Fast Memory Bank Assignment for Fixed-point Digital Signal Processors

JEONGHUN CHO
Korea Advanced Institute of Science and Technology
YUNHEUNG PAEK
Seoul National University
and
DAVID WHALLEY
Florida State University

Most vendors of *digital signal processors* (DSPs) support a Harvard architecture, which has two or more memory buses, one for program and one or more for data and allow the processor to access multiple words of data from memory in a single instruction cycle. Also, many existing *fixed-point* DSPs are known to have an irregular architecture with *heterogeneous* registers, which contains multiple register files that are distributed and dedicated to different sets of instructions. Although there have been several studies conducted to efficiently assign data to multi-memory banks, most of them assumed processors with relatively simple, homogeneous general-purpose registers. Thus, several vendor-provided compilers for DSPs that we examined were unable to efficiently assign data to multiple data memory banks; thereby often failing to generate highly optimized code for their machines. As a consequence, programmers for these DSPs often manually assign program variables to memories so as to fully utilize multi-memory banks in their code. This paper reports our recent attempt to address this problem by presenting an algorithm that helps the compiler to efficiently assign data to multi-memory banks. Our algorithm differs from previous work in that it assigns variables to memory banks in separate, *decoupled* code generation phases, instead of a single, tightly-coupled phase. The experimental results have revealed that our decoupled algorithm greatly simplifies our code generation process; thus our compiler runs extremely fast, yet generates target code that is comparable in quality to the code generated by a coupled approach.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation/ compilers/optimization*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*Parallel processors*

General Terms: Algorithms

Additional Key Words and Phrases: Compiler, dependence analysis, DSP, dual memory banks, maximum spanning tree, and non-orthogonal architecture

Corresponding author's address: Yunheung Paek, School of Electrical Engineering, Seoul National University, Seoul 151-744, Korea, ypaek@ee.snu.ac.kr. This research was supported in part by NSF grants CCR-9904943, EIA-0072043, and CCR-0208892.

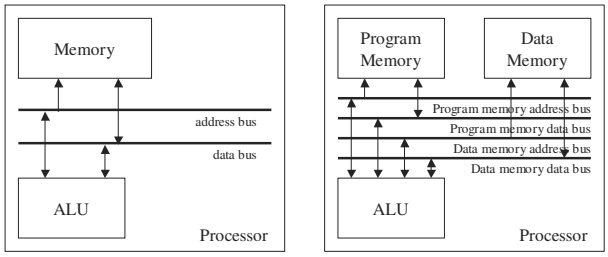
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 1084-4309/20YY/0400-0001 $5.00

1. INTRODUCTION

DSPs are vital to the design of embedded systems today. As performance and cost requirements from the embedded system market become increasingly demanding, DSP architectures are increasing in complexity, which continuously drives state-of-the-art compiler technology to the limit in order to generate the code that meets the desired performance constraints. Often, as indicated by most system designers [Liem 1997], the lack of powerful compilers becomes the greatest stumbling block in the development of embedded systems.

As the speed gap between processor and memory is steadily growing, numerous advanced memory architectures have been proposed to remove the memory bottleneck. As one such effort, system-on-chip DSP architectures support on-chip memory which is internal to the processor for rapid data access, thus filling the speed gap between the processor and off-chip main memory. Although the size of internal memory is quite limited, accessing the off-chip memory causes great performance overhead in terms of time and energy; hence, the code embedded in the system is generally designed to fit into the on-chip memory. Therefore, this paper focuses on utilizing the on-chip memory architecture.

Conventional memory systems use a von Neumann architecture, which has a single memory bank with shared data and address bus shown in Figure 1(a). Efficient utilization of on-chip internal memory is extremely important in embedded applications. In order to better utilize this internal memory, many DSPs employ a Harvard architecture which consists of program and data memory banks shown in Figure 1(b). In this architecture, two memory banks are connected through two independent address and data buses. One obvious advantage of a Harvard architecture, therefore, is that it can simultaneously access one instruction word and one data word in a single instruction cycle.



(a) Von Neumann Memory Architecture (b) Harvard Memory Architecture

Fig. 1. Comparison of memory architectures

To maximize the speed of data memory accesses, the original design of a Harvard architecture has been enhanced by the vendors of DSPs. In one design supported by many DSPs, such as Analog Device ADSP2100, DSP Group PineDSPCore, Motorola DSP56000 and NEC uPD77016, three memory banks are provided: a program memory bank plus two data memory banks each with an independent address space. These three memory banks increase the memory bandwidth by allowing the machine to access the program and the two data memories in parallel. This type of memory architecture has been shown to be effective for many DSP functions such as a FIR filter

$$c(n) = \sum_{i=0}^{N-1} [a(i) \times b(n-i)].$$

In fact, a C implementation of the FIR filter shown in Figure 2(a) can be executed at an

ideal rate of one tap per instruction cycle on a DSP with the three memory banks. However, it can be clearly seen from the code in Figure 2(b) that this ideal speed of execution is only possible with one condition: the two variables (`a[i]` and `b[N-1-1]`) to be fetched for the `mac` instruction should be assigned to different data memory banks. In the example, arrays a and b are assigned respectively to the *dual data memory banks* X and Y, thereby enabling the processor to compute a single filter tap per cycle.

```
int fir_filter(int *a, int *b) {
    int c = 0, i;
    for (i = 0; i < N; i++)
        c += a[i] * b[N-1-i];
    return c; }
```

(a) An example of C code for a FIR filter

```
        rep     #N-1
mac     x0,y0,a x:(r0)+,x0  y:(r4)-,y0
macr    x0,y0,a             (r4)+
movep   a,y:c
```

(b) An example of assembly code of DSP56000

Fig. 2. Implementation of a finite impulse response (FIR) filter

Unfortunately, several existing vendor-provided compilers that we tested were not able to exploit this hardware feature of multiple memory banks efficiently; thereby failing to generate highly optimized code for their target DSPs. This inevitably implies that the programmers for these DSPs should hand-optimize their code in assembly to fully exploit dual data memory banks, which makes programming the processors quite complex and time consuming.

The objective of this work is to describe our implementation of a code generation algorithm that helps the compiler to efficiently assign data to multi-memory banks for DSPs. This paper extends our earlier findings and brief discussion in [Cho and Paek 2002] by presenting more recent progress in exploiting this hardware feature and analyzing in depth the results of a more extensive experiment. This paper introduces two novel techniques respectively based on MST (maximum spanning tree) and graph coloring algorithms, for register and multi-memory bank assignment for DSPs.

In Sections 2 and 3, we start our discussion with an overview of our approach to address the memory bank assignment problem by comparing it with previous compiler approaches related to ours. We provide in Section 4 an overview of the multi-memory bank architecture that we are targeting. We detail in Sections 5 and 6 our memory bank assignment and additive techniques, which includes assignment of variables to memory banks via MST and graph coloring algorithms. We then present in Section 7 the experimental results with a set of DSP benchmarks on a commercial DSP and compare the performance of our compiler with others. Finally we conclude our discussion in Section 8.

2. PREVIOUS WORK

Not until recently had code generation for DSPs or other types of embedded processors received much attention from the main stream of conventional compiler research. One prominent example of a compiler study targeting DSPs may be that of Araujo and Malik [Araujo and Malik 1998] who proposed a linear-time optimal algorithm for instruction selection, register allocation, and instruction scheduling for expression trees. Like most other previous studies for DSPs, their algorithm was not designed specifically for the multi-memory bank DSPs.

