Published in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp.234–243, October 1996.

Exploiting Dual Data-Memory Banks in Digital Signal Processors

Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee
{saghir, pc, corinna}@eecg.toronto.edu
Department of Electrical and Computer Engineering, University of Toronto
10 King's College Road, Toronto, Ontario M5S 3G4
CANADA

Abstract

Over the past decade, digital signal processors (DSPs) have emerged as the processors of choice for implementing embedded applications in high-volume consumer products. Through their use of specialized hardware features and small chip areas, DSPs provide the high performance necessary for embedded applications at the low costs demanded by the high-volume consumer market. One feature commonly found in DSPs is the use of dual data-memory banks to double the memory system's bandwidth. When coupled with high-order data interleaving, dual memory banks provide the same bandwidth as more costly memory organizations such as a dual-ported memory. However, making effective use of dual memory banks remains difficult, especially for high-level language (HLL) DSP compilers.

In this paper, we describe two algorithms – compaction-based (CB) data partitioning and partial data duplication – that we developed as part of our research into the effective exploitation of dual data-memory banks in HLL DSP compilers. We show that CB partitioning is an effective technique for exploiting dual data-memory banks, and that partial data duplication can augment CB partitioning in improving execution performance. Our results show that CB partitioning improves the performance of our kernel benchmarks by 13%–49% and the performance of our application benchmarks by 3%–15%. For one of the application benchmarks, partial data duplication boosts performance from 3% to 34%.

1. Introduction

Outside the office, the world at large contains many examples of embedded applications [1]. These range from the simple control functions in household appliances such as microwaves, washing machines, and dryers to complex, performance-demanding programs that control the transmission drive of a car or send and receive voice signals for a cellular telephone. A common characteristic of many of these embedded applications is that they are aimed at the consumer market, which is extremely cost-sensitive. Nonetheless with advances in VLSI technology, consumer-targeted embedded applications have become ever more sophisticated while still remaining relatively inexpensive. This trend is expected to continue particularly in the rapidly expanding market of wireless communication [2].

Many of these embedded applications involve processing digital signals. Consequently, the most common type of processor used to implement these applications is the programmable digital signal processor (DSP) [3]. Such processors typically provide two memory banks for storing data plus a third memory bank for holding instructions for greater bandwidth. While exploiting separate memories for instructions and data is straightforward, compilers for DSPs rarely exploit the ability to make two data accesses simultaneously. As part of our research on compiling for DSPs [4], we have developed algorithms that effectively use the dual data-memory banks. This paper describes our efforts. In the remainder of this section, we first provide some background on DSPs and their use for embedded applications. We then describe in more detail the issues involved in exploiting dual memory banks.

1.1. Characteristics of Digital Signal Processors

Recently, it has been suggested that general-purpose microprocessors, sometimes with small extensions to their instruction-set architecture [5][6], could be used in place of DSPs for embedded applications involving image processing [7]. While it is possible for general-purpose microprocessors to provide the necessary performance, it is not clear that they can do so at the low costs demanded by the consumer market, particularly in the speech- and audio-processing segments. For example, a Motorola DSP56004 processor costs $16.80 [8] whereas the cost of an Intel Pentium processor ranges from $184 to $694 depending upon the clock frequency [9].

In addition to using inexpensive processors, embedded applications keep overall costs low in other ways. For example, designers of embedded systems use on-chip memory only and avoid the use of external memory as much as possible. While workstations are designed to execute a wide variety of different programs, embedded systems are often configured to execute a small, very application-specific set of programs. Memory costs in an embedded system can be minimized by ensuring that its few programs and possibly some static data like coefficients can be stored entirely in on-chip memory. On-chip memory ranges from 16KB to 200KB for fixed-point DSPs and from 16KB to 48KB for floating-point DSPs. Other ways for keeping system costs down include low power usage, small chip count, and use in high-volume applications.

