Memory Data Organization for Improved Cache Performance in Embedded Processor Applications

PREETI RANJAN PANDA
and
NIKIL D. DUTT
and
ALEXANDRU NICOLAU
University of California at Irvine

Code generation for embedded processors opens up the possibility for several performance optimization techniques that have been ignored by traditional compilers due to compilation time constraints. We present techniques that take into account the parameters of the data caches for organizing scalar and array variables declared in embedded code into memory, with the objective of improving data cache performance. We present techniques for clustering variables to minimize compulsory cache misses, and for solving the memory assignment problem to minimize conflict cache misses. Our experiments with benchmark code kernels from DSP and other domains on the CW4001 embedded processor from LSI Logic indicate significant improvements in data cache performance by the application of our memory organization technique.

Categories and Subject Descriptors: B.3.2 [**Memory Design Styles**]: Cache Memories; D.3.4 [**Processors**]: Compilers

General Terms: Memory Synthesis, System Design, System Synthesis

Additional Key Words and Phrases: Cache Memory, Data Cache

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 [Marwedel and Goosens 1995]. Embedded processors commonly used in the market today can be classified into two categories: (1) application specific processors, such as those in the DSP domain (e.g., TMS320 series

This work was partially supported by grants from ARPA (MDA904-96-C-1472), NSF(CDA-9422095) and ONR(N00014-93-1-1348).
Authors' Address: Dept. of Info. and Computer Science, University of California, Irvine, CA 92697.

from Texas Instruments), and (2) general-purpose processors (e.g. the CW 4000 series from LSI Logic and the ARM series from Advanced RISC Machines). In this paper, we concentrate on the performance improvement of applications running on general-purpose embedded processors. In a general-purpose embedded processor, the architecture more closely resembles traditional processors, with the following well known exceptions: (1) we now frequently have only a single application running on the processor, and (2) we are permitted longer analysis and compilation times for the application. These features raise many interesting problems which are unique to the embedded processor environment, and which have not been addressed by traditional compilers (or have been addressed only partially), largely due to restrictions on compilation times permitted.

Generation of efficient code for embedded processors has been the subject of recent investigation [Goosens et al. 1990; Paulin et al. 1995; Araujo et al. 1995]. Optimization techniques that improve the performance of application programs by exploiting the irregular architectures of some embedded DSP processors and other application specific processors have been reported [Liao et al. 1995; Sudarsanam and Malik 1995; Goosens et al. 1990; Liem et al. 1994]. Research efforts have also focussed on retargetable code generation, with an attempt to generate code from the same behavioral specification, into different target embedded processors, using a suitable processor model [Lanneer et al. 1995; Schenk 1995].

An important determinant of performance in embedded systems is the interaction between the processor and external memory. 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 [Patterson and Hennessy 1994]. Several architectural and compiler optimizations have been reported in the past that ensure spatial and temporal locality of programs so as to improve instruction and data caches .

Cache misses can be classified into three categories: (1) *compulsory misses* – caused when a memory word is accessed for the first time, (2) *capacity misses* – caused when cache data that would be needed in the future is displaced due to the working data set being larger in size than the cache, and (3) *conflict misses* – caused when cache data is replaced by other data, in spite of the presence of usable cache space. Conflict misses can be avoided by using a fully associative cache, but due to access time constraints, all practical cache memories employ a limited-associativity architecture. In fact, studies show that the simple direct-mapped cache architecture is indeed the best choice for most applications [Hill 1988]. For example, recent microprocessors including the HP 735, DEC AXP 3000, SGI Indigo2 and SUN Sparc5 are all equipped with direct-mapped data caches.

In this work, we address the problem of organization of data in main memory to improve data cache performance during the execution of code in embedded processors. Specifically, we concentrate on the minimization of compulsory and conflict misses in data caches, which typically account for roughly 50% of all cache misses for typical programs [Aho et al. 1993].

In Section 2, we discuss the various approaches that have been taken in the past to improve the performance of cache memories. Section 3 introduces the cache parameters and presents the formulation of the data organization problem with a motivating example. Sections 4 and 5 detail the solution to the problem of organizing scalars and arrays respectively, for direct-mapped caches. Section 6 presents an extension of our technique for set-associative caches. In Section 7, we present the experiments we conducted to validate our data organization strategies. We present our conclusions in Section 8.

2. RELATED WORK

In the computer architecture and compiler domains, many techniques dealing with improvement of cache performance involve additional hardware assistance [Jouppi 1990; Yamada et al. 1995], 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 [Carr and Kennedy 1992; Wolf and Lam 1991]. However, this technique can only be applied to the type of code that yields naturally to blocking. Software Prefetching [Callahan et al. 1991], where non-blocking *prefetch* instructions are inserted into the code to load memory data into cache before the actual memory accesses take place, helps avoid cache misses. Similarly, Abraham et al. [1993] propose a modification of memory load instructions to control data placement at various levels of the memory hierarchy. However, these techniques require changes to the processor's instruction set and architecture, and lead to an increase in code size. In hardware-based prefetching [Chen and Baer 1995], the data access patterns are dynamically predicted, and data is fetched from memory ahead of its use. This strategy does not need any special prefetching instructions, but adds to the complexity of the hardware, and also does not address cache conflicts. Allen and Kennedy [1992] have presented program transformations for allocating data to vector registers in vector machines. In [Gannon et al. 1988], the authors have used data dependence analysis to estimate cache size requirements in scientific code. However, they have not addressed cache conflicts in their analysis. In the parallel processing domain, data layout techniques (restricted to the choice between row-major and column-major storage of arrays) to minimize cache coherence overhead have been studied by Ju and Dietz [1991].

Code placement methods based on program traces for improvement of instruction cache performance have been reported in [McFarling 1989; Tomiyama and Yasuura 1996]. Li et al. [1995] present a technique for estimation of instruction cache performance. De Greef et al. [1995] have studied the effect of cache parameters and memory organization strategies on video and imaging applications. Rawat [1993] and Austin [1996] have addressed the problem of variable placement for improving cache performance. However, Rawat addresses only scalar variables, and Austin treats scalar and array variables alike, thereby missing array placement opportunities based on their index expressions. In comparison, we outline a strategy for first placing scalar variables, and present an extension for analysis of array reference indices to determine the placement of arrays.

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, since the generated code is expected to be independent of the target cache and memory configurations. 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).

Further, in traditional compilation, data items are assigned virtual addresses, which are converted into physical addresses of the main memory by the loader software in the operating system, which might have to schedule several processes simultaneously, with the virtual address in each process being mapped to a different physical address. Thus, the compiler has no knowledge of the exact memory location to be occupied by the data, and usually does not address the issue of relative placement of data in memory. In an embedded processor, however, there is typically a single application program executing, and the exact memory location occupied by the data can be assigned by the compiler. This creates opportunities for analyzing the data access patterns to identify possible cache conflict misses. In this work, we exploit this situation to organize data in memory in order to minimize data cache misses.

3. PROBLEM DESCRIPTION

Consider a direct-mapped cache of size C ($C = 2^m$) words, with a cache line size M words, i.e., M consecutive words are fetched from memory on a cache read miss. In our formulation, we assume a *write-through* cache with a *fetch-on-miss* policy [Jouppi 1993], though the technique remains identical for other write policies, and is equally effective.