One of the early studies that addressed this problem of *memory bank assignment* for DSPs is that of Saghir et al. [Saghir et al. 1996]. In their work, they presented two algorithms: *compaction-based data partitioning* and *partial data duplication*. However, their

algorithms assumed a processor architecture featuring a register file with a set of general-purpose registers, which are commonly found in floating-point DSPs and general-purpose processors (GPPs). Our work differs from theirs because we target *fixed-point* DSPs, such as DSP56000 and ADSP2100, which are widely available in the DSP market today [Eyre and Bier 1998]. One noticeable hardware feature of fixed-point DSPs is that their register architecture is rather *heterogeneous*. That is, their architecture lacks a large number of centralized general-purpose registers; instead, it has multiple small register files where different files are distributed and dedicated to different sets of functional units. An example of such an architecture is displayed in Figure 3. Although some techniques may be commonly applicable to both architectures despite such differences, the heterogeneous register architecture adds more complexity to the original problem of memory bank assignment with homogeneous registers, which drives the compiler to take a different approach, as indicated in [Sudarsanam and Malik 2000]. In fact, in our earlier work [Jung and Paek 2001], we also reached a similar conclusion that conventional code generation techniques originally developed for GPPs often become obsolete for a DSP.

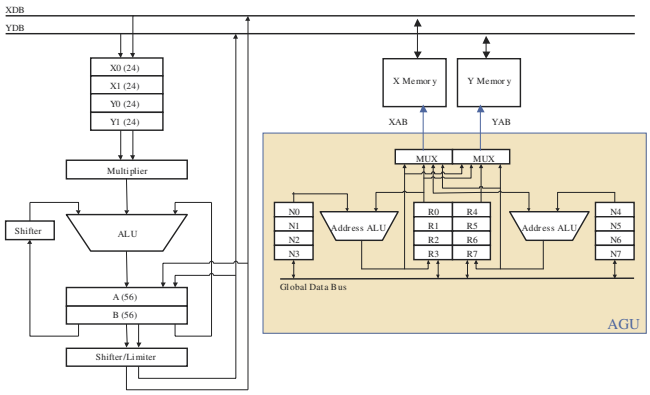


Fig. 3. Motorola DSP56000 data path with dual data memory banks X and Y

By the same token, our approach differs from the *RAW* project at MIT [Barua et al. 2001] since their memory bank assignment techniques neither assume heterogeneous registers nor even fixed-point DSP architectures. However, some of their *memory disambiguation* techniques, called *equivalence-class unification* and *modulo unrolling*, may be applicable in part to memory bank assignment for DSPs.

Panda's work [Panda 1999] also differs from ours because he handled the bank assignment problem in the area of high-level synthesis, not of compiler. In his work, the application-specific customization of memory banks for arrays was addressed during behavioral synthesis by using the k-way min-cut graph partitioning algorithm. Because he focused on minimization of memory access delay and area from the standpoint of behavioral synthesis, he did not consider conflict constraints on heterogeneous registers and memory banks as we do for a given fixed-point DSP at compile time.

In [Powell et al. 1992], Powell et. al, used a simple greedy method to handle multiple memory banks. In their approach, they synthesized optimized assembly code from signal flow block diagrams directly. Using a library, block diagrams are instantiated to a meta-assembly language with symbolic register and memory references. After a simple optimization of the meta-assembly code, scheduling and register allocation are performed.

The assignment of program variables to the X/Y banks is performed with an alternating approach, according to their access sequence. The efficiency of this simple approach strongly depends on the hand-coded library of the meta-assembly blocks, and the alternate partitioning cannot expect a good quality code for complex programs.

Most recently, this bank assignment problem was addressed by Leupers et. al,[Leupers and Kotte 2001]. In their approach, a *data dependency graph* (DDG) is first constructed for each assembly code function. For a given DDG, the *interference graph* is constructed in such a way that potential parallelism is reflected by graph edges. After a step for reducing the interference graph size, they applied an *Integer Linear Programming (ILP)* to partitioning for the interference graph. The ILP approach works well for small benchmark programs. However, the ILP partitioning technique would take an enormous amount of compilation time for a very large program; thereby requiring further study for more heuristics.

The *SPAM* project is closely related to our work, which was conducted by researchers at Princeton and MIT [Araujo and Malik 1998; Sudarsanam and Malik 2000]. In their work, they presented experimental results showing that the SPAM compiler can generate highly optimized code for a commercial DSP in most cases. However, the results also showed evidence that the compilation time may increase substantially for large applications just in the case of [Leupers and Kotte 2001], and that the compiler may not reach an optimal solution even after a long exhaustive search. Since compilation speed is relatively of less importance for compilers targeting embedded systems, even this possible long compilation time may be still tolerable for some compilers which put more emphasis on the code quality than compilation time. However, if this is excessively too long, it could be a flaw for some other industry compilers due to their ever increasing demands on faster time-to-market embedded software design and implementation.

3. MOTIVATION OF OUR APPROACH

In our quest for a more practical algorithm, we have found that the long compilation time in these previous studies results from their attempt to deal with memory bank assignment in a single, combined step, where several code generation phases are coupled and simultaneously considered to address the issue. In this sense, we deem that they took a *coupled approach* for memory bank assignment. In their approach, *reference allocation* (register allocation plus memory bank assignment) occurs *simultaneously*; that is, temporaries are allocated to physical registers at the same time they are assigned the memory banks. To solve their simultaneous reference allocation problem, they build a *constraint graph* that represents multiple constraints under which an optimal solution to their problem is sought. Unfortunately, these multiple constraints in the graph turn their problem into a typical multivariate optimum problem which is an NP-complete problem. In this coupled approach, multivariate constraints are unavoidable as various constraints on many heterogeneous registers and multi-memory banks should be all involved to find an optimal reference allocation simultaneously. As a consequence, to avoid using such an expensive algorithm, they inevitably resorted to a heuristic algorithm, called *simulated annealing*, based on a Monte Carlo approach [Gould and Tobochnik 1988]. However, their paper reported [Sudarsanam and Malik 2000] that even with this heuristic, their compiler still had to take more than 1000 seconds even for a moderately sized program. This is mainly because their constraint graph with so many constraints became rapidly large and complicated as the code size

grew.

We see that the slowdown in compilation is obviously caused by the intrinsic complexity of their coupled approach. Thus in our work, we eased the problem by choosing a more relaxed approach where we carefully *decoupled* the code generation process into several small phases. The phases were sequentialized, and local optimizations were applied along with the sequence to form a final *near-optimal* bank assignment. More specifically, in our approach, register allocation and assignment are decoupled from code compaction and memory bank assignment; thereby, the binding of physical registers to temporaries comes only after code has been compacted and variables assigned to memory banks.

Of course, one would expect a degradation of our output code quality due to the limitations newly introduced by considering physical register binding separately from memory bank assignment. In fact, conventional wisdom holds that coupled approaches always do better than decoupled ones because all factors that may affect the process of finding an optimal solution can be simultaneously considered before any decision is made. This is quite true in principle. However, in practice, a coupled approach may increase the cost of computation significantly while only achieving a slight improvement in the solution.