The cost of programmable DSPs is kept low, in part, because they use smaller dies than general-purpose microprocessors [10]. Die sizes for microprocessors range from 160 mm^2 for an Intel Pentium to 250 mm^2 for a Sun SuperSparc/60 [11]. In contrast, die sizes for DSPs typically range from 25 mm^2 to 60 mm^2 [12]. Obviously the smaller die area, in turn, restricts the amount of hardware that can be included on a die. Nonetheless, by incorporating special hardware features, programmable DSPs are still able to obtain the high performance demanded by embedded applications. These special hardware features are designed to exploit compute-intensive program constructs that occur frequently in the signal processing algorithms that are at the core of many embedded applications. These features include hardware support for low-overhead looping, accumulator-based data paths, tightly-encoded instructions that specify the parallel execution of multiple independent operations, multiple pipelined functional units, an instruction memory, and two memory banks for storing data [3].

Many embedded applications operate in real-time environments, which impose stringent performance demands on the underlying system. The key to meeting these demands is to effectively use the special hardware features of a programmable DSP. In the past, this could best be accomplished by programming in assembly language. As functionality has grown and more sophisticated algorithms developed, DSP application programmers would prefer to write in a widely used, standard, portable, high-level language such as C or C++[1]. However, C compilers often are unable to generate code that can meet the performance requirements of embedded applications [14]. Much of the research for optimizing compilers has been for general-purpose microprocessors and has focussed on machine-independent optimizations [15]. Generating high-performance code for DSPs, however, requires optimizations that exploit processor-specific features. Although machine-dependent optimizations such as instruction scheduling and register allocation can be used with DSPs, there are other hardware features that are still used ineffectively by C compilers. In particular, these compilers are unable to efficiently use dual data-memory banks.

1.2. The Dual Memory Bank Problem

Dual memory banks are beginning to appear also in the first-level caches of high-performance microprocessors [16]. These typically use low-order interleaving. In contrast, dual memory banks in DSPs are implemented using high-order interleaving. In other words, consecutive address locations are stored in the same memory bank. This is because most memory accesses that can occur in parallel in signal processing algorithms are to elements of different arrays. A typical example can be found in an Nth-order FIR filter, an algorithm that is used in many embedded applications. Figure 1(a) shows the filter written in C while Figure 1(b) shows the corresponding code in the assembly language of the Motorola DSP56001 [17]. In addition to illustrating how dual, high-order interleaved memory banks can be used, Figure 1(b) also shows how other specialized DSP hardware features are used.

The assembly version is a simple example of software pipelining [18] whereby elements of arrays A and B are pre-loaded in the iteration before the one in which the elements are actually used. This is done to expose as much parallelism as possible. Instruction `[2]` is a special *repeat* instruction that causes the subsequent instruction to be executed N-1 times.

The DSP56001 is capable of executing several independent operations per instruction in a fashion similar to a VLIW architecture. However, the types of operations that can be combined to form an instruction are more limited in the DSP56001. In Figure 1(b), instructions `[1]`, `[3]`, and `[4]` are examples of such restricted VLIW instructions. Instruction `[1]` executes five operations: the accumulator is cleared (`CLR A`); register X0 is loaded from X memory and address register R0 is post-incremented (`X: (R0)+,X0`); and register Y0 is loaded from Y memory and address register R4 is post-incremented (`Y: (R4)+,Y0`). Instruction `[3]` executes a multiply-accumulate operation (`MAC`) and two memory loads from X and Y memory into the X0 and Y0 registers; it also post-increments address registers R0 and R4. Finally, instruction `[4]` executes a multiply-accumulate-and-round (`MACR`) operation.

```
sum = 0;
for (i = 0; i < N; i++)
  sum += A[i] * B[i];
```

(a)

```
[1]   CLR    A          X:(R0)+,X0    Y:(R4)+,Y0
[2]   REP    #N-1
[3]   MAC    X0,Y0,A    X:(R0)+,X0    Y:(R4)+,Y0
[4]   MACR   X0,Y0,A
```

(b)

Figure 1: (a) C-language implementation of an N-th order FIR filter (b) Motorola DSP56001 assembly language implementation of the same filter.

1. There are also graphical languages, such as the Signal Dataflow Language developed at the University of California at Berkeley [13], which allow programmers of embedded applications to express their algorithms in a more natural manner. Compilers for such graphical languages, nevertheless, often generate C code; thus a good C compiler is still a necessary component to high performance.