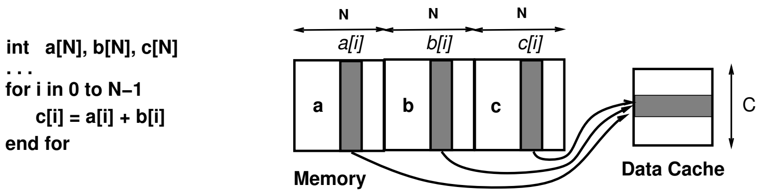

Fig. 1. $a[i], b[i]$ and $c[i]$ map into the same cache line

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 $N \geq C$. Assuming that a single array element occupies one 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 that would contain a word located at memory address A, is given by: $(A \bmod C)/M$. In the above example, array element $a[i]$ would be 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:

$$f(a[i]) = (i \bmod C)/M$$
$$f(b[i]) = ((N + i) \bmod C)/M = (i \bmod C)/M \text{ (since } N \text{ is divisible by } C)$$

$f(c[i]) = ((2N + i) \bmod C)/M = (i \bmod C)/M$
i.e., $a[i], b[i]$, 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 an exact power of two [Yamada et al. 1995]. In such situations, increasing the cache size is not an efficient solution, because the cache misses are not caused due to lack of capacity. Note that the only active cache line during one iteration of the loop is $f(a[i])$. The conflict-misses can be avoided if the cache size C is made greater than N, but there is an associated area and access time penalty incurred when cache size is increased. Reorganizing 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 in the above simple example is to introduce L dummy memory words between two consecutive arrays that are accessed in an identical pattern in the loops. In other words, in the above example, array a begins at 0, array b begins at location: $N + M$ (instead of N) and array c begins at: $2N + 2M$ (instead of $2N$). This will ensure that the array elements are not mapped into the same cache line. We have:

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

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. [1] In Sections 4 and 5, we discuss the strategy we use for organizing scalars and arrays respectively into memory, for reducing data cache conflicts.

4. ORGANIZATION OF SCALAR VARIABLES IN MEMORY

In considering the memory organization, we assume that the scheduling and register allocation of the code has already been performed, and the sequence of accesses to variables is fixed. The steps involved in organizing scalar variables into memory are, in brief:

(1) Build a *Closeness Graph*, representing the degrees of desirability for keeping sets of variables in the same vicinity of main memory. For example, a set of L consecutive words in memory could be read into a cache line on a single cache

[1] A second strategy is to *interleave* the storage of arrays a, b, and c in memory, so that $a[i], b[i]$ and $c[i]$ are in consecutive memory locations. However, in this technique, the storage can be optimized only for one loop, and it is difficult to generalize the interleaving when there are several loops with different access patterns for the same array. We do not consider interleaving further in this work. A *victim-cache* [Jouppi 1990] can also be help avoid the above conflicts. However, it incurs extra hardware cost compared to the alignment approach.

miss.

(2) Group the variables into clusters of L words, where L is the cache line size.

(3) Build a *Cluster Interference Graph*, representing the non-desirability of mapping of clusters into the same cache line (to avoid conflict misses).

(4) Assign memory locations to clusters.

4.1 Constructing the Closeness Graph

We first generate an Access Sequence, which is a graph representing memory references in the code. In the current work, we do not differentiate between memory loads and stores, and represent an *access* to mean either memory *read* or *write*. We create a node in the variable access sequence, corresponding to every memory access in the code, and a directed edge between nodes representing successive accesses. Figure 2(a) shows an example Access Sequence. The label 3 on edge $e \longrightarrow a$ 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. For example, if M words that are accessed successively from memory are placed in consecutive locations, a single memory read could bring them all into cache, thereby reducing upto $M - 1$ extra memory accesses caused due to *compulsory* cache misses.

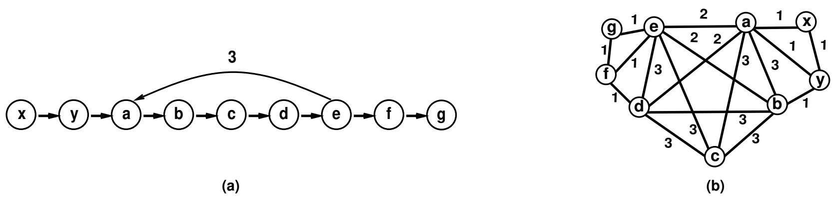

Fig. 2. (a) Access Sequence (b) Closeness Graph for $M = 3$

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 (undirected) 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 (undirected) 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' (i.e., both from u' to v' and from v' to u'). 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.

For a cache line size of M, the construction of the Closeness Graph requires, in the worst case, $O(Mn^2)$ time, where n is the number of nodes in the access sequence.

4.2 Grouping of Variables into Clusters

After constructing the Closeness Graph of the variables, 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 in the Closeness Graph between two variables u and v represents a potential reduction in the number of memory accesses, if the two variables are stored close to each other, so that both can be accessed into the cache with a single memory read. In other words, it identifies an opportunity for avoiding compulsory misses in the data cache. To maximize the sharing of the cache lines by closely correlated variables, we now solve the following problem: *Partition the nodes of the Closeness Graph* CG *(i.e., the set of variables) into clusters of size M, so that the total weight of edges in all the clusters is maximized.*

An exact solution to the above problem has a computational complexity of $O(n^M)$, where n is the number of nodes in the Closeness Graph (i.e., no. of variables involved in memory accesses in the code), since there are $C^n_M (= O(n^M))$ ways of choosing the clusters. In most practical cache implementations, the line size M is usually less than 16. Hence, if the number of scalar variables is large, doing an exhaustive subset search of the Closeness Graph is usually impractical for $M \geq 4$. We employ the greedy heuristic in Figure 3 to perform the grouping of variables into clusters of size M. Essentially, the procedure attempts to identify clusters of M scalar variables whose memory accesses are temporally correlated, so that they can possibly be placed in consecutive memory locations. The heuristic uses the sum of incident edge weights as a criterion.

Procedure *PerformClustering*
Input: $CG(V, E)$ – Closeness Graph; M – Cache Line Size
Output: Set F – Set of clusters of size M
  For each vertex u in V, find the sum of incident edge weights $S(u) = \sum_{v \in V} e(u, v)$
  Let X = vertex set V and $Y = \phi$
  -- *Y keeps track of variables that are already 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 value for T, (ii)
      where $T = \sum_{u \in C, v \in X - C} e(u, v)$ -- *i.e., x is the variable with maximum sum of edge weights with nodes already in C*
      $C = C \cup \{x\}$
      $X = X - \{x\}$
    end while
    Set $e(u, v) = 0 \forall (u \in C)$ or $(v \in C)$ -- *i.e., delete all edges connecting to vertices in the cluster C just formed*
    Update $S(v) \forall v \in X$
  end while
end Procedure

Fig. 3. Procedure for clustering the variables of the Closeness Graph

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: $C_1 : [b, c, d]; C_2 : [a, e, g]; C_3 : [x, y, f]$.

To analyze the computational complexity of procedure *PerformClustering*, we note that the inner **while** loop eliminates one node every time it iterates, so it executes n times. In each iteration, it has to consider the edges from the $O(M)$ nodes in the cluster C, to the $O(n)$ remaining in X, i.e., each iteration takes $O(Mn)$ time. Similarly, the updating of $S(v)$ values requires $O(n)$ time for each vertex deleted. Since a total of n vertices are considered, the total complexity of the procedure is $O(Mn^2)$.

