Memory Organization for Improved Data Cache Performance in Embedded Processors[*]

Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA

Abstract

Code generation for embedded processors creates opportunities for several performance optimizations not applicable for traditional compilers. We present techniques for improving data cache performance by organizing variables declared in embedded code into memory, using specific parameters of the data cache. Our approach clusters variables to minimize compulsory cache misses, and solves the memory assignment problem to minimize conflict cache misses. Our experiments demonstrate significant improvement in data cache performance (average 46% in hit ratios) by the application of our memory organization technique using code kernels from DSP and other domains on the LSI Logic CW4001 embedded processor.

1. Introduction

Embedded microprocessors are a common feature in modern electronic systems due to the advantages they offer in terms of flexibility, reduction in design time and full-custom layout quality [11]. These processors are often available in the form of *cores*, which can be instantiated as part of a larger system on a chip. This is feasible in current technology due to the relatively small area occupied by the processor cores, making the rest of the on-chip die area available for RAM, ROM, coprocessors, and other modules. A core-based design methodology is driven by demands for design reuse that ultimately results in reduced development time. Apart from the processors in the Digital Signal Processing domain (such as the TMS320 series from Texas Instruments), we also find microprocessors with relatively general purpose architectures available as embedded processors. An example of such a general purpose embedded processor is LSI Logic's CW4001 [2], which is based on the MIPS family of processors.

Generation of efficient code for embedded processors has been the subject of recent investigation [1, 5, 13]. Optimization techniques that improve the performance of application programs by exploiting the irregular architectures of embedded processors have been reported [5, 9, 10, 14].

An important determinant of performance in embedded systems is the interaction between the processor and external memory. General purpose embedded processors such as the CW4001 are equipped with on-chip instruction and data caches, which interface with larger off-chip memories. Since off-chip memory accesses usually stall the CPU execution for significant durations (each access could take 10-20 processor cycles, depending on the relative processor and memory access speeds), it is important to design the interface between cache and main memory carefully.

Cache misses can be classified into three categories: *compulsory* misses, *capacity* misses, and *conflict* misses [7]. In the computer architecture and compiler domains, many techniques for achieving cache miss reduction involve additional hardware assistance [6, 17, 3], which can often be expensive in terms of additional on-chip area. A well known compiler optimization technique called *blocking* combines strip mining and loop permutation to maximize temporal locality of reused data [16]. This technique helps in reducing capacity misses in data caches, but fails to take advantage of data placement strategies to reduce conflict misses.

In the embedded processors domain, code placement methods based on program traces for improvement of instruction cache performance have been reported [15]. A technique for estimation of instruction cache performance has also been reported [8]. However, no published literature exists on the improvement of data cache performance in embedded systems.

Embedded system design is characterized by certain features that traditional compilers typically do not consider in their optimizations. For example, compilers seldom take into account the specific cache parameters such as cache line size in their optimizations, because fast compilation speed requirements preclude the complex analysis procedures. However, in embedded systems, code generation can be tuned to the specific cache configuration to be used (or the specific configuration that is being currently explored). Fur-

[*]This work was partially supported by grants from NSF(CDA-9422095) and ONR(N00014-93-1-1348).

ther, the typical execution of only a single program and the absence of virtual addresses permits the compiler to asign the exact memory location occupied by the data. In this work, we exploit this situation to organize data in memory in order to minimize data cache misses.

2. Problem Description

Consider a direct-mapped cache of size C ($C = 2^m$) words, with a cache line size L words, i.e., L consecutive words are fetched from memory on a cache read miss. In our formulation, we assume a *write-back* cache with a *fetch-on-miss* policy [7], though the technique remains identical for other write policies, and is equally effective.