Instructions `[1]` and `[3]` demonstrate the effective use of the dual memory banks. By allocating array A to one memory bank and array B to the other, it is possible to access an element from each array in one instruction. If the arrays had both been allocated to the same memory bank, the elements would have to be accessed sequentially with two instructions and thus increase the size of the loop from one instruction to two instructions, reducing performance by a factor of two.

Thus, by treating each array as a single entity and allocating the arrays to the appropriate banks, parallel accesses to all elements of the two arrays are guaranteed. Note that to avoid bank conflicts in a two-bank, *low-order* interleaved memory system, we need to consider which element of each array is accessed first when parallel accesses are possible. In contrast, we only need to know which arrays could be accessed in parallel when using high-order interleaved memory. The simpler effective use of high-order interleaved memory is likely the reason why most DSPs implement it.

A straightforward technique that uses dual memory banks efficiently with respect to performance is to duplicate all data. By doing so, there would no longer be any need for an allocation algorithm. Duplicating data, however, also increases the memory cost of an embedded system due not only to the duplicated data but also to additional store instructions needed to maintain the integrity of both copies of the data. There is no benefit in duplicating all data indiscriminately when the comparable performance could be obtained with far less memory by judiciously allocating arrays to the appropriate memory banks. Nonetheless during our investigation, we came across programs for which the only software solution to providing parallel accesses is to duplicate some of the data. In such situations, the gain in performance must be weighed against the increase in memory cost.

The remainder of this paper describes the compiler algorithms we have developed and presents performance results illustrating their effectiveness. We first discuss research related to this work in Section 2, and then in Section 3 we describe the two algorithms: compaction-based data partitioning and partial data duplication. We also discuss the performance impact of the additional bookkeeping instructions required when duplicating data. In Section 4, we demonstrate the effectiveness of these techniques by comparing the performance of code using these algorithms to the performance of an ideal system that guarantees simultaneous accesses. Because data duplication incurs additional cost in the form of increased memory storage whereas partitioning does not, we also explore the trade-off between performance gain and increased memory cost when using these techniques.

2. Previous and Related Work

Traditionally, the exploitation of dual data-memory banks in DSPs has been the responsibility of the programmer. Whether programming is done in assembly language or in a high-level language, the programmer has to allocate data manually by using assembler directives or compiler pragmas [19]. This makes it very difficult to ensure an efficient exploitation of dual memory banks, especially in large embedded applications. We are unaware of any commercial DSP compilers that automatically exploit dual data-memory banks.

In the academic community, a group of researchers at Princeton University are investigating techniques for exploiting multiple memory banks that are available on DSPs such as the Motorola DSP56001 and the NEC77016 [20]. Such processors place constraints on which registers can be used to store data from a particular data bank, thus making register allocation and data allocation interrelated problems. This increases the dimensions of the solution space and has led the researchers to develop a computationally-intensive, simulated annealing algorithm that labels a constraint graph whose nodes represent symbolic register and memory references, and whose edges represent dependences and constraints among those references. The researchers also considered a simpler greedy algorithm that processes the variables in the order that they are used and allocates them to the memory banks in an alternating manner. Interestingly, the results of their study show that the performance gained from the simulated annealing algorithm is comparable to that gained from the greedy algorithm, thus suggesting that the problems of register allocation and data partitioning could be decoupled without any loss in performance. By contrast, the DSP architecture that we use places no restrictions on the usage of registers (we briefly describe the architecture in Section 3). We are therefore able to exploit the orthogonality between register allocation and data allocation to derive a simpler solution that produces good results. This is one of our goals in developing new DSP architectures that combine high performance with HLL programmability.

Another approach to generating DSP code is to rely on code synthesis tools such as the Ptolemy package developed at the University of California at Berkeley [13]. Ptolemy is an environment for simulation, prototyping, and software synthesis of heterogeneous systems. It enables designers to specify embedded applications in the form of hierarchical dataflow graphs of functional blocks. A block may be implemented either as a hand-optimized assembly-language routine or as a high-level language routine. To manage the interaction between different blocks, Ptolemy uses dataflow scheduling techniques, and synthesizes code that effectively stitches the different block implementations together. To simplify the interaction between different block implementations, the exploitation of dual memory banks is kept to a minimum, especially for data that is shared among different blocks. The resulting code may therefore not take full advantage of dual data-memory banks.