4.3 Building the Cluster Interference Graph

After grouping the variables into clusters of size M, we build a *Cluster Interference Graph (CIG)*, which represents the desirability to store clusters in memory, so that they do NOT map into the same cache line.

Each node in the Cluster Interference Graph represents one cluster of variables obtained in Section 4.2. 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.

In procedure *BuildCIG* (Figure 4), we identify the clusters that should NOT map into the same cache line by assigning high edge weights to the edges between them.

Procedure *BuildCIG*
Input: A – Variable Access Sequence; F – Set of Clusters
Output: CIG – Cluster Interference Graph
    Convert the Variable Access Sequence A into a Cluster Access Sequence
    by renaming each node u in the sequence by the cluster C, where $u \in C$.
    Create a node in CIG for each cluster in F.
    Assign edge weight $e(u, v)$ between nodes u and v = the number of times
    the access to clusters u and v alternate along the execution path.
end Procedure

Fig. 4. Procedure for generating the Cluster Interference Graph

Figure 5(b) shows the Cluster Access Sequence derived from the Variable Access Sequence shown in Figure 5(a), for $M = 2$. [2] In Figure 5(b), the pair of nodes x and y alternate twice in the execution path, due to the edges $x \rightarrow y$ and $y \rightarrow x$. Hence, we have $e(x, y) = 2$ in the Cluster Interference Graph in Figure 5(c). The composition rules to be followed for conditionals and loops are identical to those used for building the Variable Access Sequence (Section 4.1).

For a Variable Access Sequence with n nodes, the process of generating a Cluster Access Sequence requires $O(n)$ time, since each node is renamed in constant time. A Variable Access Sequence with n nodes and a cache line size of M results in $r = \frac{n}{M}$ clusters. In order to obtain the Cluster Interference Graph from the Cluster

[2] Note that this clustering is for illustrative purposes only. If $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$ was the only execution chain in the program, a better clustering is $x : (a, b), y : (c, d), z : (e)$.

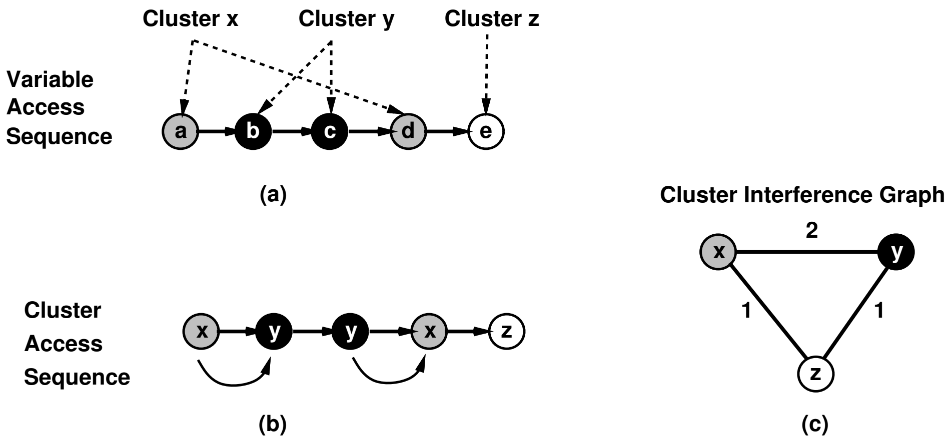

Fig. 5. (a) Variable Access Sequence (b) Cluster Access Sequence (c) Cluster Interference Graph

Access Sequence, we maintain a 2-dimensional $(r \times r)$ array, and update one row of the array (in $O(r)$ time) every time we encounter a node during the traversal of the Cluster Access Sequence. Hence, the total time required is $O(nr)$, i.e, $O(n^2/M)$ time.

4.4 Assignment of Memory Locations to Clusters

The final assignment of variables to memory locations should take into account the clustering and conflict-penalty information in the Cluster 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* as follows:

$$MemAssignCost\ (CIG) = \sum_{x,y \in V(CIG)} e(x,y) \times P(x,y)$$

where $e(x,y)$ is the edge weight, and

$$P(x,y) = \begin{cases} 1 & \text{if memory locations for } x \text{ and } y \text{ map into the same cache line} \\ 0 & \text{otherwise} \end{cases}$$

Figure 6(b) shows a sample memory assignment for a CIG with six clusters (Figure 6(a)), on a cache with four lines. We note that cluster pairs (a,e) and (b,f) map into the same cache lines in a direct-mapped cache, where the cache line address is computed for memory location i, using the equation: $line = i \bmod 4$. The cost of the assignment (Figure 6(c)) is computed to be $1 + 3 = 4$, from the edge weights $e(a,e) = 1$ and $e(b,f) = 3$ for the two conflicting cluster pairs.

In order to minimize conflict misses, we need to solve the following *Cluster Assignment* problem: *Find an assignment of clusters in a CIG to memory locations, such that MemAssignCost (CIG) is minimized.*

The problem of assigning clusters in the Cluster Interference Graph to memory

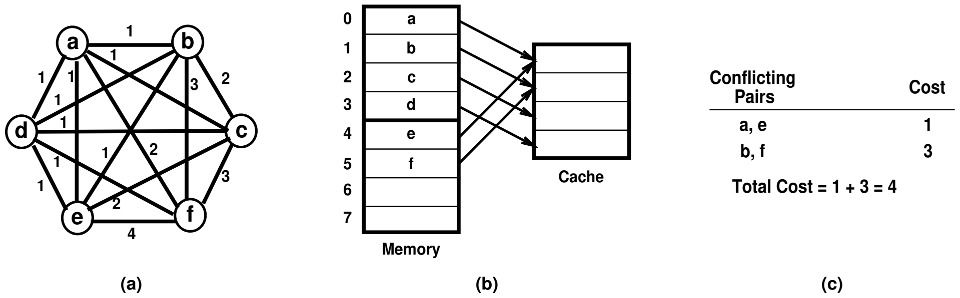

Fig. 6. (a) Cluster Interference Graph (b) Memory Assignment (c) Cost of Assignment

locations can be shown to be NP-hard, by showing a reduction from the Graph Colouring problem. The Graph Colouring problem can be stated as follows: is a given graph $G(V, E)$ k-colourable? That is, does there exist a colouring of vertices that uses $\leq k$ colours, in which no two adjacent vertices of G are assigned the same colour?

THEOREM 1. *The Cluster Assignment Problem is NP-Hard*

PROOF. Consider an instance of the Graph Colouring problem, where we are given a graph $G(V, E)$ where edge weights $e(u, v) = 1$ or 0, representing the presence or absence of an edge (u, v). The problem is to determine whether this graph is k-colourable. This problem is known to be NP-Complete [Garey and Johnson 1979].

