Dataflow Analysis for Energy-Efficient Scratch-Pad Memory Management

Guangyu Chen and Mahmut Kandemir
Computer Science and Engineering Department
The Pennsylvania State University
University Park, PA 16802, USA
{gchen,kandemir}@corporation.com

ABSTRACT

Scratch-Pad Memories (SPMs) are a serious alternative to conventional cache memories in embedded computing since they allow software to manage data flowing from and into memory components, resulting in a predictable behavior at runtime. The prior studies considered compiler-directed SPM management using both static and dynamic approaches. One of the assumptions under which most of the proposed approaches to data SPM management operate is that the application code is structured with regular loop nests with little or no control flow within the loops. This assumption, while it makes data SPM management relatively easy to implement, limits the applicability of those approachs to the codes involve conditional execution and complex control flows. To address this problem, this paper proposes a novel data SPM management strategy based on dataflow analysis. This analysis operates on a representation that reflects the conditional execution flow of the application and, consequently, it is applicable to a large class of embedded applications, including those with complex control flows.

Categories and Subject Descriptors

D.3.m [**Software**]: Programming Languages—*Miscellaneous*

General Terms

Languages

Keywords

Scratch Pad Memory (SPM), data flow analysis, compiler

1. INTRODUCTION

Recent research has demonstrated that conventional hardware-managed cache hierarchies may not necessarily be the best option for embedded application codes that manipulate large data structures using nested loops. The main reason for this is the fact that these applications have compiler analyzable data access patterns and an optimizing compiler would be in a better position than hardware to manage data transfers across memory hierarchies. One of the architectures employed for enabling software-managed memory space is scratch-pad memories (SPMs). An SPM is similar to a conventional cache memory; the difference is that, while data transfers into and out of caches are determined by hardware at runtime, those in SPM are determined by a compiler at compilation time. In the context of real-time embedded computing, SPMs are preferable as they allow accurate worst case execution time analysis, which is not easy with conventional caches.

Several research groups investigated SPMs from hardware and software perspectives. As far as software support is concerned, one of the most important questions is how to decide which subset of data should be kept in the SPM at any given time. The answer to this question is critical since SPM space is typically limited (it is on-chip) and cannot accommodate all the data manipulated by a reasonably-sized embedded application. Among the solutions proposed for data SPMs are static approaches (e.g., [8]) that keep SPM-resident data fixed throughout execution and dynamic approaches (e.g., [6]) that change the SPM contents based on dynamic data access and reuse patterns. One of the implicit assumptions under which most of the proposed approaches to SPM management operate is that the application code is structured with regular loop nests with little or not control flow within and between the loops. This assumption, while it makes SPM management relatively easy to implement (as we do not deal with dynamic conditional control flow), limits the applicability of those approaches to the codes involve conditional execution and complex control flows (both intra and inter-procedural). When applied to such codes, these techniques need to be conservative, which means in practice missing some opportunities for further code optimization and losing some potential performance/energy benefits.

Focusing on large array-based embedded applications, this paper proposes a compiler-guided *dataflow analysis* based approach to data SPM management. The proposed scheme is very general and applicable to a large class of embedded applications, including those with complex control flows.

The rest of this paper is organized as follows. Section 2 discusses related work on SPM management. Section 3 revises several important concepts and terms related to dataflow analysis and control flow graphs. Section 4 gives our dataflow analysis formulation for SPM management and explains how it is used for optimization. Section 5 gives our concluding remarks.

2. RELATED WORK

There exist several prior studies on using SPMs for instruction accesses. For example, Sias et al [10] present compiler techniques, which arrange for 70-99% of the fetched operations to come from a

ISLPED'05, August 8–10, 2005, San Diego, California, USA

statically managed 256-instruction loop buffer, allowing instruction fetch power savings and eliminating branch penalties. Bellas et al. [2] present an SPM management scheme for instruction accesses. Steinke et al. [11] focus on a strategy for placing program and data objects into SPM for saving energy. As compared to our approach presented in this paper, all these schemes focus on instruction accesses. In comparison, we target data references. Also, in most of these studies, the SMP management is static; that is, the set of data elements that are mapped to SPM is decided before the execution and are fixed for the entire execution. In contrast, in our approach, the set of data elements mapped to SPM is dynamically changed as the execution progresses. Lee et al. [7] focus on reducing the energy consumption due to instruction accesses using a software-managed SPM (called loop cache). While their scheme is also dynamic, it is fundamentally different from ours as it focuses on instruction accesses. In a similar way, our work is also different from that of Steinke et al [4], where they propose a solution to the same problem using integer linear programming (ILP). Note that some of the current embedded processors, such as SH-DSP, StarCore SC140 and ST Microelectronics ST120, also support software-controlled loop caches.