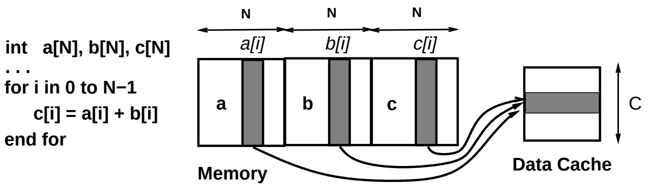

Figure 1. $a[i], b[i]$ **and** $c[i]$ **map into the same cache line**

We use a small example to illustrate the problem and our approach. Suppose the code fragment in Figure 1 is executed on a processor with the above cache configuration, where N is an exact power of 2, and $N > C$. Assuming one array element per memory word, let array a begin at memory location 0, b at N, and c at $2N$. Let $f(x)$ denote the cache line to which the program variable x is mapped. In a direct-mapped cache, the cache line containing a word located at memory address M, is given by: $(M \bmod C)/L$. In the above example, array element $a[i]$ is located at memory address: i. Similarly, we have *b[i]:* $N + i$ and *c[i]:* $2N + i$. We find that the corresponding cache lines to which each of them will be mapped are: $f(a[i]) = (i \bmod C)/L$; $f(b[i]) = ((N + i) \bmod C)/L = (i \bmod C)/L$ (*since N is divisible by C*); $f(c[i]) = ((2N + i) \bmod C)/L = (i \bmod C)/L$. In other words, $a[i], b[i]$, and $c[i]$ are mapped onto the same cache line (Figure 1).

Thus, the sequence of events that take place in one loop iteration is: a cache miss occurs while accessing $a[i]$; the line $f(a[i])$ is filled; accessing $b[i]$ now causes a miss, since, even if it was present in cache before, it was displaced by the last access to $a[i]$ (since $f(a[i]) = f(b[i])$); $f(a[i])$ is now filled by the line from b; the write to $c[i]$ also causes a miss, since $f(c[i]) = f(b[i])$; in a fetch-on-miss cache, this causes the same cache line to be displaced by elements of array c. The same cycle repeats in other iterations. In other words, *every memory access results in a cache miss!* Such memory access patterns are known to result in extremely inefficient cache utilization, especially because many applications deal with arrays whose dimensions are a perfect power of two [17]. In such situations, simply increasing the cache size does not present an efficient solution, because the cache misses are not caused due to lack of capacity. The conflict-misses can be avoided if the cache size C is made greater than $3N$, but this is often infeasible when N is large, and where feasible, there is an associated area and access time penalty incurred when cache size is increased. Reorganization of the data in memory results in a more elegant solution, while keeping the cache size relatively small.

One way of preventing the thrashing caused by excessive cache conflicts for this simple example is to pad L dummy memory words between two consecutive arrays that are accessed in an identical pattern in the loops. For this example, if array a begins at 0, array b begins at location: $N + L$ (instead of N) and array c begins at: $2N + 2L$ (instead of $2N$). We have:

$f(a[i]) = (i \bmod C)/L$
$f(b[i]) = ((N + L + i) \bmod C)/L = (i \bmod C)/L + 1$
$f(c[i]) = ((2N + 2L + i) \bmod C)/L = (i \bmod C)/L + 2$

This ensures that $a[i], b[i]$, and $c[i]$ are always mapped into different cache lines, and their accesses do not interfere with each other in the data cache. We extend this basic idea to organize scalars (Section 3) and arrays (Section 4).

3. Memory Organization of Scalar Variables

We assume that the scheduling and register allocation of the code has already been performed, and the sequence of accesses to variables is fixed.

3.1. Constructing the Closeness Graph

We first generate an Access Sequence, which is a graph representing memory references (loads and stores are treated alike) in the code. Figure 2(a) shows an example Access Sequence. The label 3 on edge $e \to a$ represents a loop with bound = 3. We then construct a *Closeness Graph* of the variables, which represents the degree of desirability for keeping sets of variables in the same vicinity in memory. E.g., if L words accessed successively from memory are placed in consecutive locations, a single memory access could fetch them all into cache, thereby reducing upto $L - 1$ extra memory accesses caused due to *compulsory* misses.

We define the *distance* between two nodes u and v in the access sequence as: *distance*(u, v) = number of *distinct* variable nodes encountered on a path from u to v, or v to u (including u and v). The Closeness Graph *CG(V,E)* is constructed from the Access Sequence by first creating a node $v \in V$ for every variable in A, and initializing all edge weights $e(u, v) = 0$. For all occurrences u' of variable u during traversal of the Access Sequence, we examine a window of width M preceding and following u'. For all node instances v' (of variable v) in this window (where *distance*$(u', v') \leq M$), we update the edge weight $e(u, v) = e(u, v) + k$, where k is the number of times control is expected to flow between u' and v' [1].