The actual motivation of this research is to explore the benefits and detriments of the decoupled approach, as opposed to the coupled one; that is, through this work, we tried to find how fast the compilation time can be in real cases by trading-off the code quality. In their papers [Sudarsanam 1998; Sudarsanam and Malik 2000], Sudarsanam, et. al, in fact, discussed possible drawbacks of a decoupled approach, as compared to their coupled one. However, we believe that *careful decoupling* may alleviate such drawbacks in practice while maximizing the advantages in terms of compilation speed, which is often a critical factor for industry compilers. To verify this, we implemented a memory bank assignment algorithm and compared the performance of our approach with the coupled one in an experiment with the same benchmarks on the same commercial DSP. As will be reported later in this paper, the results were quite encouraging. First of all, we found that the code generation time was dramatically reduced by a factor of up to four orders of magnitude. This result was somewhat already expected because our decoupled code generation phases greatly simplified the bank assignment problem overall. Meanwhile, the benchmarking results also showed almost in every case that we generated code that is nearly identical in quality to the code generated by the coupled approach.

4. OPERATIONS WITH MULTIPLE DATA MEMORY BANKS

In this section, we discuss how operations are performed on DSPs with multiple data memory banks, and how memory bank assignment affects the performance of these operations. As an example of multi-memory bank DSPs, we will use Motorola DSP56000 whose data path was shown in Figure 3.

In the DSP56000, ALU operations are divided into data operations and address operations. Data ALU operations are performed on a data ALU (see Figure 3) with *data registers* which consist of four 24-bit input registers (X0, X1, Y0 and Y1) and two 56-bit accumulators (A and B). Address ALU operations are performed in the *address generation unit* (AGU), which calculates memory addresses necessary to indirectly address data operands in memory. Since the AGU operates independently from the data ALU, address calculation can occur simultaneously with data ALU operations.

As shown in Figure 3, the AGU is divided into two identical halves, each of which has

an address ALU and two sets of 16-bit register files. One set of the register files consists of four *address registers* (R0, R1, R2 and R3) and four *offset registers* (N0, N1, N2 and N3), and the other consists of four address registers (R4, R5, R6 and R7) and four offset registers (N4, N5, N6 and N7). The two address ALUs are identical in that each contains a 16-bit full adder, called an *offset adder*, which either performs auto-increment/decrement or add/subtract the contents of the offset register Ni in its set to/from the contents of the selected address register Ri.

The address output multiplexers select the source for the XAB, YAB. The source of each effective address may be the output of the address ALU for indexed addressing or an address register for register-indirect addressing. At every cycle, the addresses generated by the ALUs can be used to access two words in X and Y memory banks in parallel, each of which consists of 512-word $\times$ 24-bit memory.

Possible memory reference modes of the DSP56000 are of four types: X, Y, L and XY. In X and Y memory reference modes, the operand is a single word either from the X or Y memory bank. In L memory reference mode, the operand is a long word (two words each from X and Y memories) referenced by one operand address. In XY memory reference mode, two independent addresses are used to move two word operands to memory simultaneously: one word operand is in X memory, and the other word operand is in Y memory. Such independent moves of data in the same cycle are called a *parallel move*. In Figure 3, we can see two data buses XDB and YDB that connect the data path of DSP56000 to two data memory banks X and Y, respectively. Through these buses, a parallel move is made between memories and data registers.

These architectural features of the DSP56000, like most other DSPs with multi-memory banks, allow a single instruction to perform one data ALU operation and two move operations in parallel per cycle, but only under certain conditions due to hardware constraints. In the case of the DSP56000, the following four requirements should be met to maximize the utilization of the dual memory bank architecture: (1) two memory accesses or a pair of one memory access and one register transfer can be performed in parallel, (2) destination registers have to be different, (3) two effective addresses should reference different memory banks, and (4) the X data memory access is performed with X0, X1, A, or B, and the Y data memory access is performed with Y0, Y1, A, or B.

In the following sections, we will discuss how we generate code for dual memory banks by making the parallel move conditions meet in the code so that as many parallel moves as possible can be generated in it.

5. MEMORY BANK ASSIGNMENT

Figure 4 shows the overall structure of our compiler, called *Soargen*. Our code generation process is divided into six phases: instruction selection, register class allocation, code compaction, memory bank assignment, register assignment, and memory offset assignment.

In this paper, we will focus only on memory bank assignment since all other phases are explained in our earlier literature[Cho and Paek 2002]. Figure 5 shows an example code generated after the code compaction phase.

After code compaction, each variable in the resulting code is assigned to one of multiple memory banks (in this example, two banks X or Y of the DSP56000). The first step of this phase is to construct a weighted undirected graph, which we called the *simultaneous reference graph* (SRG). The graph contains variables referenced in the code as nodes. An

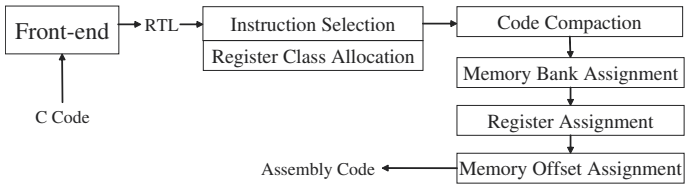


Fig. 4. Overall structure of the Soargen compiler

```
MOVE            a,r0  b,r1
MOVE            c,r2  d,r3
MAC  r0, r1, r2  e,r4  f,r5
MAC  r3, r4, r5  low(r2),low(v)
MOVE            high(r2),high(v)  low(r5),low(w)
MOVE            high(r5),high(w)
```

Fig. 5. Code sequence after compacting the code

edge e = (v_j,v_k) in the SRG means that both variables v_j and v_k are referenced within the same instruction word in the compacted code. Figure 6(a) shows an SRG for the code from Figure 5. The weight on an edge between two variables represents the number of times the variables are referenced within the same word.

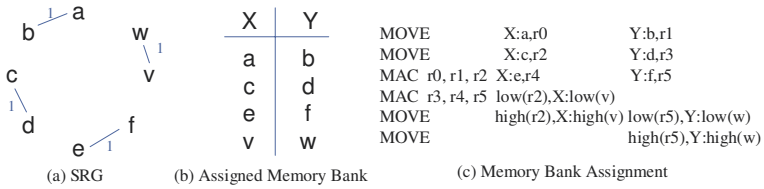


Fig. 6. Code result after memory bank assignment determined from its simultaneous reference graph built for the code in Figure 5

According to the parallel move conditions, two variables referenced in an instruction word must be assigned to different memory banks in order to fetch them in a single instruction cycle. Otherwise, an extra cycle would be needed to access them. Therefore, the strategy that we take to maximize the memory throughput is to assign a pair of variables referenced in the same word to different memory banks whenever it is possible. If a conflict occurs between two pairs of variables, the variables in one pair that appear more frequently in the same words shall have a higher priority over those in the other pair. Notice here that the frequency is denoted by the weight in the SRG.

Figure 6(b) shows that the variables a, c, e, and v are assigned to X memory, and the remaining ones b, d, f, and w are to Y memory. This is optimal because all pairs of variables connected via edges are assigned to different memories X and Y, thus avoiding extra cycles to fetch variables, as can be seen from the resulting code in Figure 6(c). In the case of variables v and w, we still need two cycles to move each of them because they are long type variables with double-word length. However, they also benefit from the optimal memory assignment as each half of the variables is moved together in the same cycle.