Now, if we had an optimal polynomial time algorithm A to solve the Cluster Assignment problem, we could interpret the same graph G as a Cluster Interference Graph – the input to algorithm A, where k is the number of cache lines. Let the cost of the optimal assignment computed by A for this graph be T. We can now arrive at the decision for k-colourability of G as follows:

If $T = 0$, then G is k-colourable, else G is not k-colourable. This follows from the equivalence between *two nodes being assigned the same colouring in* G, and *two clusters being assigned the same cache line in CIG*. Cost $T = 0$ for the memory assignment for CIG indicates an assignment, where no two nodes x and y such that $e(x, y) > 0$ are mapped into the same cache line. This is equivalent to no two adjacent nodes being assigned the same colour in the k-colourability problem for graph G.

In other words, if we had a polynomial-time algorithm to solve the Cluster Assignment problem optimally, we could also solve the Graph Colourability problem in polynomial time. Since the Graph Colourability problem is known to be NP-Complete, we have thus established the NP-hardness of the Cluster Assignment problem (i.e., it is at least as hard as an NP-Complete problem).

□

Procedure *AssignClusters* (Figure 7) is a greedy heuristic to solve the Cluster Assignment problem for a cache of size k that works along the same lines as the *PerformClustering* procedure.

We proceed to make the memory assignments *page* by *page*, where a page consists of k cache lines – the size of the data cache. Note that k consecutive clusters in memory will never conflict in cache, since, for the set of addresses $x = A, A + 1, A + 2, \ldots, A + k - 1$, the cache mapping function $y = x \bmod k$ will always return distinct values. 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: $CIG(V, E)$ – Cluster Interference Graph
Output: Assignment of Clusters to Memory Locations
  Sort the vertices of CIG in descending order of $S(u)$
  $--$ *$S(u)$ is the sum of edge weights incident on vertex u*
  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 $i = 0 \ldots k - 1$
      Delete u from X
    end while
  end while
end Procedure

Fig. 7. Procedure for assigning clusters to memory locations

For the example CIG in Figure 6(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: $f(13), c(9), e(9), b(8), a(6)$ 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 (the first available line with minimum *cost*). 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: $(0 - d; 1 - a)$.

To analyze the complexity of procedure *AssignClusters*, we note that the inner loop eliminates one node in each iteration, so it runs for a total of n iterations (where n is the number of clusters in the graph). Computation of the costs takes $O(n)$ time in this loop. Hence, the total complexity of the algorithm is $O(n^2)$.

At the end of this step, we have an assignment of all the scalar variables in the code to memory locations, that would minimize the compulsory and conflict misses in the data cache when they are accessed during program execution.

5. MEMORY LAYOUT FOR ARRAY VARIABLES

In Section 4, we addressed the problem of memory location assignment for scalar variables, with the objective of minimizing compulsory and conflict misses in the data cache. In this section, we address the problem of organizing array variables into memory, with the intent of minimizing data cache conflict misses when the arrays are accessed at run time.

Since accesses to arrays usually take place inside loops, we focus our attention on the loops while addressing the memory organization issue for arrays. In the current work, we limit the scope to array accesses in the innermost loops in the behavior, since most of the computation tends to occur in these sections of the code [Aho et al. 1993]. Consequently, the impact of cache hit ratios is the greatest here.

We solve the memory layout 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. 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 larger than the size of the cache itself.

5.1 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 3). 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 L to the edge weights between each pair of arrays. This signifies that a total of 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. Procedure *BuildArrayIG* (Figure 8) outlines the construction of the Interference Graph.

Procedure *BuildArrayIG*
Input: Code with array accesses
Output: Interference Graph IG of arrays
    Create a node u for every array u in the code
    Initialize edge weights $e(u,v) = 0$ for all u,v
    for all (innermost) loops l in the code **do**
        Let L be the loop bound of loop l
        Let X = set of all arrays accessed in l
        Update $e(u,v) = e(u,v) + L$ for all $u, v \in X$
    end for
end Procedure

Fig. 8. Procedure for building Interference Graph for arrays

The complexity of procedure *BuildArrayIG* is $O(Ln^2)$, where L is the number of loops, and n is the number of arrays in the code.

In Figure 9(b), we show the Interference Graph derived from the code shown in Figure 9(a). The first loop causes $e(a,b) = 8$. Subsequently, the second loop adds 16 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.

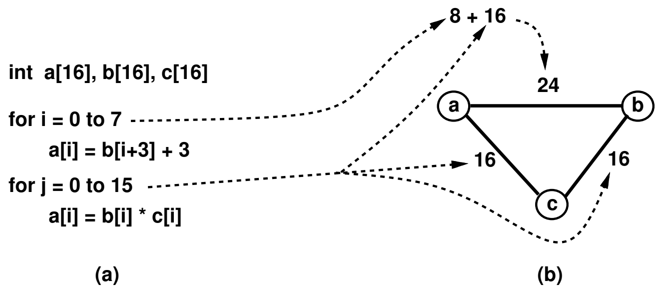

Fig. 9. (a) Code showing arrays accessed in loops (b) Interference Graph: edge weights contributed by loop bounds

5.2 Memory Assignment to Array Variables

In solving the problem of memory assignment of arrayed variables, we make the assumption 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 memory locations, and multidimensional arrays are stored in row-major format [Aho et al. 1993] [3]. The memory assignment problem is NP-hard, because the degenerate case, when the array dimension = 1, itself happens to be NP-hard (Section 4.4).

From the Interference Graph, we use the $S(u)$ values for each node u (the sum of incident edge weights on node u (Section 4.2)) 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, each line being M words wide, 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:

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

where n is any integer.

[3]The issue of selection of a good storage technique for multi-dimensional arrays is addressed in works such as [Ju and Dietz 1991; Cierniak and Li 1995; Panda and Dutt 1996].

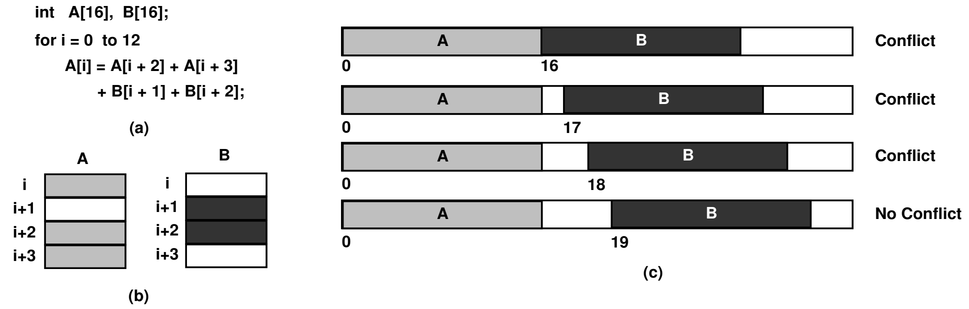

Fig. 10. (a) Sample Code (b) Accessed Array Elements (c) Conflicts during assignment to B

Figure 10(a) shows a sample code, for which the array access pattern (in the i-th iteration) is shown in Figure 10(b). In the general case, for an array W starting at location T, $W[i]$ is located at address: $T + i$. Let us attempt to arrive at a memory assignment for array B, by first assuming that A begins at address 0, and the parameters for the cache are: $M = 4, k = 4$.