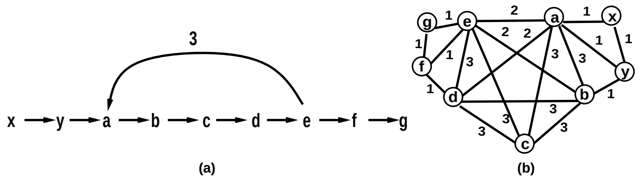

Figure 2. (a) Access Sequence (b) Closeness Graph

Figure 2(b) shows the Closeness Graph derived from the Access Sequence in Figure 2(a), with $M = 3$.

3.2. Grouping of Variables into Clusters

The next step is to group the variables into clusters of M words, where M is the number of words in a data cache line. Intuitively, a higher edge weight between two variables u and v in the Closeness Graph represents a reduction in the number of memory accesses, if the two variables are stored in the same cache line. I.e., we can reduce cache misses by clustering variables with higher edge weights into the same cache line. We formulate the problem of maximizing the sharing of the cache lines by closely correlated variables as follows: *Partition the nodes of the Closeness Graph* CG *(i.e., the set of n variables) into clusters of size M, so that the total weight of edges in all the clusters is maximized.* Since an exact solution to the above problem has a computational complexity of $O(n^M)$, we employ the following greedy heuristic, with complexity $O(Mn^2)$.

```
Procedure PerformClustering
Input: $CG(V,E)$: Closeness Graph; $M$: Cache Line Size
Output: Set $F$: Set of clusters of size $M$
    For each $u$ in $V$, find $S(u) = \sum_{v \in V} e(u,v)$
    Let $X$ = vertex set $V$ and $F = \phi$
    -- $X$ keeps track of variables not yet assigned to clusters
    while ($X \neq \phi$) do
        Let $u$ = vertex $v \in X$ with maximum $S(v)$          (i)
        Create new cluster $C = \{u\}$
        while (size of cluster $C \neq M$) and ($X \neq \phi$) do
            Let $x$ be the variable $\in X$ with maximum $T$,      (ii)
            where $T = \sum_{u \in C, v \in X-C} e(u,v)$    -- $x$ has max
            -- sum of edge weights with nodes already in $C$
            $C = C \cup \{x\}$    -- add x to cluster C
            $X = X - \{x\}$    -- remove x from X
        Set $e(u,v) = 0$ for all $(u \in C)$ or $(v \in C)$
        -- delete all edges to nodes in cluster C just formed
        Update $S(v)$ for all $v \in X$; $F = F \cup \{C\}$
```

When procedure *PerformClustering* is applied on the graph in Figure 2(b), node b is selected first (line i). Next, line (ii) causes nodes c and d to be selected into the first cluster C_1. When we have equal T values for multiple nodes, we select one at random. Nodes b, c, d, and all connecting edges are now deleted. From the resulting graph, a, e, and g form the next cluster C_2. The final clustering is: $C_1 : [b, c, d]; C_2 : [a, e, g]; C_3 : [x, y, f]$.

[1]The required values of k in case of conditionals and loops could be obtained by using profiling information. However, in this work, we use the often-used simplifying assumption that branch probability is 0.5 for an *if*-statement, and that the loop bounds are always known at compile time.

3.3. The Cluster Interference Graph

After grouping the variables into clusters of size M, we build an *Interference Graph (IG)* of the clusters, which represents the desirability to store clusters in memory, so that they do *not* map into the same cache line. Each node in the Interference Graph represents one cluster of variables obtained from procedure *PerformClustering*. A high edge weight between two nodes indicates a large number of conflict misses in the data cache, if the respective clusters were to map into the same cache line. We first convert the Variable Access Sequence A into a Cluster Access Sequence by renaming each node u in the sequence by the cluster C, where $u \in C$. We then construct the Cluster Interference Graph by first creating a node in IG for each cluster in F (the set of clusters), and then assigning edge weight $e(u,v)$ between nodes u and v to be the number of times the access to clusters u and v alternate along the execution path. E.g., for the variable sequence $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e, M = 2$, and the clustering $x : [a, d]; y : [b, c]$ and $z : [e]$, we have the Cluster Access Sequence $x \rightarrow y \rightarrow y \rightarrow x \rightarrow z$. This results in an Interference Graph *IG*, with edges $e(x, y) = 2, e(x, z) = 1$, and $e(y, z) = 1$. The pair of nodes x and y alternate twice in the execution path, due to the edges $x \rightarrow y$ and $y \rightarrow x$, causing $e(x, y) = 2$. The other pairs change orders only once. The composition rules to be followed for conditionals and loops are identical to those used for building the Variable Access Sequence (Section 3.1).