In a somewhat different context, low-order interleaved memory banks are used in the Multiflow VLIW computer and a memory bank disambiguator is used by the Multiflow compiler to identify the memory banks that memory operations access [21]. This enables the compiler to identify instances where different memory operations can occur in parallel. Data accesses that cannot be disambiguated at compile time are issued to a slower, central memory controller. Using a memory bank disambiguator is a different approach to the problem of using multiple data banks effectively. In contrast, our algorithm determines which bank data should be allocated to so as to increase the opportunities for parallel accesses. Since the algorithm determines where data are stored, there is no need for a memory bank disambiguator in our scheme. Instead, our algorithm requires alias information to determine which data a memory operation accesses. Because this may not always be possible to determine at compile-time, as when pointers are passed as parameters on the stack, our algorithm may result in a conservative data allocation. To overcome this limitation, we are currently considering the use of source-code data annotations such as those used in the SUIF compiler system [22].

3. Exploiting Dual Memory Banks

We will now describe the two compiler algorithms that we developed to use dual memory banks effectively. For both approaches, we treat an array as a monolithic entity that is allocated in its entirety to a single memory bank. This restriction is a direct consequence of the fact that the two memory banks are high-order interleaved. To provide the reader with a context for understanding the algorithms, we first describe the main features of our target DSP architecture and the general structure of our optimizing C compiler.

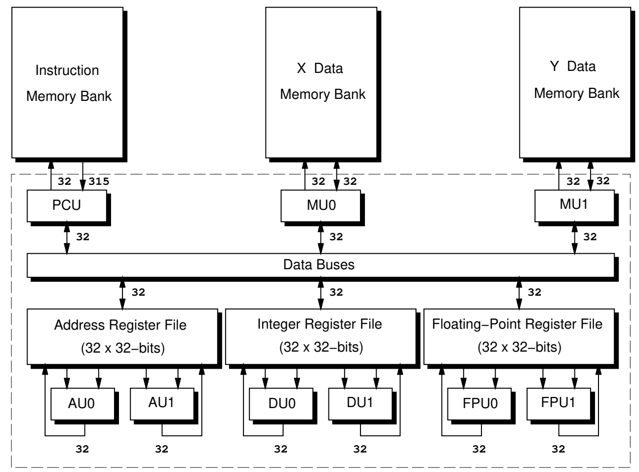


Figure 2: VLIW Model Architecture

Figure 2 shows the target processor, which is a Very Long Instruction Word (VLIW) architecture [23] with nine functional units. Each functional unit has a single clock-cycle latency, a common feature among most DSP implementations. The features that are of interest to this paper are the two single-ported data-memory banks (X and Y) and the two memory-access units (MU0 and MU1). Memory unit MU0 is used to access bank X while MU1 accesses bank Y.

To exploit the parallelism in the hardware, the compiler is responsible for generating VLIW instructions that specify the execution of parallel machine operations. We mainly chose a VLIW model architecture because of the flexibility it offers the compiler in exploiting parallelism, and its ability to support common DSP architectural features. This enabled us to develop the compiler with little concern for architectural constraints. However, a straightforward implementation of this design is too costly for an embedded system, and our future research will involve exploring alternative, less costly implementations that have comparable performance.

Our optimizing C compiler currently consists of a GNU-C [24] compiler front-end and an optimizing back-end. The GNU-C front-end translates C-language source programs into a sequence of unpacked, machine operations. These operations are fed into the optimizing back-end, which applies a number of architecture-specific optimizations to them and generates VLIW instructions. Of particular interest to this paper are the data allocation pass and the operation compaction pass. The goal of the allocation pass, which executes before the compaction pass, is to assign variables to the two data-memory banks so as to expose as much parallelism among load and store operations as possible. The compaction pass then uses the memory assignments determined by the allocation pass to pack memory operations as well as other operations that can execute in parallel into VLIW instructions.