The memory bank assignment problem that we face in reality is not always as simple as the one in Figure 6. To illustrate a more realistic and complex case of the problem, consider Figure 7 where the SRG has five variables.

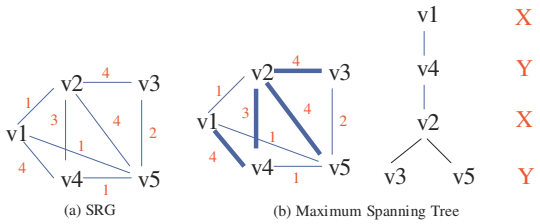


Fig. 7. More complex example of a simultaneous reference graph and the maximum spanning tree constructed from it

We view the process of assigning n memory banks as that of dividing the SRG into n disjoint subgraphs; that is, all nodes in the same subgraph are assigned a memory bank that corresponds to the subgraph. In our compiler, therefore, we try to obtain an optimal memory bank assignment for a given SRG by finding a *partition* of the graph with the minimum cost according to Definition 1.

DEFINITION 1. *Let $G = (V, E)$ be a connected, weighted graph where V is a set of nodes and E is a set of edges. Let w_e be the weight on an edge $e \in E$. Suppose that a* **partition** *$P = <G_1, G_2, \cdots, G_n>$ of the graph G divides G into n disjoint subgraphs $G_i = (V_i, E_i)$, $1 \leq i \leq n$, such that $(v_j, v_k) \in E_i$ if $(v_j, v_k) \in E$ for $v_j \in V_i$ and $v_k \in V_i$. Then, the* **cost** *of the partition P is defined as*

$$\sum_{i=1}^{n} \sum_{e \in E_i} w_e.$$

Finding such an optimal partition with the minimum cost is another NP-complete problem. So, we developed a greedy approximation algorithm with $O(|E| + |V|lg|V|)$ time complexity, as shown in Figure 8. Since in practice $|E| \approx |V|$ for our problem, the algorithm usually runs fast in $O(|V|lg|V|)$ time, as demonstrated in Section 7. In the algorithm, we assume $n = 2$ because virtually no existing DSPs have more than two data memory banks. But, this algorithm can be easily extended to handle the cases for $n > 2$. First, we describe the algorithm for dual memory banks in the next subsection, and then we generalize the algorithm for n memory banks in the following subsection.

5.1 When $n = 2$

In our memory bank assignment algorithm, we first identify a *maximum spanning tree* (MST) of the SRG. Given a connected graph G, a *spanning tree* of G is a connected acyclic subgraph that covers all nodes of G. A MST is a spanning tree whose total weight of all its edges is not less than those of any other spanning trees of G. One interesting property of a spanning tree is that it is a bipartite graph as any tree is actually bipartite [Cormen et al. 1990]. So, given a spanning tree T for a graph G, we can obtain a partition $P = <G_1, G_2>$ from T by, starting from an arbitrary node, say u, in T, assigning to G_1 all nodes an even distance from u and to G_2 those an odd distance from u.

Based on this observation, our algorithm is designed to first identify a spanning tree from the SRG, and then, to compute a partition from it. But we here use a heuristic that chooses not an ordinary spanning tree but a maximum spanning tree. The rationale for the heuritic is that, if we build a partition from a MST, we can eliminate heavy-weighted edges of the MST, thereby increasing the chance to reduce the overall cost of the resultant partition. Unfortunately, constructing a partition from a MST does not guarantee the

```
Input:  a simultaneous reference graph G_SR = (V_SR, E_SR)
        two memory banks M_X and M_Y
Output: a set V_SR whose nodes are all colored either with M_X or M_Y
        a set E_UN whose edges are unselected in MST
Algorithm: DMBA
  S_T ← Q ← ∅;                          // S_T is a set of MSTs and Q is a priority queue
  for all nodes v in V_SR do unmark v;
  u ← select_unmarked_node_in(V_SR);  // Return ⊥ if every node in V_SR is marked
  i ← 1;  create a new MST T_i;
  while u ≠ ⊥ do  // Find all MSTs for connected subgraphs of G_SR
    mark u;
    E_u ← the set of all edges incident on u;
    sort the elements of E_u in increasing order by weights, and add them to Q;
    while Q ≠ ∅ do
      remove an edge e = (w, z) with highest priority from Q;
      if z is unmarked then T_i ← T_i ∪ {e};  u ← z;  break;
      if w is unmarked then T_i ← T_i ∪ {e};  u ← w;  break;
    od
    if u is marked then       // All nodes in a connected subgraph of G_SR have been visited
      u ← select_unmarked_node_in(V_SR); // Select a node in another subgraph, if any, of G_SR
      add T_i to S_T;  i++;  create a new MST T_i;
    fi
  od
  E_UN = Q;       // For extended memory bank assignment
  for all nodes v in V_SR do uncolor v;
  for every MST T_i ∈ S_T do  // Assign variables in T_i's to memory banks M_X and M_Y
    next_visitors_Q ← ∅;
    m ← # of nodes in V_SR of M_X-color − # of nodes in V_SR of M_Y-color;
    select an arbitrary node v in T_i;
    if m > 0 then             // More nodes have been M_X-colored
      color v with M_Y-color;
    else                      // More nodes have been M_Y-colored
      color v with M_X-color;
    repeat
      for every node u adjacent to v do
        if u is not colored then
          color u with a color different from the color of v;
          append u to next_visitors_Q;
        fi
      v ← extract one node from next_visitors_Q;
    until all nodes in T_i are colored;
  od
  m ← # of nodes in V_SR of M_X-color − # of nodes in V_SR of M_Y-color;
  while m > 0 do              // While there are more M_X-colored nodes than M_Y-colored ones
    if ∃ uncolored node v ∈ V_SR then
      color v with M_Y-color;  m--;
    else break;
  while m < 0 do              // While there are more M_Y-colored nodes than M_Y-colored ones
    if ∃ uncolored node v ∈ V_SR then
      color v with M_X-color;  m++;
    else break;
  if m = 0 then
    for any uncolored node v in V_SR do
      color v alternately with M_X and M_Y colors;
  return V_SR and E_UN;
```

Fig. 8. A memory bank assignment algorithm for dual memories M_X and M_Y

optimum solution. However, according to our empirical study [Cho and Paek 2002], the notion of a MST provides us an important idea about how to find a partition with low cost, which is in turn necessary to find a near-optimal memory bank assignment. For instance, our algorithm can find an optimal partitioning for the SRG in Figure 7.

To find a MST, our algorithm uses Prim's MST algorithm [Prim 1957]. Our algorithm is global; that is, it is applied across basic blocks. For each node, the following sequence is repeatedly iterated until all SRG nodes have been marked. In the algorithm, the edges in the *priority queue* Q are sorted in the order of their weights, and an edge with the highest weight is removed first. When there is more than one edge with the same highest weight, the one that was inserted first will be removed. Note here that the simultaneous reference graph G_{SR} is not necessarily connected, as opposed to our assumption made above. Therefore, we create a set of MSTs one for each connected subgraph of G_{SR}. Also, note in the algorithm that at least one of the nodes w and z should always be marked because the edges of a marked node u was always inserted in Q earlier in the algorithm. Figure 7(b) shows the spanned tree obtained after this algorithm is applied to the SRG given in Figure 7(a). We can see that X memory is assigned in even depth and Y memory in odd depth in this tree.