3.4. Memory Location Assignment

The final assignment of variables to memory locations should consider the clustering and conflict-penalty information in the Interference Graph. To minimize the conflict misses in the data cache during code execution, we need to ensure that cluster pairs with large edge weights do not map to the same cache line when we assign memory locations.

We define the *cost of a memory assignment* (C) as follows: $C = \sum_{x,y \in V(IG)} e(x,y) \times P(x,y)$ where $e(x,y)$ is the edge weight, and $P(x,y) = 1$ or 0, depending on whether memory locations for x and y map into the same cache line or not. Figure 3(b) shows the effect of a sample memory assignment for an IG with six clusters (Figure 3(a)), on a cache with four lines. We have $P(a,e) = P(b,f) = 1$ and $C = e(a,e) + e(b,f) = 1 + 3 = 4$. We solve the following problem: *Find an assignment of clusters in IG to memory locations, such that the assignment cost C is minimized.* This problem can be easily shown to

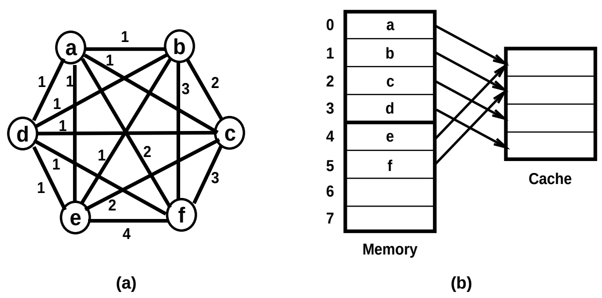

Figure 3. (a) Interference Graph (b) Memory Assignment

be NP-hard, by using a reduction from the Graph Colouring problem. We present below a greedy $O(n^2)$ heuristic (where n is the number of clusters) to solve the Cluster Assignment problem for a cache of size k that is similar to the *PerformClustering* procedure.

We proceed to make the memory assignments *page* by *page*, where a memory page consists of k cache lines – the size of the data cache. Note that k consecutive clusters in memory will never conflict in cache. We define the cost of assigning cluster u to cache line i as *cost*(u, i) = $\sum_{v \in X} e(u, v)$, where X is the set of clusters that have already been assigned to cache line i. This cost is the sum of edge weights of u with all nodes that are already assigned to map into cache line i.

```
Procedure AssignClusters
Input: $IG(V, E)$ – Cluster Interference Graph
Output: Assignment of Clusters to Memory Locations
    Sort the vertices of $IG$ in descending order of $S(u)$
    $--$ $S(u)$ is defined in Section 3.2
    Let $X$ be this sorted list of vertices
    while ($X \neq \phi$) do
        Create new page $P$ in memory
        while (size of page $P < k$) and ($X \neq \phi$) do
            $u$ = head of list $X$
            Assign $u$ to line $i$ of page $P$, where cost($u, i$) is
            minimum over $i = 0 \dots k-1$
            Delete $u$ from $X$
```

For the example IG in Figure 3(a), the page size is $k = 4$ lines. When we apply procedure *AssignClusters* on this example, we first sort the vertices in decreasing order of the sum of their incident edge weights: $f(13), c(11), e(9), b(8), a(6)$ and $d(5)$. Clusters f, c, e, and b are placed into the first page P_0. While attempting to assign a into the second page P_1, we find: cost$(a, 0) = 2$ (since $e(a, f) = 2$), cost$(a, 1) = 1$, cost$(a, 2) = 1$, and cost$(a, 3) = 1$. Thus, we choose a line within page P_1 that minimizes the cost, and assign a to line 1. Cluster d has: cost$(d, i) = 1$ for all i, so we assign line 0 of P_1 to d. The final assignment is: P_0: $(0 - f; 1 - c; 2 - e; 3 - b)$ and P_1: $(0 - d; 1 - a)$.

For an n-way associative cache, we use the same definition of *cost*, except that the cost remains zero for the first n clusters assigned to the same cache line.

4. Memory Organization for Array Variables

We solve the memory organization problem for arrays by first constructing the *Interference Graph* among arrays in the code, and then assigning memory addresses to each array by minimizing the possibility of cache conflicts with other arrays in the code. [2]

4.1. Constructing the Interference Graph

In the case of arrays, we note that if two arrays A and B are accessed repeatedly within a loop, then there is a possibility that the accesses to A and B might cause conflict misses in the data cache (Section 2). The Interference Graph (IG) of arrays represents the possibility of cache conflicts between the arrays in the code.