We first attempt to place B at address 16 (since A and B are arrays of 16 words each, and A begins at 0). Specifically, let us determine whether $B[i+1]$ and $A[i+3]$ map into the same cache line. $A[i+3]$ is located at address: $(i+3)$ and $B[i+1]$ is located at address: $16+i+1 = 17+i$. To check whether Condition (1) is satisfied, we note that: $(X - Y)/M = ((17+i) - (i+3))/4 = 14/4 = 3.5$, which satisfies Condition 1 for $n = 1$, as we have: $4 - 1 < 3.5 < 4 + 1$. Thus, $A[i+3]$ and $B[i+1]$ would conflict in cache. Similarly, we notice conflicts for assigning addresses 17 and 18 to B. However, address 19 does not entail any cache line conflicts between elements of A and B accessed in the same iteration (Figure 10(c)). Thus, we assign address 19 to B. The three locations 16, 17, and 18, could either be left blank, or occupied by scalar variables.

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.

We now formalize the strategy we presented in the example above into a procedure to perform the memory assignment of arrays. We first describe the cost function *AssignmentCost* (Figure 11) that returns the expected number of conflicts when an array is tentatively assigned a specific location. In practice, the inner **for**-loop is terminated if the assignment cost is zero for any assignment.

In line (i) of *AssignmentCost*, the number of times control alternates between v_l and u is determined by Condition (1). w represents the number of cache misses in the loop due to conflict between v_l and u – this is determined from the Access Sequence. Procedure *AssignArrayAddresses* (Figure 12) outlines the strategy for determining the addresses for each array.

Function *AssignmentCost* requires $O(P)$ time, where P is the total number of distinct array accesses in the code, since, in the worst case, $O(P)$ comparisons might be required. The worst case complexity of procedure *AssignArrayAddresses* could be $O(nkP)$, where n is the number of nodes (arrays), k is the number of cache

```
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) ≠ 0, v_l already assigned        -- i.e., all assigned arrays that have
    an edge with u in IG
        for each loop (bound L) in which accesses to v_l and u occur
            w = no. of times control alternates between elements of v_l and u
            that map into the same cache line, using Condition (1)                    (i)
            cost = cost + w × L      -- w = 0 if there is no conflict
        end for
    end for
    return cost
end Function
```

Fig. 11. Cost function for expected number of conflicts

```
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) (sum of incident edge weights)
    Let the list of nodes be: v_0 ... v_{n-1}
    for i = 0 ... n - 1
        Initialize cost c = ∞
        min = 0          -- keeps track of cache line with minimum mapping cost
        for j = 0 ... k - 1
            if AssignmentCost(v_i, A + j) < c then
                c = AssignmentCost(v_i, A + j)
                min = j
            end if
        end for
        Assign address (A + min) to first element of v_i
        A = A + min+ size(v_i)          -- updating A for next iteration
    end for
end Procedure
```

Fig. 12. Procedure for assigning addresses to arrays

lines. However, in real behaviors, we have observed that the assignment cost in loop $j = 0 \ldots k-1$ tends to converge to zero very soon (typically less than 2 or 3 iterations), because the number of different array elements accessed in inner loops of code is usually small.

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

6. MAPPING INTO SET-ASSOCIATIVE CACHES

Sections 4 and 5 considered the layout of data in memory in order to minimize misses in a direct-mapped cache. We now consider the case of mapping of data into a *set-associative* cache. In an a-way associative cache with total capacity C elements and cache line size L, the number of lines $n = \frac{C}{L}$, and number of sets $s = \frac{n}{a}$. A word residing in block number M in memory would reside in set number: $(M \bmod s)$ in the set-associative cache (as opposed to line number: $(M \bmod n)$ in a direct-mapped cache of the same capacity).

Clearly, the clustering technique (Sections 4.1 and 4.2) to minimize compulsory misses remains identical for the associative cache, since the clustering depends only on the cache line size. However, after the Cluster Interference Graph is obtained (Section 4.3), the memory assignment phase differs from that used in the direct-mapped cache.

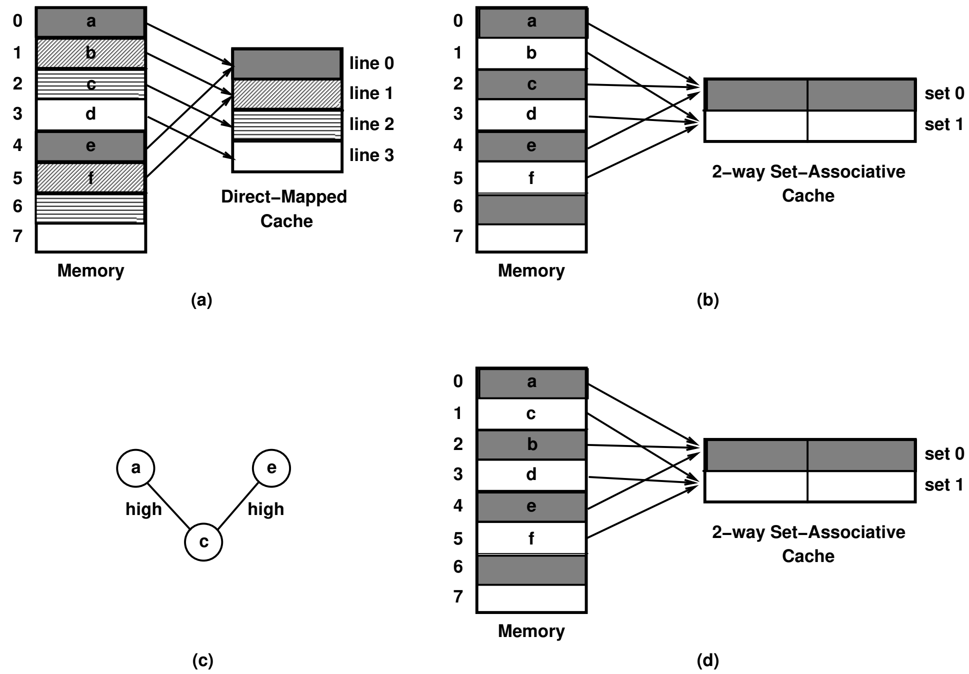

Fig. 13. (a) Mapping into 4-element direct-mapped cache (b) Mapping into 4-element 2-way set-associative cache (c) Unfavourable Interference Graph for 2-way set-associative cache (d) Optimal data placement for 2-way set-associative cache

We show that the optimal memory assignment for direct-mapped caches is not necessarily optimal for set-associative caches, with an example in Figure 13. Figure 13(a) shows the distribution of six elements: a, b, c, d, e, and f located consecutively in memory into locations of a direct-mapped cache with size 4 elements, and cache line size = 1 (i.e., one element per line). We assume that this memory assignment is optimal for the given direct-mapped cache, as determined from Section 4.4. Figure 13(b) shows the distribution of the same memory elements in a 2-way associative cache of size 4 elements. We notice that elements a, c, and e map into the same set in the cache. This mapping could lead to undesirable conflicts, for example, when the interference graph is of the form shown in Figure 13(c). The weights of edges $e(a, c)$ and $e(c, e)$ are high, indicating expensive conflicts if the pairs compete for a cache location[4]. The conflicts are avoided in the direct-mapped cache (Figure 13(a)), because c is mapped into a different line than a and e.