5.2 When $n > 2$

Virtually no existing DSPs currently support more than two memory banks. However, in the next generation DSPs, it is likely that the number of memory operations will go up and this will be supported by having memories with more memory banks. Therefore, to handle n memory banks, we extend our DMBA (dual memory bank assignment) algorithm shown in Figure 8. After dual memory bank assignment is performed, all nodes of each spanning tree of unselected edges which can cause extra costs are assigned to the same memory bank. For example, Figure 9 (a) shows v_1 and v_2 (in a spanning tree $v_1 \overset{1}{-} v_2$) are currently assigned to memory bank $M1$ with cost 1, and v_3, v_4, and v_5 ($v_4 \overset{1}{-} v_5 \overset{2}{-} v_3$) are assigned in $M2$ with cost 3. The edge (v_1, v_5) is not included in the MST since it does not add any extra cost because v_1 and v_5 are assigned to different memory banks. When one more memory bank is added to this resulting SRG, we choose the one with higher cost, which is $v_4 \overset{1}{-} v_5 \overset{2}{-} v_3$ in this example, and again apply the DMBA algorithm to further partition the graph. The variables v_3, v_4, and v_5 in this new SRG with cost 3 are assigned to either memory bank $M2$ or $M3$. As a result, the SRG and memory banks shown in Figure 9 (b) are generated. Through one more iteration, all variables are assigned to four memory banks and parallel moves can be used without extra cost.

Figure 10 shows our algorithm extended from the DMBA algorithm to handle n memory banks. As mentioned previously, the DMBA does not always result in the optimum solution. So, this extended algorithm based on greedy heuristics may not also find an optimal bank assignment for some cases. However, our experience revealed that this extended algorithm results in optimal or near-optimal partitionings for n memory banks in many cases, as in the case of the example in Figure 9.

6. NAME SPLITTING AND MERGING

As discussed in [Cho and Paek 2002], our compiler, like most optimizing compilers, uses graph coloring to improve register assignment. The central idea of graph coloring is to

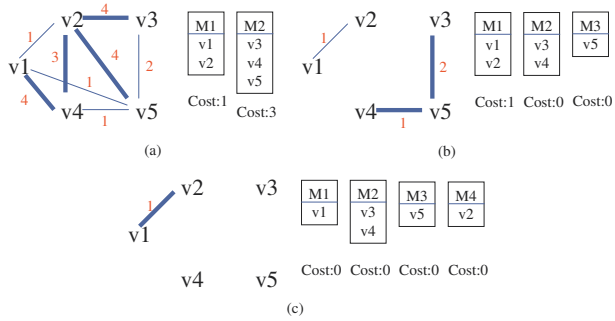


Fig. 9. Extended memory bank assignment for n =2, 3, and 4

Input: a simultaneous reference graph $G_{SR} = (V_{SR}, E_{SR})$
the number of memory banks n
Output: a set V_{SR} whose nodes are all colored with one of $M = \{M_1, M_2, ..., M_n\}$
Algorithm: EMBA
$B \leftarrow \emptyset$; *// B is a set of assigned banks*
$i \leftarrow M_1$; $j \leftarrow M_2$;
(V_{SR}, E_{UN}) = **DMBA**(V_{SR}, E_{SR},i,j);*// Memory bank assignment with M_1 and M_2*
$B \leftarrow B \cup \{M_1\} \cup \{M_2\}$;
for every memory bank m in $\{M_3, M_4, ..., M_n\}$ **do**
  if $E_{UN} = \emptyset$ **then** break;
  $k = select_maximum_cost_in(B)$;
  for every edge (v, w) in E_{UN} **do**
    if v and w are colored with k **then**
      $E_{tmp} \leftarrow E_{tmp} \cup (v, w)$; $E_{UN} \leftarrow E_{UN} - (v, w)$;
  od
  (V_{SR}, E_{tmp}) = **DMBA**(V_{SR}, E_{tmp},k,m);
  $E_{UN} \leftarrow E_{UN} \cup E_{tmp}$; $B \leftarrow B \cup \{m\}$;
od
return V_{SR};

Fig. 10. A memory bank assignment algorithm extended for n memory banks

partition each variable into separate live ranges, where each live range is a candidate to be allocated to a register rather than entire variables. We have found that the same idea can be also used to improve the original memory bank assignment algorithm described in Section 5.

Similar to conventional graph coloring approaches, we build an undirected graph, called the *memory bank interference graph*, to determine which live ranges conflict and could not be assigned to the same memory bank. Disjoint live ranges of the same variable can be assigned to different memory banks after giving a new name to each live range. This additional flexibility of a graph coloring approach can sometimes result in more efficient allocation of variables to memory banks, as we will show in this section.

Two techniques, called *name splitting* and *merging*, have been newly implemented to help the memory bank assignment benefit from this graph coloring approach by relaxing the name-related constraints on variables that are to be assigned to memory banks. Figure 11 presents an example of memory bank assignment where variables are assigned to memory banks using the basic techniques discussed in Section 5.

Figure 11(a) shows an example of code that is generated after code compaction, and Figure 11(b) depicts the live ranges of each of the variables. Note that the variables a and d each have multiple live ranges. Figures 11(c) and 11(d) show the SRG and the

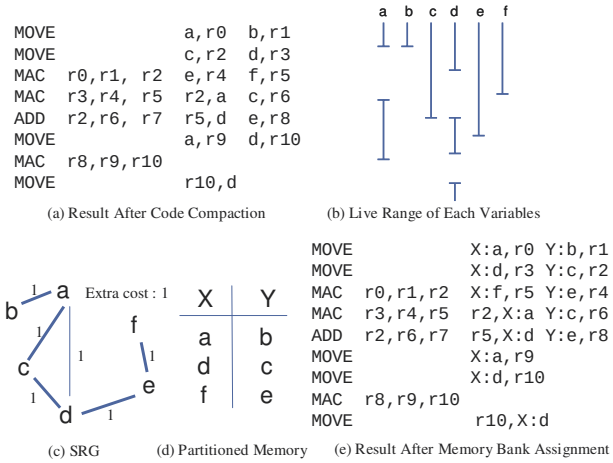


Fig. 11. Code example and data structures to illustrate name splitting and merging

assignment of variables to memory banks. We can see that a single parallel move cannot be exploited in the example because a and d were assigned to the same data memory. Finally, Figure 11(e) shows the resulting code after memory banks are assigned using the memory partitioning information from Figure 11(d).

Figure 12 shows how name splitting can improve the same example in Figure 11. Name splitting is a technique that tries to reduce the code size by compacting more memory references into parallel move instructions. This technique is based on a well-known graph coloring approach. Therefore, instead of presenting the whole algorithm, we will describe the technique with an example given in Figure 12.