The two algorithms we developed – compaction-based data partitioning and partial data duplication – are both performed in the data allocation pass of the post-optimizer. The partitioning algorithm uses an interference graph to partition a program's variables into two sets, which correspond to the two memory banks. The duplication algorithm identifies memory accesses that could be executed in parallel but cannot because their corresponding vari-

```
for (every basic block){
  generate data-dependence graph;
  assign priority to each op;
  calculate data-ready set (DRS);
  while (DRS not empty) {
     sort DRS by priority;
     form new long instruction;
     for (all ops in DRS) {
        if (op data-compatible) {
           if (op function-unit compatible) {
              add op to instruction and mark it as scheduled;
              if (op == ld/st)
                 ld/st_i = op;
           }
           else if (op == ld/st) {
              ld/st_j = op;
              if (ld/st_i and ld/st_j access different vars/arrays)
                 add edge to graph;
              else
                 mark var/array for duplication;
           }
     }
     calculate new DRS;
}
```

Figure 3: Pseudo-Code for Building Interference Graph

ables cannot be allocated appropriately. Duplicating such variables makes parallel execution possible, albeit at the cost of increased storage requirements.

3.1. Compaction-Based Data Partitioning

Ideally, data should be allocated so that simultaneous accesses are possible without having to resort to duplicating data. This can be done by partitioning the data into two sets, one for each memory bank. Partitioning data, in turn, requires addressing three interrelated issues. First, we must establish partitioning relationships among different data. This involves identifying instances when pairs of data should be stored in separate memory banks. Second, we must define a suitable cost metric that can be assigned to every partitioning relationship. This helps quantify the importance of satisfying a particular relationship and reflects its impact on performance. Finally, we need to define rules for partitioning the data in a manner that satisfies as many partitioning relationships as possible and yields the best performance.

To represent the partitioning relations among the program data, we use an undirected interference graph. The nodes of this graph represent the variables of a program, and an edge connecting a pair of nodes indicates that the corresponding variables may be accessed in parallel and that those variables should be stored in separate memory banks to allow parallel access to actually occur. As we shall see shortly, it is not always possible to allow the parallel access to happen.

Determining the nodes of the interference graph is straightforward. Adding the edges is done by augmenting the operation-compaction algorithm of the compiler to identify all pairs of memory operations in a basic block that can execute in parallel. The compaction algorithm is based on the list scheduling algorithm used in local microcode compaction [25]. Figure 3 shows the pseudo-code for the algorithm; italics are used to show the portions of the algorithm used for the data allocation pass. Note that this version of the compaction algorithm is used in the allocation pass to construct the interference graph, and that operations are not actually scheduled in long instructions until the compaction pass.

Because the compaction algorithm is local in its scope, it is applied to every basic block. A data-dependence graph is first generated to determine the order in which operations must be scheduled. A priority, equal to the number of descendents an operation has in the dependence graph, is also assigned to every operation to facilitate operation scheduling. The data-ready set (DRS), which contains all operations that are ready for scheduling at any given time, is then calculated. The algorithm's main loop iterates until all operations in the block are scheduled and a new DRS can no longer be calculated.

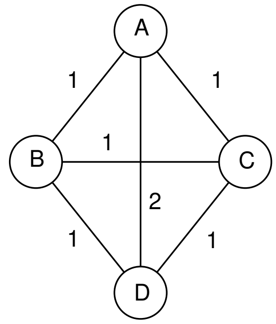
During each iteration, the DRS is first sorted according to the priority values of its operations. A new, empty, long instruction is then formed to schedule as many operations from the DRS as possible. Each operation in the DRS is checked for data- and function-unit-compatibility with already scheduled operations. Checking for data-compatibility helps generate tighter code by allowing an operation to be scheduled if it has an anti-dependency with another operation that is already scheduled. Checking for function-unit-compatibility ensures that no hardware resource conflicts will occur. If an operation is both data- and function-unit-compatible, it is added to the instruction and marked as being scheduled. Once all operations in the DRS have been processed, a new DRS is calculated and another iteration is executed.

To add edges to the interference graph, we augmented the compaction algorithm with some additional processing of memory operations. The first memory operation in the DRS can be both data-compatible and function-unit-compatible. In this case, it is scheduled in the instruction and is also saved in the event another memory operation is encountered when processing the current DRS.