We first create a node for each array in the specification. Next, we determine the arrays that are repeatedly accessed in each loop, and add the loop bound B_l to the edge weights between each pair of arrays. This signifies that a total of B_l cache conflicts could possibly arise between each pair of arrays during execution of this loop. The resulting IG gives us a criterion to prioritize the order in which we assign memory addresses to arrays. The complexity of this procedure is $O(Ln^2)$, where L is the number of loops, and n is the number of arrays in the code.

```
int  a[16], b[16], c[16];
for i = 0 to 7
      a[i] = b[i+3] + 3
for j = 0 to 15
      a[i] = b[i] * c[i]
```

(a)

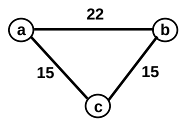

Figure 4. (a) Code showing arrays accessed in loops (b) Interference Graph

In Figure 4(b), we show the Interference Graph derived from the code shown in Figure 4(a). The first loop causes $e(a, b) = 7$. The second loop adds 15 to $e(a, b), e(a, c)$, and $e(b, c)$. The IG helps identify the order in which the memory address assignment to arrays should be done.

4.2. Memory Assignment to Array Variables

In solving the problem of memory assignment of array variables, we assume that the loop bounds and array dimensions are known at compile time. We also assume that a unidimensional array of N elements is stored in N consecutive

[2]The problem of clustering of variables to avoid compulsory misses is not relevant in the case of arrays, as most arrays are usually much larger than a cache line – often much larger than the cache itself.

memory locations, and multidimensional arrays are stored in row-major format. (The issue of selection of a good storage technique for multi-dimensional arrays is addressed in [12]). The memory assignment problem is NP-hard, because the degenerate case, when the array dimension = 1, itself happens to be NP-hard (Section 3.4).

From the Interference Graph, we use the $S(u)$ values for each node u (defined in Section 3.2) to determine the order of assignment of arrays. $S(u)$ signifies the relative importance of the nodes, because a higher $S(u)$ indicates that u could possibly be involved in many cache conflicts.

Central to the technique we use for memory assignment of arrays, is a computation of the *cost* of assigning an array (u) to begin at a specific memory address A. This cost is equal to the *expected number of cache conflicts with all arrays that have already been assigned*, if u were to begin at A. Note that if the first element of u is fixed at address A, all the other elements of u are automatically assigned their respective locations.

To determine whether two specific accesses to two arrays in the same loop will map into the same cache line (i.e, cause cache conflict miss), we perform a symbolic evaluation of the equality checking function. Two memory locations X and Y will map into the same cache line in a direct-mapped cache with k lines (M words per line), if the condition:

$$\left(\left\lfloor \frac{X}{M} \right\rfloor - \left\lfloor \frac{Y}{M} \right\rfloor \right) \bmod k = 0$$

i.e., $(\lfloor X/M \rfloor - \lfloor Y/M \rfloor)$ is an integral multiple of k, which resolves to:

$$(nk - 1) < \frac{X - Y}{M} < (nk + 1) \qquad (1)$$

where n is any integer. Clearly, the symbolically evaluated expression: $(X - Y)/M$, might not always reduce to a constant, because X and Y could be arbitrary functions of any variable in the code. If the expression does not resolve to a constant, then we conclude that the two arrays do not conflict.

To formalize a strategy to perform the memory assignment of arrays, we first describe the cost function *AssignmentCost* that returns the expected number of conflicts when an array is tentatively assigned a specific location.

```
Function AssignmentCost
Input: $u$ – Array under test; $A$ – Proposed start address;
    Access Sequence; Array assignments already completed;
    IG – Interference Graph
Returns: Expected number of cache conflicts for this assignment
    Initialize cost = 0
    for all $v_l | e(v_l, u) \neq 0$, $v_l$ already assigned
    −− all assigned arrays having an edge with $u$ in IG
        for each loop (bound $L$) with accesses to $v_l$ and $u$
            $w$ = no. of times control would alternate between
            elements of $v_l$ and $u$ mapping into same cache line  (i)
            cost = cost + $w \times L$
    return cost
```

