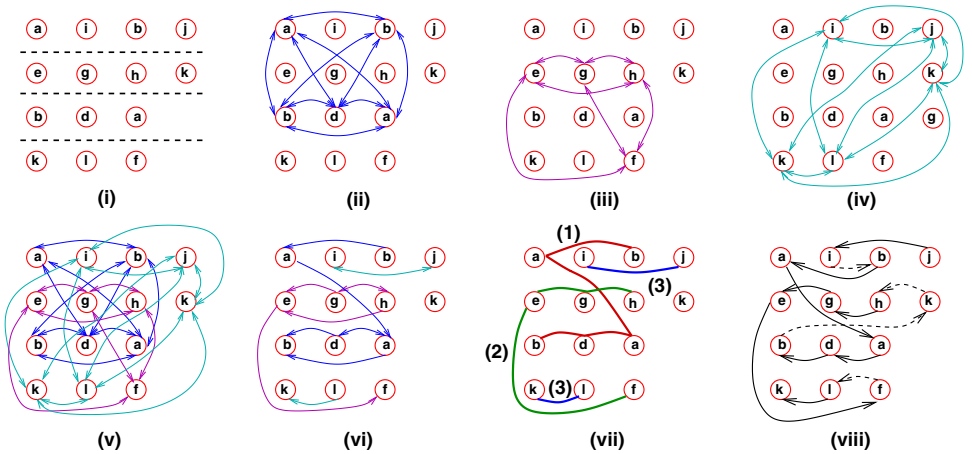

Figure 2: (i) Four layers (corresponding to four statements) for a basic block. (ii-iv) sub-LTGs induced by different virtual lines. (v) the overall LTG. (vi) the corresponding CLTG. (vii) four representative paths in the CLTG. (viii) the final traversal of nodes.

- 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 (that is, 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 observed.

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 (inter-statement and intra-statement) 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. Due to lack of space, we only present an example and omit the details of transforming the program code in accordance with the longest path, which is a challenging problem.

We illustrate the operation of this transformation mechanism using the example in Figure 2. 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.

3.2 Multiple Basic Blocks

Our procedure-wide (global) optimization 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. [7]. 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 (i) the fact that it gives priority in optimization to the most frequently executed basic blocks, and (ii) that it propagates variable layouts between basic blocks to reach a globally (procedure-wide) acceptable solution. It should be noted that once a memory location is 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).

4 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 [10] and evaluated using four codes: `int_mxm`, an integer matrix multiply pro-