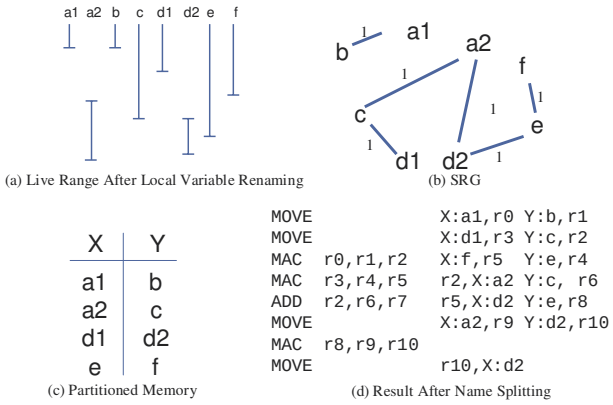


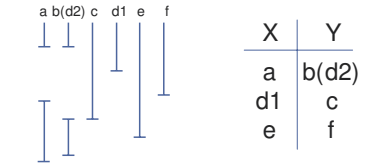
Fig. 12. Name splitting for local variables

We can see in Figure 12(a) that each live range of the variable is a candidate for being assigned to a memory bank. In the example, the two variables a and d with disjoint live ranges are *split*; that is, each live range of the variables are given different names.

Figures 12(b) and 12(c) show the modified SRG and the improved assignment of variables to memory banks. Figure 12(d) demonstrates that, by considering live ranges as opposed to entire variables for bank assignment, we can place the two live ranges of d in

different memory banks, which allows us to exploit a parallel move after eliminating one MOVE instruction from the code in Figure 11(e).

Although name splitting helps us to further reduce the code size, it may increase the data space, as we monitored in Figure 12. To mitigate this problem, we *merge* names after name splitting. Figure 13 shows how the data space for the same example can be improved using name merging. In the earlier example, we split `a` into two names `a1` and `a2` according to the live ranges for `a`, and these new names were assigned to the same memory bank. Note that these live ranges do not conflict. This means that they can in turn be assigned to the same location in memory.



(a) Live Range After Local Variable Merging (b) Partitioned Memory

Fig. 13. Name merging for local variables

The compiler cannot only merge nonconflicting live ranges of the same variable, as in the case of the variable `a`, but also nonconflicting live ranges of different variables. We see in Figure 13(b) that two names `b` and `d2` are merged to save one word in Y memory.

The algorithm for name splitting and merging has practically polynomial time complexity even though name splitting and merging basically use a theoretically NP-complete graph coloring algorithm. That is, asymptotically the time required for name splitting and merging scales at worst case as $n2^n$ for dual data memory banks. This is yet much faster than conventional graph coloring for register allocation, whose time complexity is $O(nm^n)$ where m is typically more than 32 for GPPs. It has already been empirically proven that in practice, register allocation with such high complexity runs in polynomial time thanks to numerous heuristics such as pruning. Name splitting and merging also runs in polynomial time, as we will demonstrate in the following section.

7. EXPERIMENTS

To evaluate the performance of our memory bank assignment algorithm, we implemented the algorithm in our Soargen compiler and conducted experiments with DSP benchmark suites on a commercial DSP, the Motorola DSP56000 [Motorola Inc. 1995]. The performance is measured in two metrics: size and time. In this section, we report the performance obtained in our experiments, and compare our results with other work.

7.1 Measurements of Code Sizes and Compilation Times

We compare our performance with that of SPAM to demonstrate the effectiveness of our memory bank assignment. Besides, by comparing with SPAM, we can analyze the pros and cons of our decoupled approach as opposed to their coupled approach. Unfortunately, for some reason, we could not port SPAM successfully on our machine platform. However, in their recent literature [Sudarsanam and Malik 2000], they reported several measurements taken with a set of programs from *ADPCM* [Lee et al. 1997] and *DSPStone* [Zivojnovic et al. 1994] benchmark suites. In this paper, therefore, we borrow the numbers from their literature in a comparison with our experimental result. In Table I, we list the programs

Table I. Comparison of sizes of the code generated by both approaches for ADPCM and DSPStone programs and comparison of their compilation times for code generation

	Coupled approach			Decoupled approach			
Benchmarks	C_{unopt} (words) / C_{opt} (words)	Size ratio	Compile $time_{Sun}$	C_{unopt} (words) / C_{opt} (words)	Size ratio	Compile $time_{Intel}$	Compile $time_{Sun}$
complex_multiply	33 / **30**	0.90	2	55 / **40**	0.72	0.08	0.31
convolution	49 / **45**	0.91	308	40 / **37**	0.92	0.04	0.16
fir2dim	178 / **158**	0.88	5482	63 / **58**	0.92	0.2	0.86
iir_biquad_N_sections	132 / **128**	0.92	1632	124 / **116**	0.93	0.06	0.23
least_mean_square	115 / **102**	0.88	2776	176 / **159**	0.90	0.36	1.62
matrix_multiply_1	89 / **85**	0.95	1011	129 / **120**	0.93	0.04	0.14
adapt_quant	235 / **227**	0.96	4399	256 / **252**	0.98	0.05	0.13
adapt_predict_1	231 / **209**	0.90	8105	273 / **264**	0.96	0.19	0.91
iadpt_quant	85 / **81**	0.95	324	63 / **59**	0.93	0.11	0.51
scale_factor_1	74 / **66**	0.89	266	60 / **58**	0.96	0.03	0.08
speed_control_2	277 / **247**	0.89	5217	217 / **200**	0.92	0.75	3.68
tone_detector_1	84 / **75**	0.89	536	75 / **71**	0.94	0.06	0.25

that were compiled by the SPAM compiler and evaluated on their machine platform along with their performance figures.

Table I shows the values in the column labeled $Code_{unopt}$ that represent the sizes (in words) of the unoptimized codes, which are obtained immediately after instruction selection. The unoptimized code is fed into the subsequent code generation phases to produce the final code optimized for dual data memory banks. The sizes of these optimized codes are listed in the column labeled $Code_{opt}$. In the table, we compare the sizes of these unoptimized codes in both approaches. Note here that these initial code sizes for each approach are different. This is mainly because our code generator runs on a compilation infrastructure completely different from theirs [Araujo and Malik 1998].

In the table, we also compare the optimized code sizes and the code size reduction. We compute the amount of size reduction, which is listed in the column labeled *Size ratio*, by dividing the optimized code size by the unoptimized code size. Overall the sizes of our optimized code are comparable to those of their code. In particular, the code size reduction in both approaches ranges approximately from 5 to 10%, which is also quite comparable. These results indicate that our memory bank assignment algorithm is as effective as their simultaneous reference allocation algorithm in most cases.

To the contrary, the difference of compilation times is significant, as depicted in Table I. According to their literature [Sudarsanam and Malik 2000], all experiments of SPAM

were conducted on Sun Microsystems Ultra Enterprise featuring eight processors and 1GB RAM. We experimented on two machine platforms: one is the same Sun Microsystems Ultra Enterprise but with two processors and 2GB RAM, and the other is a Linux system running on dual Intel Pentium III and 512MB RAM. Although our compilation times on the Sun processors are about four times slower than those on the Intel processors, they both are still up to four orders of magnitude faster than SPAM's compilation times. Despites these differences of machine platforms, therefore, we believe that such large difference of compilation times clearly demonstrates the advantage of our approach over theirs in terms of compilation speed. We credit this mainly to the decoupled approach which facilitated our application of various fast heuristic algorithms that individually conquer each subproblem encountered in the code generation process for the dual memory bank system.