The solution we propose to obtain an optimal memory assignment for a set-associative cache with s-sets, is to consider the page size $= s$ while doing the memory assignment in Section 4.4 (instead of page size $= n$ in a direct-mapped cache). This decision is based on the observation that any s consecutive clusters in memory will always be mapped into distinct sets in the cache, and hence will not conflict. This argument is analogous to that used for direct-mapped caches – any n consecutive clusters in memory will always map into distinct cache lines. In the example shown in Figure 13, we have $n = 2$ and $s = 2$. To obtain the optimal memory assignment for the set-associative cache, we compute the memory assignment with page size = 2. Figure 13(d) shows an example solution. We observe that in this assignment, c is mapped into a different set from a and e, thereby avoiding the expensive conflicts indicated earlier.

In summary, the data layout strategy for an a-way set-associative cache is identical to that for a direct-mapped cache of the same capacity (n), except that in the memory assignment phase, we use a page size of $s = \frac{n}{a}$, instead of n for the direct-mapped cache.

7. EXPERIMENTS AND RESULTS

In this section, we describe the experiments we performed on several benchmark examples to validate our memory data organization strategy. For our experiments, we used the CW4001 simulator kit from LSI Logic running on a SUNSparc5 workstation, that simulates code and reports performance statistics of code generated for execution on LSI Logic's CW4001 embedded processor. The specific performance statistics of interest for our experiments were the *data cache hit ratio* and the *total number of processor cycles* needed for execution of the code on the embedded processor.

We first give a brief description of the benchmark examples we considered, followed by the experimental results on a direct-mapped and a 2-way set-associative data cache of the same capacity. For these experiments, the *read* and *write latencies* (the number of processor cycles required to access an external memory word) were assumed to be 5 processor cycles. These values are generally among the lowest pos-

[4]Figure 13(c) shows only the relevant portion of the interference graph. Other edges (with low edge weights) are not shown.

sible (i.e., for the fastest memories). Since the performance difference widens even more for higher penalties (higher access times), the improvements we have shown are the minimum possible (i.e., most conservative). A Write Buffer [Patterson and Hennessy 1994] of size 4 words was used in the data cache, which was configured to be a *write-through* cache employing *no-fetch-on-write, no-write-allocate* write miss policy, with a cache line of size 4 words. We finally present a study on the variation in performance for different memory latencies.

7.1 Benchmark Examples

All the examples on which we performed experiments are benchmark code kernels used in image processing, telecommunication, and other applications in the DSP and scientific domain. Table (1) presents a profile of these benchmark examples. Column 1 shows the benchmark names; Column 2 gives the number of scalar variables in the code; Column 3 gives the number of array variables; and Column 4 gives the number of array references in the code for each example. For the examples where the array sizes were not specified, we used a size of 64 for one-dimensional arrays, and 16×16 for two-dimensional arrays. *SOR* and *Laplace* [Panda and Dutt 1995] are C-implementations of algorithms frequently used in DSP applications such as image processing. *Dequant, leaf_plus* and *leaf_comp* are modules from the MPEG decoder application. *Idct* is the Inverse Discrete Cosine Transform routine, used in many DSP applications including the MPEG. *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 two dimensional arrays treated as matrices. *Hydro, eqn_of_state, 1D_PIC, implicit_cond, 2D_hydro, ordinates_transport*, and *2D_implicit_hydro* are other code kernels of typical scientific applications, all of which form part of the Livermore Loops set of benchmark suite.

Benchmark	No. of Scalar variables	No. of Array variables	No. of array accesss in code
SOR	4	7	13
Laplace	2	2	10
dequant	7	5	5
FFT	20	4	20
idct	20	3	9
leaf_comp	5	3	3
matrix_add	2	3	3
hydro	6	2	6
inner_prod	2	3	2
tri_diag_elim	2	3	4
eqn_of_state	5	4	10
1D_PIC	4	12	33
implicit_cond	11	7	10
2D_hydro	8	9	44
ordinates_transport	7	9	15
2D_implicit_hydro	5	6	11

Table 1. Profile of Benchmark Examples

7.2 Experiment on Direct-Mapped Cache

Our first experiment was on the CW4001 processor core configured with a 1 KByte instruction cache and 256 Byte direct-mapped data cache with a 16-byte cache line. Figure 14(a) presents a comparison of the data cache hit ratios for the *Unoptimized* and *Optimized* memory data organizations, i.e., for the straightforward organization, and our technique respectively. In all the examples (except *Laplace*), we notice that the difference in the hit ratios is substantial. On an average, we observe a difference of 48 % in the data cache hit ratios.

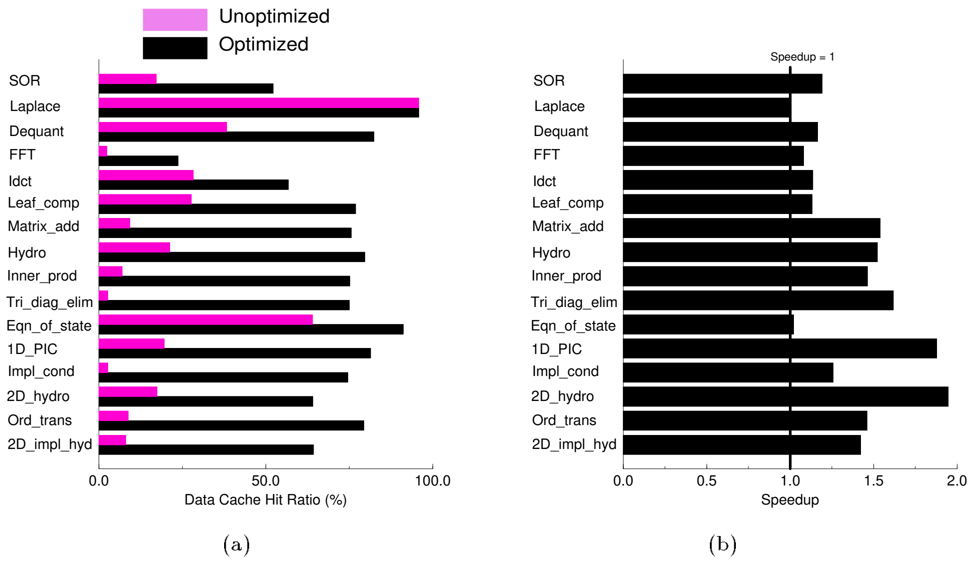

Fig. 14. (a) Comparison of cache hit ratios in direct-mapped cache (b) Speed-up in direct-mapped cache

Figure 14(b) shows the speed-up in execution time on the CW4001 embedded processor resulting from our data layout technique, which is defined as:

$$\text{Speedup} = \frac{\#\ \text{Processor cycles for } \textit{Unoptimized} \text{ case}}{\#\ \text{Processor cycles for } \textit{Optimized} \text{ case}}$$

The *Laplace* example, for which no speed-up is observed, involves only two arrays, *image1* and *image2*, with *image1* being read and *image2* written. Since the cache implements a *write-through, no-fetch-on-write* policy, the two arrays do not conflict in the cache. Hence, the cache performance is good in the unoptimized case and our technique does not lead to any improvement. In all the other examples, cache conflicts are reduced by our data layout technique, leading to reductions in execution time.