On the data accesses side, several studies focused on the use of SPMs. Panda et al. [8] present a powerful static data partitioning scheme for efficient utilization of scratch-pad memory. Their approach is oriented towards eliminating the potential conflict misses due to limited associativity of on-chip cache. This approach benefits applications with a number of small (and highly reused) arrays that can fit in the SPM. Benini et al. [3] discuss an elegant memory management scheme that is based on keeping the most frequently used data items in a software-managed memory (instead of a conventional cache). Kandemir et al. [6] propose a dynamic SPM management scheme for data accesses. Their framework uses both loop and data transformations to maximize the reuse of data elements stored in the SPM. Hallnor and Reinhardt [5] propose a software-managed cache architecture and a data replacement algorithm. Wang et al. [12] discuss a framework for analyzing the flow of values and data reuse for on-chip memories. In contrast to these studies, our work focuses on improving the energy behavior of an SPM, is based on dataflow analysis, and is applicable to a larger class of applications.

3. REVIEW OF DATAFLOW ANALYSIS

Dataflow analysis reveals opportunities for optimization by reasoning about the runtime flow of data values at compile-time. It is performed on a control flow graph (CFG), which is usually build from the source code of a program. A CFG is a directed graph. Each node in a CFG represents a basic block – a sequence of consecutive program statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except maybe at the end [1]. Each directed edge from one basic block to another indicates a potential control flow during execution from the former to the latter. CFG is a conservative representation of the program since many of the edges in CFG may not exercised in a typical execution.

Dataflow analysis builds and solves dataflow equations to discover facts about what can occur at runtime during execution. Dataflow equations have following forms:

$$in[b] = \bigcup_{p \in pred(b)} out[p,b];$$
$$out[p,b] = gen[p] \cup (in[p] - kill[p,b]), p \in pred(b);$$

where $gen[p]$ and $kill[p,b]$ represent the set of properties generated by and killed by basic block p, respectively. On the other hand, $out[p,b]$ gives the set of properties that propagate from p to b along the edge $p \rightarrow b$ in the CFG, and $in[b]$ gives the set of properties that reach the beginning of b. In the remainder of this paper, we present a dataflow analysis for SPM management (a problem that has not been considered previously in a dataflow framework) and evaluate its effectiveness.

4. DATAFLOW FORMULATION

Our dataflow analysis is performed in three steps: (1) computing the constraints on array access instructions, (2) constructing reuse graphs, and (3) determining the regions of the arrays that need be loaded into or evicted from SPM. Our goal is to decide what to bring to SPM and what to displace from it for each point in the program code. Although we focus on a data SPM in this paper, our analysis can also be used to generate hints that can help applications make better use of a conventional hardware-managed data cache.

In this section, we limit our analysis to a single perfectly-nested and normalized loop nest. A perfectly nested loop nest is a loop nest whose instructions are all in the inner-most loop. A normalized loop nest has the following properties: (1) the lower bound of each loop in the nest is zero; (2) the upper bound of each loop is either a constant or a function of the iteration indices of the outer loops; and (4) the increment (i.e., step size) of each loop is one.[1] In Section 4.4, we will extend our analysis to more general cases.

A perfectly-nested and normalized loop nest has the following form:

$$\text{for } \vec{I} = \vec{0} \text{ to } \vec{U} \ \{ \text{ loop body } \}$$

Vector $\vec{I} = (i_1, i_2, ..., i_n)^T$ is the iteration vector of the loop nest, and i_k ($1 \leq k \leq n$) is the iteration index of the k^{th} loop. Particularly, i_1 and i_n are the iteration indices for the outer-most and inner-most loops, respectively. Vector $\vec{U} = (u_1, u_2, ..., u_n)^T$ is the upper bounds vector, where u_k ($1 \leq k \leq n$) is the upper bound for the k^{th} loop. Further, u_1 is a constant, and u_k ($1 < k \leq n$) is the functions of vector $(i_1, i_2, ..., i_{k-1})^T$, i.e., $u_k = f_k(i_1, i_2, ..., i_{k-1})$. In addition, we assume that f_k is an affine function.