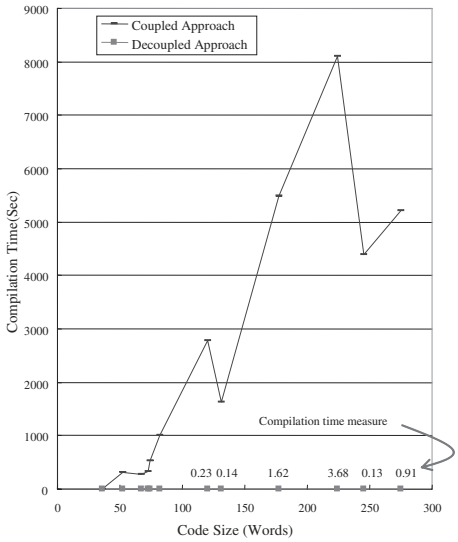


Fig. 14. Comparison of compilation time increases for both approaches as the code size increases

We can show this fact more clearly in Figure 14, which shows compilation time increasing according to code size. Although compilation time is also influenced to some extent by various other factors that include code complexity, number of variables, and dependencies between variables, the figure indicates that code size appears to be the most significant factor that affects compilation time. One thing we can clearly notice from Figure 14 that compilation times of the coupled approach dramatically (almost exponentially) increase as the code size increases while our approach results in quite consistently small compilation times, as compared to the coupled approach.

7.2 Measurements of Execution Times

To estimate the impact of code size reduction on the running time, we generated three versions of the code as follows.

uncompacted code. The first version is our uncompacted code generated without memory bank assignment phase.

compiler-optimized code. The uncompacted code is optimized for DSP 56000 by using the techniques in Section 5 to produce the code.

Table II. Comparison of execution times of Soargen output code with those produced by hand-optimization

Benchmark	Time_{unopt}	Time_{opt}	$\text{S}_{\frac{opt}{unopt}}$	Time_{hand}	$\text{S}_{\frac{hand}{opt}}$	$\text{S}_{\frac{hand}{unopt}}$
complex_multiply	62	**48**	22.5%	**44**	8.3%	29.0%
convolution	640	**592**	7.5%	**496**	16.2%	22.5%
fir2dim	1172	**1094**	6.6%	**924**	15.5%	21.1%
iir_biquad_N_sections	157	**145**	7.6%	**140**	3.4%	10.8%
least_mean_square	2320	**2052**	11.5%	**1821**	11.2%	21.5%
matrix_multiply_1	4632	**4282**	7.5%	**3926**	8.3%	15.2%
adapt_quant	226	**216**	3.5%	**212**	1.8%	6.2%
adapt_predict_1	4224	**3946**	6.5%	**3898**	1.2%	7.7%
iadpt_quant	130	**122**	6.0%	**118**	1.6%	9.2%
scale_factor_1	168	**164**	2.3%	**158**	3.2%	6.0%
speed_control_2	410	**374**	8.7%	**358**	4.2%	12.7%
tone_detector_1	176	**168**	4.5%	**160**	4.7%	9.1%
Average	-	-	7.9%	-	6.6%	14.2%

hand-optimized code. The uncompacted code is optimized by hand. We hand-optimized the same code that the compiler used as the input so that the hand-optimized one may provide us with the upper limit of the performance of the benchmarks on DSP56000.

Table II shows measurements of execution times represented in the number of instruction cycles that each benchmark program takes to run. For direct comparison with the compiler generated code, we hand-optimized the same code that our compiler used as the input so that the hand-optimized code may provide us with the upper limit of the performance of the benchmarks on DSP56000. We also compute speedups of the execution times of one version Y over those of the other X as follows:

$$\text{Speedup}_{\frac{X}{Y}} = (1 - \frac{\text{Time}_X}{\text{Time}_Y}) \times 100.$$

In Table II, we can see that the average speedup of our compiler-optimized code over the unoptimized code is 7.9%, and those of hand-optimized code over the compiler-optimized code is 6.6%. These results indicate that the compiler has achieved roughly the half of the performance we could get by hand optimization. Although these numbers may not be that impressive, the results also indicate that, in six benchmarks out of the twelve, our compiler has achieved comparable performance gains, as compared with hand optimization.

Of course, we also have several benchmarks, such as `fir2dim`, `convolution` and `least_mean_square`, in which our compiler has much room for improvement. According to our analysis, the main cause that creates such difference in execution time between the compiler-generated code and the hand optimized code is the incapability of our compiler to efficiently handle loops. To illustrate this, consider the example in Figure 15, which shows a typical example where software pipelining is required to optimize the loop.

Notice in the example that a parallel move for variables `a` and `b` cannot be compacted into the instruction word containing `ADD` because there is a dependence between `MPY` and them. However, after placing one copy of the parallel move into the preamble of the loop, we can now merge the move with `ADD`. Although this optimization may not reduce the total code size, it eliminates one instruction within the loop, which undoubtedly would reduce the total execution time noticeably.

This example informs us that, since most of the execution time is spent in loops, our compiler cannot match hand optimization in run time speed without more advanced loop

```
    DO  #16, L10
    MOV          X:a,X0 Y:b,Y0
    MPY X0,Y0,A  X:c,X1 Y:d,Y1
…
    ADD X1,Y1,A
    MOV          A,X:e
L10
```

(a) Compiled Compacted Code by Our Approach

```
    MOV          X:a,X0 Y:b,Y0
    DO  #15, L10
    MPY X0,Y0,A  X:c,X1 Y:d,Y1
…
    ADD X1,Y1,A  X:a,X0 Y:b,Y0
    MOV          A,X:e
L10
…
```

(b) Hand-Optimized Compacted Code

Fig. 15. Compaction difference between our compiled code and hand-optimized code

optimizations, such as software pipelining, which are based on rigorous dependence analysis. Currently, this issue remains for our future research.

7.3 Extension to n Memory Banks

In Section 5.2, we extended our dual memory bank assignment algorithm for arbitrary n data memory banks. To see the impact of our extended algorithm on the performance, we made an experiment to estimate the impact of code size reduction and dynamice instruction counts, according as the number of memory banks was increased.

Tables III and IV respectively show these results of applying the extended algorithm of Figure 10 to benchmark programs for various numbers of ALUs and memory banks. The results were obtained by simulating the code that was generated for multiple ALUs and multiple memory banks.

In Table III, we can see that the average ratio of code size reduction is the best result 0.86 when the machine has two ALUs and three memory banks and three ALUs and three memory banks. Also in Table IV, the average ratio of dynamic count is the best result 0.88 when the machine has two ALUs and two memory banks. When the machine has a single ALU, these results indicate that we do not see much improvement in terms of both the code size and the dynamic count for more than two memory banks. The main reason is that at most two source operands need to fetched from memory for each ALU operation in our compacted code. Therefore, we applied our algorithm to the machine having multiple ALUs. From these results we have concluded that improving performance by exploiting more than two data memory banks can be only expected for SIMD (single instruction multiple data) or VLIW architectures. This is because these machines have more than one ALUs, which consequently would require higher memory bandwidth for simultaneously fetching more source operands at each cycle than conventional SISD (single instruction single data) machines.