There is a significant reduction in the total cycle time for most of the applications. The total cycle time reduced an average of 34% over all the examples. The reason why the percentage difference tends to be lower than the difference data cache hit ratios is that, the total execution time also factors in all the other determinants of program execution speed, most notably, the instruction cache. The instruction cache has an identical performance in both the optimized and unoptimized cases, since in this work, we have concentrated on the performance of the data cache alone. Further, the speed-up is a function of the memory latency; it increases with memory latency (Section 7.4).

Table (2) gives a summary of the overhead incurred by our data organization strategy, in terms of memory space and compilation time. Columns 2 and 3 show the memory space required for storing the variables in the *Unoptimized* and *Optimized* cases respectively. Column 4 shows that in all cases, the wasted memory space due to the data layout is negligible in comparison to the size of the data. On an average, the overhead incurred is 2.17%. Columns 5 and 6 show the compilation times required for the *Unoptimized* and *Optimized* strategies. As we observe in Column 7, the compilation time overhead for performing the data placement analysis was a maximum of 7.4%. On an average, the compilation time overhead was only 4.0%.

	Memory Required (Bytes)			Compilation Time (seconds)		
Benchmark	*Unopt*	*Opt*	Overhead	*Unopt*	*Opt*	Overhead
SOR	7184	7408	3.1%	0.51	0.54	5.1 %
Laplace	2056	2072	0.8%	0.42	0.44	4.1 %
dequant	2332	2364	1.4%	0.40	0.42	4.0 %
FFT	2128	2176	2.3%	0.58	0.59	1.9 %
idct	1616	1632	1.0%	0.50	0.52	3.4 %
leaf_comp	1556	1588	2.1%	0.41	0.43	3.4 %
matrix_add	3080	3112	1.0%	0.38	0.40	4.9 %
hydro	1048	1080	3.1%	0.35	0.36	2.6 %
inner_prod	1036	1052	1.6%	0.39	0.40	3.1 %
tri_diag_elim	1544	1576	2.1%	0.44	0.45	3.2 %
eqn_of_state	2068	2116	2.3%	0.42	0.44	4.2 %
1D_PIC	3628	3724	2.6%	0.51	0.54	5.6 %
implicit_cond	3088	3264	5.7%	0.45	0.47	3.1 %
2D_hydro	8096	8224	1.6%	0.92	0.95	3.6 %
ordinates_transport	4636	4764	2.8%	0.53	0.56	4.1 %
2D_implicit_hydro	5396	5476	1.5%	0.44	0.48	7.4 %

Table 2. Memory and Compilation time Overheads

Most of the performance improvement reported in Figure 14 arises from the placement of arrays. This is because the number of scalar variables present in the examples was relatively low. The miss ratio reductions as a result of our data placement strategy are significantly higher than those observed in [Austin 1996]. This is due to several reasons. Firstly, we have a scheme to analyze array indices and determine the relative placement of arrays in memory. Secondly, most of the benchmark examples we have used are code kernels exhibiting serious cache conflicts in the absence of data alignment. Thirdly, we have used a relatively smaller cache, thereby increasing the possibility of conflicts. The conflicts in the *Unoptimized* case will decrease when larger data caches are used. However, our experiments show that good cache performance can be obtained even for smaller caches by data alignment. Finally, the array sizes in the code kernels of our benchmarks were multiples of the cache size. It is possible that for other randomly chosen array sizes, the *Unoptimized* case would incur fewer conflicts. However, the special case of array sizes being close to a multiple of the cache size occurs frequently – e.g., image processing applications frequently involve images of size 512×512 or 1024×1024, which are very likely to result in multiples of cache sizes, since cache sizes are typically a power of 2. Our technique helps prevent pathological cache conflicts irrespective of array sizes.

7.3 Experiment on 2-way Set-Associative Cache

Our second experiment was performed on the CW4001 processor configured with a 1 KByte instruction cache and 256 Byte, 2-way set-associative data cache with a 16-byte cache line. Figure 15(a) and (b) summarize the data cache hit ratios and execution time speed-up respectively on the new cache configuration.

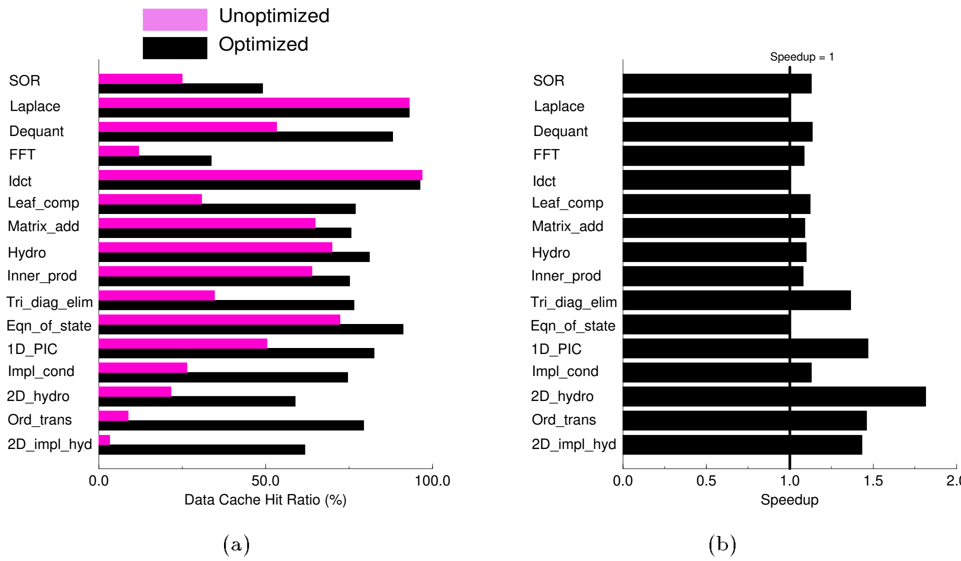

Fig. 15. (a)Comparison of cache hit ratios in 2-way associative cache (b) Speed-up in 2-way associative cache

A comparison of Figure 14(a) and Figure 15(a) shows that the cache hit ratios are generally higher in the set-associative cache, for both *Unoptimized* and *Optimized* cases. However, our data layout technique still results in a significant (average of 29%) improvement in hit ratios for the set-associative cache. Our technique improves upon the straightforward strategy (*Unoptimized*) in all the examples, with the exceptions of *Laplace* (where the performance remains indentical) and *Idct* (where there is a 0.6% degradation). The *Laplace* example is discussed in Section 7.2. In the case of *Idct*, only two arrays are read in the inner loop, which results in the 2-way set-associative cache performing well in the *Unoptimized* case due to the lack of any conflicts.

Figure 15(b) shows an average speed-up in execution time of 21% for our data layout technique over the *Unoptimized* technique. The memory space wasted for the set of examples was also low, and averaged 2.17%.

7.4 Variation of Performance with Memory Latency

In our experiments in Sections 7.2 and 7.3, we assumed a memory latency of 5 processor cycles for fetching a cache line from external memory. This latency is generally among the minimum possible in modern memories. The widening disparity between processor and memory speeds indicates that this value will continue to rise in the future. The execution time increases for higher latencies, resulting in higher speed-ups for our technique.