Another assumption under which our analysis operates is that the subscript vector for each array access instruction is an affine function of the iteration vector and loop-invariant constants. An affine function can be represented as $M\vec{I} + \vec{o}$, where $\vec{I}$ is an iteration vector with n elements, M is an n-column matrix (referred to as *access matrix*), and $\vec{o}$ (referred to as *offset*) is a constant vector with n elements. For example, $X[i_1+2i_2-1, 3i_1-4i_2+5]$ can be represented as $X[M\vec{I} + \vec{o}]$, where

$$M = \begin{pmatrix} 1 & 2 \\ 3 & -4 \end{pmatrix} \text{ and } \vec{o} = \begin{pmatrix} -1 \\ 5 \end{pmatrix}.$$

4.1 Computing Constraints

In a loop nest containing conditional branches, a basic block may not be executed at every iteration. For each basic block b in the body of loop nest $\mathcal{L}$, we use dataflow analysis to compute the set of iterations at which b may be executed. For b_0, the entry block of the loop body of $\mathcal{L}$, we have $in[b_0] = \{\vec{I} | \vec{0} \preceq \vec{I} \preceq \vec{U}\}$, where $\vec{U}$ is the upper bounds for $\mathcal{L}$. For all the blocks, we have $gen[b] = \phi$. For $p \in pred(b)$, $kill[p,b]$ is the set of iterations where control flow does not transition from p to b. Let us assume that the last instruction of basic block p is a conditional branch depending on the value of a boolean expression e. We have $kill[p,b]$ equal to the set of iterations where $e = false$ if the control transitions from p to

[1]There exist compiler techniques [13] to make a loop nest normalized.

b when $e = true$. On the other hand, $kill[p, b]$ is equal to the set of iterations where $e = true$ if the control transitions from p to b when $e = false$. If the last instruction of p is not a conditional branch, we have $kill[p, b] = \phi$. All the sets in our analysis are expressed using Presburger formulas [9]. We do not consider the sets that cannot be expressed in Presburger formulas.

Let us assume that C_i is the set of iterations where array access instruction A_i is executed. We have:

$$C_i = in[b], \text{ where } A_i \text{ is in basic block } b.$$

In the rest of this paper, we refer to C_i as the *constraint on* A_i. For convenience, we use the following notation:

$$C_i(f(\vec{I})) \equiv \{\vec{I} \mid f(\vec{I}) \in C_i\}, \text{ where } f \text{ is a function of } \vec{I}.$$

4.2 Constructing Reuse Graphs

Let us first define a function $\delta(A_i, A_j)$ that computes the reuse vector from A_i to A_j, where A_i and A_j are array access instructions with same access matrix M. We compute $\delta(A_i, A_j)$ as follows. For A_j at iteration $\vec{I}+\vec{d}$ to reuse the array element used by A_i at iteration $\vec{I}$, we must have:

$$M\vec{I} + \vec{o}_i = M(\vec{I} + \vec{d}) + \vec{o}_j,$$

where $\vec{o}_i$ and $\vec{o}_j$ are the offset vectors of A_i and A_j, respectively. By assuming $\vec{b} = \vec{o}_i - \vec{o}_j$, we obtain:

$$M\vec{d} = \vec{b}. \tag{1}$$

At this point, we consider three possible cases:

Case 1. $\vec{b} \neq \vec{0}$, and $\vec{d}$ has integer solution that is lexicographically greater than $\vec{0}$:

$$\delta(A_i, A_j) = \min_{M\vec{d}=\vec{b} \wedge \vec{d} \succeq \vec{0}} \{\vec{d}\};$$

Case 2. $\vec{b} = \vec{0}$, and A_i can be executed before A_j within the same loop iteration[2]: $\delta(A_i, A_j) = \vec{0}$;

Case 3. Otherwise: $\delta(A_i, A_j) = \infty$.