8. CONCLUSIONS & FUTURE WORK

Many DSP vendors provide a dual data memory bank system that allows applications to access two memory banks simultaneously. Unfortunately, several existing compilers were not able to fully exploit this dual memory feature. In this paper, we proposed a decoupled approach for supporting multiple memory architecture, where *register class allocation*, *code compaction*, *memory bank assignment*, *register assignment*, and *memory offset assignment* are performed separately. We presented a novel technique based on an MST algorithm for multiple memory bank assignment. We also presented *name splitting* and *merging* as additional techniques that improves our MST-based algorithm by using conventional graph coloring. This decoupled structure of code generation phases led us to simplify our data allocation algorithm for multiple memory banks and to run the algorithm

Table III. Comparison of code size according to the number of ALUs and memory banks for ADPCM and DSPStone programs

	1 ALU			2 ALUs			3 ALUs		
Benchmarks	n = 1	n = 2	n = 3	n = 1	n = 2	n = 3	n = 1	n = 2	n = 3
complex_multiply	55	40	40	54	38	34	54	38	34
convolution	40	37	36	40	35	32	40	35	32
fir2dim	63	58	57	61	55	55	61	55	55
iir_biquad_N_sections	124	116	114	120	112	112	120	112	112
least_mean_square	176	159	156	174	154	152	174	154	152
matrix_multiply_1	129	120	119	125	118	115	125	118	115
adapt_quant	256	252	248	253	248	242	253	248	242
adapt_predict_1	273	264	258	269	258	240	269	258	240
iadpt_quant	63	59	59	63	57	57	63	57	57
scale_factor_1	60	58	58	60	56	56	60	56	56
speed_control_2	217	200	194	212	196	182	212	196	182
tone_detector_1	75	71	71	73	69	69	73	69	69
Average ratio	1	0.92	0.91	0.98	0.89	0.86	0.98	0.89	0.86

Table IV. Comparison of dynamic instruction counts according to the number of ALUs and memory banks

	1 ALU			2 ALUs			3 ALUs		
Benchmarks	n = 1	n = 2	n = 3	n = 1	n = 2	n = 3	n = 1	n = 2	n = 3
complex_multiply	55	40	40	54	38	34	54	38	34
convolution	914	842	828	914	834	786	914	834	786
fir2dim	918	838	830	886	806	806	918	806	806
iir_biquad_N_sections	124	116	114	120	112	112	124	112	112
least_mean_square	2126	1914	1888	2094	1880	1864	2126	1880	1864
matrix_multiply_1	4234	3910	3896	4182	3824	3782	4234	3824	3782
adapt_quant	106	104	102	105	103	101	105	103	101
adapt_predict_1	3273	3235	3187	3241	3153	3009	3241	3153	3009
iadpt_quant	63	58	58	63	57	57	63	57	57
scale_factor_1	50	48	48	50	46	46	50	46	46
speed_control_2	147	139	135	145	138	132	145	138	132
tone_detector_1	62	59	59	61	58	58	61	58	58
Average ratio	1	0.92	0.91	0.99	0.90	0.88	0.99	0.90	0.88

reasonably fast.

The comparative analysis of the experiments revealed that our compiler achieved comparable results in code size, yet runs considerably faster than a previously described coupled approach. The analysis also showed that exploiting multiple memory banks by carefully assigning scalar variables to the banks brought about the speedup at run time. Finally, we presented some experimental evidence that dual data memory banks are generally sufficient for ordinary SISD-style machines, and that providing more than two banks would be mainly useful to SIMD and VLIW architectures which require higher memory bandwidth.

A number of interesting topics still remain open for future work. For instance, while our approach was limited to only scalar variables, memory bank assignement for arrays can result in a large performance enhancement because most computations are performed on arrays in DSP programs. This is actually illustrated in Table II, where even highly hand-optimized code could not make a significant performance improvement in terms of

speed although we made a visible difference in terms of size. This is mainly because the impact of scalar variables on the performance is relatively low as compared with the space they occupy in the code. Another interesting topic would be to perform memory bank assignment on arguments passed via memory to functions. This would require interprocedural analysis since otherwise a calling convention must be used so the caller can know the memory access patterns of the callee for passing arguments. Also, certain loop optimization techniques, like those listed in Section 7.2, need to be implemented to further improve the execution time of the output code.

REFERENCES

ARAUJO, G. AND MALIK, S. April 1998. Code Generation for Fixed-point DSPs. *ACM Transactions on Design Automation of Electronic Systems 3*, 2, 136–161.

BARUA, R., LEE, W., AMARASINGHE, S., AND AGARWAL, A. Nov. 2001. Compiler Support for Scalable and Efficient Memory Systems. *IEEE Transactions on Computers*.

CHO, J. AND PAEK, Y. June 2002. Efficient Register and Memory Assignment for Non-orthogonal Architectures via Graph Coloring and MST Algorithms. In *Workshop on Languages, Compilers and Tools for Embedded Systems*.

CORMEN, T., LEISERSON, C., AND RIVEST, R., Eds. 1990. *Introduction to Algorithms*. The MIT Press and McGraw Hill Book Company.

EYRE, J. AND BIER, J. Aug. 1998. DSP Processors Hits the Mainstream. *IEEE Computer*, 51–59.

GOULD, H. AND TOBOCHNIK, J. 1988. *Computer Simulation Methods*. Addison-Wesley Publishing Company, New York.

JUNG, S. AND PAEK, Y. Nov. 2001. The Very Portable Optimizer for Digital Signal Processors. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 84–92.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. Nov. 1997. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annaul IEEE/ACM Internation Symposium on Microarchitecture*. 330–335.

LEUPERS, R. AND KOTTE, D. 2001. Variable Partitioning for Dual Memory Bank DSPs. In *Proceedings of IEEE International Conference on Acoustics Speech and Signal Processing*. 1121–1124.

LIEM, C. 1997. *Retargetable Compilers for Embedded Core Processors*. Kluwer Academic Publishers.

Motorola Inc. 1995. *DSP56000 24-Bit Digital signal Processor Family Manual*. Motorola Inc., Austin, TX.

PANDA, P. 1999. Memory Bank Customization and Assignment in Behavioral Synthesis. In *Proceedings of ICCAD*. 477–481.

POWELL, D., LEE, E., AND NEWMAN, W. 1992. Direct Synthesis of Optimizaed DSP Assembly Code from Signal Flow Block Diagrams. In *Proceedings of IEEE International Conference on Acoustics Speech and Signal Processing*. 553–556.

PRIM, R. 1957. Shortest Connection Networks and Some Generalizations. *Bell Systems Technical Journal 36*, 6, 1389–1401.

SAGHIR, M. A. R., CHOW, P., AND LEE, C. G. 1996. Exploiting Dual Data-Memory Banks in Digital Signal Processors. *ACM SIGOPS Operating Systems*, 234–243.

SUDARSANAM, A. May 15, 1998. Code Optimization Libraries For Retargetable Compilation For Embedded Digital Signal Processors. Ph.D. thesis, Princeton University Department of EE.

SUDARSANAM, A. AND MALIK, S. April 2000. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. *ACM Transactions on Design Automation of Electronic Systems 5*, 2, 242–264.

ZIVOJNOVIC, V., VELARDE, J., SCHAGER, C., AND MEYR, H. 1994. DSPStone - A DSP oriented Benchmarking Methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*.