In line (i) above, the number of times control alternates between v_l and u is determined by the access sequence, and whether they map to the same cache lines is determined by Condition (1). w represents the number of cache misses in the loop due to conflict between v_l and u. The procedure *AssignArrayAddresses* below outlines the strategy for determining the addresses for each array. $S(u)$ values are used to prioritize the order of assignment, and each array is assigned to start from the first available memory location that generates the lowest *AssignmentCost*, taking the arrays already assigned into consideration.

```
Procedure AssignArrayAddresses
Input: IG – Interference Graph; $k$ – no. of cache lines
Output: Assignment of addresses to all arrays (nodes in IG)
    Address $A$ = 0
    Sort nodes in IG in decreasing order of $S(u)$: $v_0 \ldots v_{n-1}$
    for $i = 0 \ldots n-1$
        Initialize cost $c = \infty$, $min = 0$
        for $j = 0 \ldots k-1$
            if AssignmentCost$(v_i, A+j) < c$ then
                c = AssignmentCost$(v_i, A+j)$; $min = j$
        Assign address $(A + min)$ to first element of $v_i$
        Update $A = A + min +$ size$(v_i)$ for next iteration
```

The worst case complexity of procedure *AssignArrayAddresses* could be $O(nkP)$, where n, k, and P are the number of nodes (arrays), cache lines, and total array accesses in the specification respectively. However, in real behaviors, we have observed that the loop $j = 0 \ldots k-1$ tends to converge very soon (typically less than 2 or 3 iterations), because the number of different array elements that are accessed in inner loops of code is usually small and finite.

This completes the memory address assignment of scalar and array variables in the behavior.

5. Experiments and Results

We now describe the experiments performed on several benchmark examples to validate our memory organization strategy. Our experimental platform was the CW4001 embedded processor core simulator from LSI Logic running a SUN SS-5, using a sample configuration of: 1 KB instruction cache; 256 byte data cache; Line size = 4 words; Array dimension = 16 (for 1-dimensional) and 16 $\times$ 16 (for 2-dimensional); Memory latency = 5 cycles. The latency number is an aggressively low value for the fastest memories. Since the performance difference widens even more for higher memory latencies, the improvements we have shown are the minimum possible.

Column 1 of Table 1 shows the example designs on which we performed our experiments, and Column 2 gives the number of scalar and array variables in each. All the examples are benchmark code kernels used in image processing, telecommunication, and other applications in the DSP

Benchmark	sc/ar	Hit Ratio (%)		#Cycles(x1000)		Red
		Unopt	Opt	Unopt	Opt	(%)
SOR	4/7	17.2	52.2	24.3	20.4	16.0
Laplace	2/2	95.8	95.8	11.6	11.6	0.0
dequant	7/5	38.3	82.4	7.3	6.3	13.7
FFT	20/4	2.4	23.7	66.4	61.5	7.4
idct	20/3	28.3	56.8	23.0	20.2	12.2
leaf_comp	5/3	27.7	76.9	5.4	4.8	11.1
matrix_add	2/3	9.3	75.6	11.5	7.5	34.8
hydro	6/2	21.3	79.7	58.0	38.1	34.3
inner_prod	2/3	7.0	75.2	31.9	21.8	31.7
tri_diag_elim	2/3	2.7	75.0	38.5	23.8	38.2
lin_recur_1	3/2	50.6	65.7	100.5	99.1	1.4
eqn_of_state	5/4	64.0	91.2	93.5	91.5	2.1
ADI_integ	17/6	53.4	61.6	60.6	57.8	4.6
2D_PIC	8/8	60.2	79.7	26.5	20.7	21.9
1D_PIC	4/12	19.6	81.4	89.3	47.6	46.7
implicit_cond	11/7	2.7	74.6	46.4	36.9	20.5
2D_hydro	8/9	17.4	64.1	521.3	267.7	48.6
gen_lin_recur	5/3	2.1	80.6	51.7	40.4	21.9
ord_transport	7/9	8.8	79.4	121.4	83.2	31.5
planckian	4/5	2.7	75.2	48.5	33.6	30.7
2D_impl_hydro	5/6	8.1	64.3	150.6	105.8	29.7
Average		25.7	72.0			21.9