If $M\vec{d} = \vec{0}$ has only the trivial solution (i.e., $\vec{d} = \vec{0}$), equation (1) has at most one solution. In this case, each instruction can never use an array element more than once (i.e., there is no reuse). On the other hand, if $M\vec{d} = \vec{0}$ has non-trivial solutions, equation (1) can have an infinite number of solutions. In this case, an instruction may use the same array element multiple times. To address this situation, we define $R(M)$ as the minimum sized set such that for all integer vector $\vec{y}$ such that $\vec{y} \succ \vec{0}$ and $M\vec{y} = \vec{b}$, there exist n non-negative integers $c_1, c_2, ..., c_n$ and n vectors $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n \in R(M)$ such that:

$$\vec{y} = \sum_{i=1}^{n} c_i \vec{x}_i.$$

It can be proven that any reuse vector from A_i to A_j can be represented as:

$$\vec{d}_{i,j} = \delta(A_i, A_j) + \sum_{k=1}^{|R(M)|} c_k \vec{x}_k,$$

where $c_k \in Z^+ \cup 0 \wedge \vec{x}_k \in R(M)$.

Now, we can define $RG(\mathcal{L}, X, M)$, the *Reuse Graph* for loop nest $\mathcal{L}$, array X, and access matrix M, as follows:

$$RG(\mathcal{L}, X, M) \equiv (V, E, \chi),$$

where

$$\begin{aligned} V &\equiv \{A_i | A_i \text{ in } \mathcal{L} \wedge A_i \text{ accesses } X[M\vec{I} + \vec{o}_i]\}; \\ E &\equiv \{(A_i, A_j) | A_i, A_j \in V \wedge \big((i \neq j \wedge \delta(A_i, A_j) \neq \infty) \\ &\quad \vee (i = j \wedge R(M) \neq \phi)\big)\}; \end{aligned}$$

and, for each $(A_i, A_j) \in E$, we have:

$$\chi(A_i, A_j) = \begin{cases} \{\delta(A_i, A_j)\}, & i \neq j \\ R(M), & i = j \end{cases}$$

Let us assume that a is the number of instructions in the loop nest $\mathcal{L}$ that access array X using access matrix M. The number of the vertices in the corresponding reuse graph $RG(\mathcal{L}, X, M)$ is equal to a, and the number of the edges in this graph cannot exceed a^2.

4.3 Determining the Contents of SPM

For instruction A_i that accesses array X, let us define the set $RI(N, i)$ as the set of loop iterations such that each element of X accessed by A_i at iteration $\vec{I} \in RI(N, i)$ will be reused *at least* N times. Figure 1 gives the algorithm for computing $RI(N, i)$. The idea behind this algorithm can be briefly explained as follows. First, we find all the paths start from A_i with N edges in the given reuse graph. And then, we compute the constraint on each of these paths. For a given path $p = (A_{k_0}, A_{k_1}, A_{k_2}, ..., A_{k_N})$, we compute the constraint on p as:

$$C(p) = \bigcap_{j=0}^{N-1} C_{k_j}(\vec{I} + \sum_{r=0}^{j} \vec{d}_{k_r, k_{r+1}}),$$

where C_{k_j} is the constraint on A_{k_j}, and $\vec{d}_{k_r, k_{r+1}} \in \chi(A_{k_r}, A_{k_{r+1}})$.

For example, given a path (A_0, A_1, A_2), if an array element is used by A_0 at $\vec{I}$, and then reused by A_1 at $\vec{I} + \vec{d}_{0,1}$, and by A_2 at $\vec{I} + \vec{d}_{0,1} + \vec{d}_{1,2}$, then $\vec{I}$ must satisfy the following constraint:

$$C_0(\vec{I}) \cap C_1(\vec{I} + \vec{d}_{0,1}) \cap C_2(\vec{I} + \vec{d}_{0,1} + \vec{d}_{1,2}).$$

Finally, we compute $RI(N, i)$ as follows:

$$RI(N, i) = \bigcup_{p \in P(i) \wedge |p| \geq N+1} C(p),$$

where $P(i)$ is the set of paths starting from A_i, and $|p|$ denotes the length of path p.