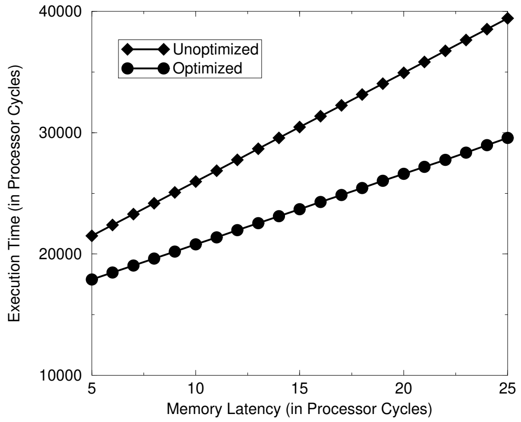

Fig. 16. Variation of execution time with memory latency for SOR example

In Figure 16, we show the variation of execution time with memory latency for the *SOR* benchmark example. The graph shows the variation of execution times for the *Unoptimized* and *Optimized* cases, for memory latency values of all integral multiples of the processor cycles between 5 and 25. We notice that the speed-up increases from 22% for memory latency = 5 cycles, to 33% for memory latency = 25 cycles.

In summary, the execution time speed-up due to our data layout strategy increases with increasing memory latency.

8. 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 report, we have demonstrated how a careful data layout strategy for scalar and array variables that takes into account the parameters of the data cache, such as cache line size and cache size, could effect 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 observed a significant improvements in the data cache hit ratios and execution times for the benchmark examples for which we generated code that was executed on the simulator kit for the CW4001 embedded processor core from LSI Logic.

The strategy we presented for mapping of variables to memory in the case of a set-associative cache is a simple extension of our formulation for direct-mapped caches, and is computationally inexpensive. However, it is possible that a more aggressive strategy incorporating the cache replacement policy (random, least recently used, etc.) can lead to a more efficient data layout. The increased complexity in compilation time can usually be tolerated in embedded systems.

ACKNOWLEDGMENTS

We thank Professor Rolf Ernst for his comments during the ISSS'96 symposium. We also thank the anonymous referees for their valuable suggestions.

References

Abraham, S. G., Sugumar, R. A., Windheiser, D., Rau, B. R., and Gupta, R. 1993. Predictability of load/store instruction latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture* (Nov. 1993).

Aho, A. V., Sethi, R., and Ullman, J. D. 1993. *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Allen, R. and Kennedy, K. 1992. Vector register allocation. *IEEE Transactions on Computers 41*, 10 (Oct.), 1290–1317.

Araujo, G., Devadas, S., Keutzer, K., Liao, S., Malik, S., Sudarsanam, A., Tjiang, S., and Wang, A. 1995. Challenges in code generation for embedded systems. In P. Marwedel and G. Goosens Eds., *Code Generation for Embedded Processors*, pp. 48–64. Kluwer Academic Publishers.

Austin, T. M. 1996. Hardware and software mechanisms for reducing load latency. Ph. D. thesis, University of Wisconsin - Madison.

Callahan, D., Kennedy, K., and Porterfield, A. 1991. Software prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1991), pp. 40–52.

Carr, S. and Kennedy, K. 1992. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing* (Nov. 1992).

Chen, T.-F. and Baer, J.-L. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers 44*, 5 (May), 609–623.

Cierniak, M. and Li, W. 1995. Unifying data and control transofrmations for distributed shared memory machines. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation* (June 1995).

Gannon, D., Jalby, W., and Gallivan, K. 1988. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing 5*, 5 (Oct.), 587–616.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractibility – A Guide to the Theory of NP-Completeness*. W.H. Freeman.

Goosens, G., Rabaey, J., Vandewalle, J., and Man, H. D. 1990. An efficient microcode compiler for application specific dsp processors. *IEEE Transactions on CAD/ICAS 9*, 9 (Sept.), 925–937.

Greef, E. D., Catthoor, F., and Man, H. D. 1995. Memory organization for video algorithms on programmable signal processors. In *IEEE International Conference on Computer*

Design (Oct. 1995), pp. 552–557.

HILL, M. D. 1988. The case for direct-mapped caches. *IEEE Computer 21*, 12 (Dec.), 25–41.

JOUPPI, N. P. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture* (May 1990), pp. 364–373.

JOUPPI, N. P. 1993. Cache write policies and performance. In *International Symposium on Computer Architecture* (May 1993), pp. 191–201.

JU, Y.-J. AND DIETZ, H. 1991. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Fourth International Workshop on Languages and Compilers on Parallel Computing* (Aug. 1991).

LANNEER, D., PRAET, J. V., KIFLI, A., SCHOOFS, K., GEURTS, W., THOEN, F., AND GOOSENS, G. 1995. Chess: Retargetable code generation for embedded dsp processors. In P. MARWEDEL AND G. GOOSENS Eds., *Code Generation for Embedded Processors*, pp. 65–84. Kluwer Academic Publishers.

LI, Y.-T. S., MALIK, S., AND WOLFE, A. 1995. Performance estimation of embedded software with instruction cache modeling. In *IEEE/ACM International Conference on Computer Aided Design* (Nov. 1995), pp. 380–387.

LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. 1995. Storage assignment to decrease code size. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation* (June 1995), pp. 186–195.

LIEM, C., MAY, T., AND PAULIN, P. 1994. Instruction-set matching and selection for dsp and asip code generation. In *Proceedings, European Design and Test Conference* (March 1994), pp. 31–37.

MARWEDEL, P. AND GOOSENS, G. Eds. 1995. *Code Generation for Embedded Processors.* Kluwer Academic Publishers.

MCFARLING, S. 1989. Program optimization for instruction caches. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1989), pp. 183–191.

PANDA, P. R. AND DUTT, N. D. 1995. 1995 high level synthesis design repository. In *International Symposium on System Synthesis* (Sept. 1995).

PANDA, P. R. AND DUTT, N. D. 1996. Reducing address bus transitions for low power memory mapping. In *Proceedings, European Design and Test Conference* (March 1996), pp. 63–67.

PATTERSON, D. A. AND HENNESSY, J. L. 1994. *Computer Organization & Design – The Hardware/Software Interface.* Morgan Kaufman.

PAULIN, P., LIEM, C., MAY, T., AND SUTARWALA, S. 1995. Flexware: A flexible firmware development environment for embedded systems. In P. MARWEDEL AND G. GOOSENS Eds., *Code Generation for Embedded Processors*, pp. 65–84. Kluwer Academic Publishers.

RAWAT, J. 1993. Static analysis of cache performance for real-time programming, Iowa State University.

SCHENK, W. 1995. Retargetable code generation for parallel, pipelined processor structures. In P. MARWEDEL AND G. GOOSENS Eds., *Code Generation for Embedded Processors*, pp. 119–135. Kluwer Academic Publishers.

SUDARSANAM, A. AND MALIK, S. 1995. Memory bank and register allocation in software synthesis for asips. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design* (Nov. 1995), pp. 388–392.

TOMIYAMA, H. AND YASUURA, H. 1996. Optimal code placement of embedded software for instruction caches. In *European Design and Test Conference* (March 1996), pp. 96–101.

WOLF, M. E. AND LAM, M. 1991. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation* (June 1991), pp. 30–44.

YAMADA, Y., JOHNSON, T. L., HAAB, G., GYLLENHAAL, J. C., AND HWU, W. W. 1995. Reducing cache misses in numerical applications using data relocation and prefetching. Technical Report CRHC-95-04, University of Illinois, Urbana.