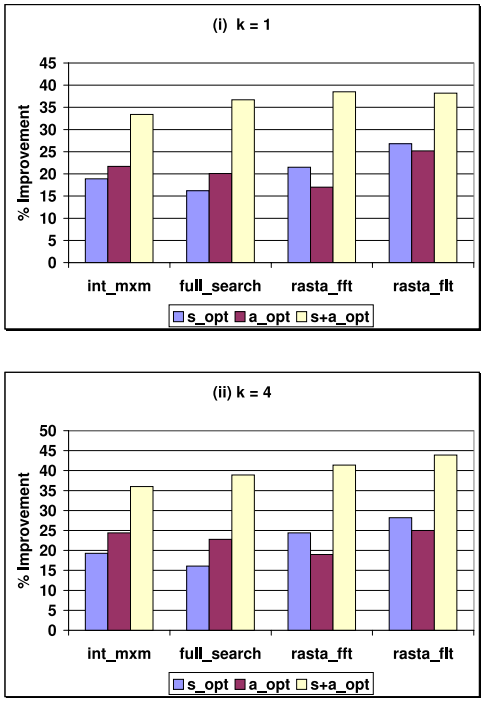

Figure 3: Data cache energy percentage improvement over original

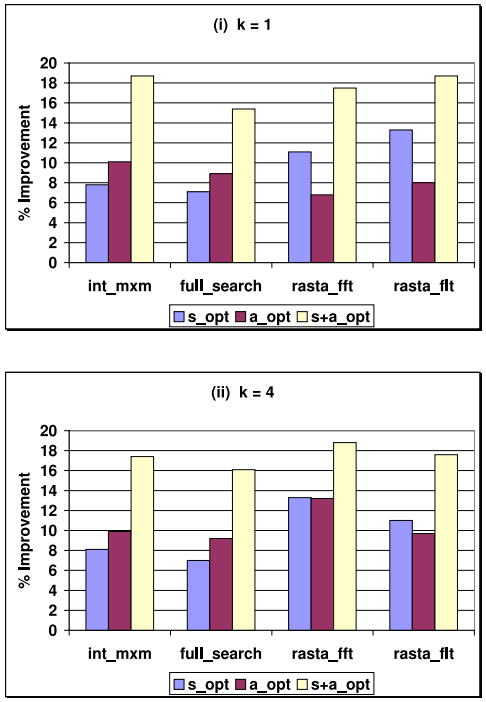

Figure 4: Overall memory energy percentage improvement over original

gram (that contains one initialization and one multiplication nest); `full_search`, a motion estimation code; `rasta_fft`, a discrete Fourier analysis code; and `rasta_flt`, a filtering routine. The data set sizes (resp. the number of basic blocks) for `int_mxm`, `full_search`, `rasta_fft`, and `rasta_flt` are 196K (resp. 2), 71K (resp. 11), 224K (resp. 8), and 128K (resp. 12), respectively. For each code, four different versions have been evaluated: original code, a version that uses only storage layout optimizations (`s_opt`) [7], a version that uses only access sequence optimizations (`a_opt`), and a version that uses both storage layout and access sequence optimizations (`s+a_opt`). The block buffer miss rates are obtained through the use of an in-house simulator built upon the Shade infrastructure [1]. To calculate energy consumptions, we employ the on-chip energy formulations in [8]. 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 produced similar results; so, they are omitted due to lack of space. Also, when we increase the number of block buffers (to 4 and 8), we obtained larger energy savings (though slightly sublinear in terms of the number of block buffers). Beyond 8 buffers, we noticed that the buffers themselves consume a large amount of energy.

Figure 3 gives the *percentage improvements* (*reductions*) in *data cache energy consumption* with respect to the original version for two different values of k (1 and 4). 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. Figure 4 presents *reductions in the overall data memory system energy* (*main memory energy plus data cache energy*). An amount of 4.95nJ is assumed to be spent per main memory access [8, 3]. We see that the average (over all codes and all buffer sizes) energy reductions brought about by `s_opt`, `a_opt`, and `s+a_opt` are 9.8%, 9.5%, and 17.5%, respectively.

5 Conclusions and Future Work

This paper presented a compiler-based approach that modifies code and variable layout to take better advantage of block buffering, for a class of embedded codes that make heavy use of scalar variables. Unlike previous work that uses only storage pattern optimization, we use an integrated approach that targets whole programs and employs both code restructuring and storage pattern optimizations. We have implemented our solution in a compiler using SUIF [10]. Experiments with several codes demonstrate that our solution results in up to 17% savings in overall memory energy. Work in progress includes the investigation of different ways of combining storage layout and code restructuring transformations, and incorporating partitioning of variables for multiple block buffers.

References

[1] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In Proc. *ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, May 1994, pp. 128–137.

[2] J. Edmondson et al. Internal organization of the Alpha 21164, a 300 MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, Vol. 7, No. 1, 1995, pp. 119–135.

[3] G. Esakkimuthu, N. Vijaykrishnan, M. Kandemir, and M. Irwin. Memory system energy: influence of hardware-software optimizations. In Proc. *ISLPED'00*.

[4] K. Ghose and M. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In Proc. *ISLPED'99*, pp. 70–75, 1999.

[5] M. Kamble and K. Ghose. Energy-efficiency of VLSI caches: a comparative study. In Proc. *Int. Conf. VLSI Design*, 1997.

[6] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: an energy-efficient memory structure. In Proc. *MICRO-30*, 1997.

[7] P. Panda, N. Dutt, and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Trans. Des. Auto. Elec. Sys.*, Vol. 2, No. 4, 1997.

[8] W. Shiue and C. Chakrabarti. Memory exploration for low power embedded systems. Tech. Rep., Arizona State Univ., 1999.

[9] C. Su and A. Despain. Cache design tradeoffs for power and performance optimization: a case study. In Proc. *ISLPED'95*, pp. 63–68.

[10] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 1994.

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