Further, our algorithm in Figure 1 uses the thresholds $\vec{t}$ and $\vec{T}$ to filter out the reuse paths that span too many iterations since we are interested only in the reuses that will take place in the near future.

For ease of discussion, we define:

$$RA(X, N, \vec{L}, \vec{U}) \equiv \bigcup_{\forall A_i \in \mathcal{L}} \{X[M\vec{I} + \vec{o}_i] \mid \vec{I} \in RI(N, i) \wedge \vec{U} \preceq \vec{I} \prec \vec{L}\}.$$

$RA(X, N, \vec{L}, \vec{U})$ contains the elements of array X that are used during the iterations between $\vec{L}$ and $\vec{U}$, and these elements are reused at least N times.

The compiler determines the set of data elements that are mapped to SPM at any given time, it also inserts code to the program for loading the mapped data elements into the SPM, and, if necessary, evicting some data elements from the SPM to make space for the new data. For ease of implementation, our compiler inserts the SPM management codes at the entry of each iteration of the n^{th} loop in the loop nest. Consequently, the set of data elements that are mapped to SPM is changed at iterations $\vec{I}_0, \vec{I}_1, ..., \vec{I}_n$, where $\vec{I}_0 = \vec{0}$ and

$$\vec{I}_{k+1} - \vec{I}_k = (\underbrace{0, 0, ..., 0}_{n-1}, 1, 0, ..., 0).$$

[2]Since A_i and A_j can be in different conditional branches within the loop, they are not necessarily executed together in every loop iteration.

During the iterations between $\vec{I}_k$ and $\vec{I}_{k+1}$ ($0 \leq k < n$), the set of elements mapped to the SPM is not changed. Loading data to SPM incurs performance overheads, which are amortized by the future access to these data. Therefore, only the data elements that will be reused more than N times should be loaded into the SPM, where N is the SPM loading threshold. Assuming that S_k is the set of elements of array X in the SPM right before iteration $\vec{I}_k$, the set of the elements of array X that need to be loaded into SPM at iteration $\vec{I}_k$ ($0 \leq k < n$) can be determined as:

$$Load(k) = RA(X, N, \vec{I}_k, \vec{I}_{k+1}) - S_k.$$

Further, we have:

$$S_k \subseteq \bigcup_{i=0}^{k-1} RA(X, N, \vec{I}_i, \vec{I}_{i+1}) = RA(X, N, \vec{I}_0, \vec{I}_k).$$

If $|S_k \cup RA(X, N, \vec{I}_k, \vec{I}_{k+1})| > C_{SPM}$, where C_{SPM} is the capacity of the SPM, we need to evict some elements from the SPM. Specifically, we evict the elements that are used less than M times during the iterations between $\vec{I}_k$ and $\vec{I}_{k+1}$, where M ($M < N$) is the SPM eviction threshold. Therefore, the set of elements that can be evicted is:

$$Evict(k) = S_k - RA(X, M, \vec{I}_k, \vec{I}_{k+1}).$$

Note that the elements not in $Evict(k)$ are not evicted. If we cannot find enough space by eviction, we do not load all the elements in $Load(k)$. This policy ensures that each array element that has been loaded into SPM remains in the SPM until it is reused by at least $N - M$ times. After that, it might be evicted due to the insufficient SPM space (depending on the reuse patterns of other elements). The thresholds N and M are determined by the capacity of SPM (C_{SPM}), the cost for loading a data element into the SPM ($Cost_{load}$), and the per access costs for the SPM ($Cost_{SPM}$) and the main memory ($Cost_{Memory}$). The following constraints on M and N should hold. First, the total size of data elements in the SPM cannot exceed the capacity of the SPM:

$$\sum_{\forall X, \forall k} |RA(X, N, \vec{I}_k, \vec{I}_{k+1})| \leq C_{SPM}.$$

Second, the cost for loading must be amortized by the future accesses:

$$(N - M)(Cost_{Memory} - Cost_{SPM}) > Cost_{load}.$$

4.4 Dealing with Non-Perfectly-Nested Loops

So far, we have discussed our analysis for perfectly-nested normalized loop nests. However, in reality, not the all loop nests found in embedded application codes are perfectly-nested and normalized. Wolfe [13] presents an algorithm for loop normalization. In this section, we briefly discuss how our analysis can be extended to analyze the non-perfectly-nested loops.