Table 1. Summary of Results

and scientific domain. *SOR* and *Laplace* are algorithms frequently used in DSP applications such as image processing. *Dequant, leaf_comp* and *Idct* are modules from the MPEG decoder application. *FFT* is the Fast Fourier Transform routine, also popular in the DSP domain. *Matrix_add, inner_product* and *tri_diagonal_elim* are frequently used in routines involving multi dimensional arrays treated as matrices. *Hydro, lin_recur_1*, and the rest are other code kernels of typical scientific applications, constituting the Livermore Loops benchmark suite. Columns 3 and 4 show a comparison between the *data cache hit ratios* for the *Unoptimized* (no regard to cache parameters) and *Optimized* (our technique) memory organizations. In almost all the examples, we notice that the difference in the hit ratios is substantial (46 % on an average). Columns 5 and 6 show the execution time in thousands of cycles). There is a significant reduction in the total cycle time for most of the applications (Column 6). The total cycle time reduced by an average of 21.9% over all the examples. This reduction is less than the data cache hit ratio because the instruction cache performance remains unchanged.

6. Conclusions and Future Work

Code generation for embedded processors reveals the scope for many optimizations that have been hitherto unaddressed in traditional compilers. An important feature that can be exploited while generating code for embedded processors is the parameters of the data cache. In this paper, we have demonstrated how a careful data layout strategy that takes into account the parameters of the data cache, such as cache line size and cache size, could induce significant performance improvements in the execution of embedded code.

We described techniques for clustering variables to minimize compulsory cache misses, and for solving the memory assignment problem with the objective of minimizing conflict cache misses. The experiments we performed on standard benchmark code kernels from the DSP and scientific domains, indicate that significant performance improvements result from our memory assignment techniques. We noticed an average improvement of 46% in the data cache hit ratios for the benchmark examples for which we generated code that was executed on the simulator for the CW4001 embedded processor core from LSI Logic.

In the future, we plan to integrate our memory assignment techniques with reordering of the memory accesses in the code. Reordering holds out the possibility of obtaining further improvements in performance through reduction in both compulsory and conflict misses in the data cache.

References

[1] G. Araujo, et. al., “Challenges in Code Generation for Embedded Systems,” in *Code Generation for Embedded Processors,* ed., P. Marwedel and G. Goosens, pp. 48-64, 1995.

[2] K. Au, et. al., “MiniRISC(tm) CW4001 - A Small, Low-Power MIPS CPU Core,” Proc. CICC, 1995.

[3] D. Callahan, K. Kennedy, and A. Porterfield, “Software Prefetching,” Proceedings, ASPLOS, pp 40-52, April 1991.

[4] D. Gannon, et. al., “Strategies for cache and local memory management by global program transformation,” Journal of Parallel and Distributed Computing, 5(5): 587-616, October 1988.

[5] G. Goosens, et. al., “An Efficient Microcode Compiler for Application Specific DSP Processors,” IEEE Transactions on CAD/ICAS, vol 9, No. 9, pp. 925-937, September 1990.

[6] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” ISCA, pp 364-373, May 1990.

[7] N. P. Jouppi, “Cache Write Policies and Performance,” ISCA, pp 191-201, May, 1993.

[8] Y-T. S. Li, et. al., “Performance Estimation of Embedded Software with Instruction Cache Modeling,” ICCAD, November 1995.

[9] S. Liao, et. al., “Storage Assignment to Decrease Code Size,” Proc. PLDI, June 1995.

[10] C. Liem, et. al., “Instruction-Set Matching and Selection for DSP and ASIP Code Generation,” ED&TC, pp. 31-37, March 1994.

[11] P. Marwedel and G. Goosens, “Code Generation for Embedded Processors,” Kluwer Academic Publ., 1995.

[12] P. R. Panda and N. D. Dutt, “Reducing Address Bus Transitions for Low Power Memory Mapping,” ED&TC, pp 63-67, March 1996.

[13] P. Paulin, et. al., “FlexWare: A Flexible Firmware Development Environment for Embedded Systems,” in *Code Generation for Embedded Processors*, ed., P. Marwedel and G. Goosens, 1995.

[14] A. Sudarsanam and S. Malik, “Memory Bank and Register Allocation in software Synthesis for ASIPs,” ICCAD, Nov, 1995.

[15] H. Tomiyama and H. Yasuura, “Optimal Code Placement of Embedded Software for Instruction Caches,” ED&TC, March 1996.

[16] M. J. Wolfe, “A Data Locality Optimizing Algorithm,” ACM SIGPLAN’91 Conf. on PLDI, June, 1991.

[17] Y. Yamada, et. al., “Reducing Cache Misses in Numerical Applications Using Data Relocation and Prefetching,” Technical Report CRHC-95-04, University of Illinois, Urbana, 1995.