Non-perfectly-nested loop nests are processed in three steps. First, we transform a non-perfectly-nested loop nest into a perfectly-nested one by moving all the instructions into the inner most loop nest. To ensure the correctness of the program, in the most general case, these moved instructions are enclosed by a conditional branch instruction. Second, we perform the dataflow analysis discussed so far on the body of the transformed loop nest and then generate the SPM management code. Finally, we extract the conditional branch instructions that have been added in the first step out of the inner most loop to eliminate the unnecessary condition tests.

5. CONCLUDING REMARKS

This paper proposes a novel SPM management scheme built upon dataflow analysis over the control flow graph representation of the program. It is able to handle a large class of embedded applications (even those with complex conditional control flow).

$RG(\mathcal{L}, X, M) = (V, E, \chi)$ — reuse graph
$\vec{t}, \vec{T}$ — thresholds for reuse vectors
N — minimum number of reuses C_i — symbolic constraint on A_i
$D[i, j]$ — set of reuse vectors from A_i to A_j
$RI[N, i]$ — the set of loop iterations such that each element of X accessed by A_i at the iteration $\vec{I} \in RI_{N,i}$ will be reused at least N times.

```
// initialize
for each (A_i, A_j) ∈ E {
  D[i, j] = φ;
  for each d⃗ ∈ χ(A_i, A_j)
    if(d⃗ ≺ t⃗) {
      S = {Ī | C_i(Ī) ∧ C_j(Ī + d⃗)};
      if(S ≠ φ)
        D[i, j] = D[i, j] ∪ {d⃗};
    }
}
// computing RI[N, i] for all N and i
for each A_i ∈ V {
  S = {Ī | C_i(Ī)};
  RI[N, i] = visit(i, 0, 0⃗, S);
}

function visit(i, n, p⃗, C) {
  if(p⃗ ≻ T⃗) return false;
  for each j ∈ V such that D[i, j] ≠ φ
    for each d⃗ ∈ D[i, j] {
      C' = C ∩ C_j(I + p⃗ + d⃗);
      if(C' ≠ φ) {
        if(n = N)
          return C';
        else
          r = visit(j, n + 1, p⃗ + d⃗, C');
          return r;
      }
    }
}
```

Figure 1: Algorithm for computing $R(N, i)$ for array X in loop nest $\mathcal{L}$ with access matrix M. This algorithm finds all the paths with N edges in the reuse graph $RG(\mathcal{L}, X, M)$, and computes the constraint on each of these paths. $\vec{T}$ and $\vec{t}$ are the thresholds that filter out the reuse paths that span too many iterations since we are interested only in the reuses that will happen in the near future.

6. REFERENCES

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools (second edition)*. Addison-Wesley, Reading, Ma, 1986.

[2] N. Bellas, I. N. Hajj, C. Polychronopoulos, and G. Stamoulis. Energy and performance improvements in microprocessor design using a loop cache. In *International Conference on Computer Design*, 1999.

[3] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. In *IEEE Design & Test of Computers*, Apr. 2000.

[4] S. S. et al. Reducing energy consumption by dynamic copying of instructions onto on-chip memory. In *ISSS'02*, Oct. 2002.

[5] E. G. Hallnor and S. K. Reinhardt. A fully-associative software-managed cache design. In *International Conference on Computer Architecture*, 2000.

[6] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *the 38th DAC*, June 2001.

[7] L. H. Lee, B. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *ISLPED*, Aug. 1999.

[8] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad-memory in embedded processor applications. In *European Design and Test Conference (ED&TC'97)*, Mar. 1997.

[9] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Aug. 1993.

[10] J. Sias, H. Hunter, and W. Hwu. Enhancing loop buffering of media and telecommunication applications using low-overhead predication. In *the Annual International Symposium on Microarchitecture*, Dec. 2001.

[11] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[12] L. Wang, W. Tembe, and S. Pande. Optimizing on-chip memory usage through loop restructuring for embedded processors. In *9th International Conference on Compiler Construction*, Mar. 2000.

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