Subsequent memory operations in the DRS can also be data-compatible, but cannot be function-unit-compatible. In this case, a memory operation is independent of the operations already scheduled in the instruction, including the first memory operation, but uses the same memory unit as the already-scheduled memory operation. The two memory operations could therefore be scheduled in parallel only if they were assigned to different memory units. The algorithm records this fact by adding an interference edge between the two variables that the memory operations access. However, if the two memory operations access the same variable or array, no edge is added to the graph, and instead, the variable is

```
D[i] = A[j] + B[k]
B[i] = B[j] - D[k]
C[i] = B[j] - C[k]
C[i] = A[j] + C[k]
for (i = 0; i < 5; i++) {
   .
   C[i] = A[i] + D[i]
   .
}
A[i] = C[j] + D[k]
```

(a)



(b)

Figure 4: (a) Example Program and (b) Corresponding Interference Graph

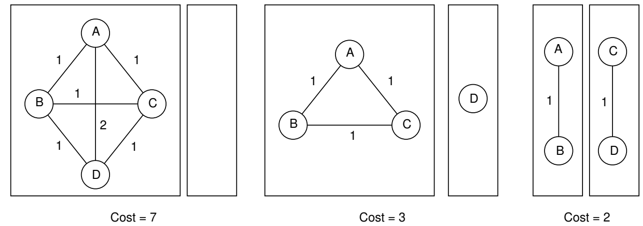
marked for duplication. The use of duplication is discussed in more detail in Section 3.2.

To account for the possibility of scheduling any subsequent memory operation in the DRS in parallel with the first memory operation, subsequent memory operations are not marked as being scheduled. This way, an interference edge is added between the variable accessed by the first memory operation and all variables accessed by subsequent memory operations. Since subsequent memory operations are not scheduled, they will be added to the next DRS that is calculated. Again, only the first memory operation in the new DRS will be scheduled and an interference edge is added between the variable it accesses and all variables accessed by subsequent memory operations. Thus, when a DRS can no longer be calculated, an interference edge will have been added between every pair of variables that could be accessed in parallel in the basic block.

Since the edges of the interference graph represent partitioning relationships among program data, we assign a weight to each edge to represent the degradation in performance if the corresponding variables are not accessed in parallel. The weight is equal to the loop nesting depth of the memory operations used to access the data. We chose this weight as a heuristic measure to ensure that the highest priority is given to exploiting load/store parallelism inside inner program loops. Other, more accurate, cost measures may also be used, including profile-driven and programmer-supplied costs. However, as the results in Section 4.1 show, using more accurate edge-weights does not always improve performance. Instead, other factors, which we discuss in Section 3.2, can also result in poor performance.

To demonstrate how the interference graph is built, Figure 4 shows an example program and its corresponding interference graph. In the program, every pairing of the arrays A, B, C, and D may be accessed simultaneously. Since each node in the graph corresponds to one of the arrays, an edge is therefore added between every pair of nodes in the graph. Since arrays A and D can be accessed in parallel inside the loop, edge *(A, D)* is assigned a weight of 2. On the other hand, since all other pairs of arrays can be accessed in parallel only outside the loop, all other edges of the graph are assigned a weight of 1.

Once the interference graph is constructed, the nodes are partitioned into two sets by searching for a minimum-cost partitioning among all possible partitionings. Although this problem is NP-complete [26], we use a greedy algorithm to ensure a near-optimal partitioning. This is by no means the only method that can be used to partition the nodes of the graph, and other algorithms, such as graph colouring, will probably work just as well. Our choice, however, was based on the simplicity of the greedy algorithm and our desire to validate our approach to data partitioning. Moreover, our performance results later showed that the greedy algorithm yields near-ideal performance, and this precluded our use of more sophisticated partitioning algorithms.



(a) (b) (c)

Figure 5: Partitioning the Nodes of an Interference Graph

Figure 5 shows how the nodes of an interference graph are partitioned. Initially, the algorithm stores all nodes in one of two sets, with the second set being empty. It also sets the initial cost for the partitioning to the sum of the weights of the edges connecting the nodes in the first set. This is shown in Figure 5(a). The algorithm then selects a node from the first set such that its removal results in the greatest decrease in cost, and stores the node in the second set. This is shown in Figure 5(b), where node D is moved from the first set to the second. The algorithm then selects another node from the first set such that its removal from the first set and its addition to the second set results in the greatest decrease in cost. Here, it should be noted that adding a node to the second set may increase the cost if the node is edge-connected to any of the nodes in the second set. That is why a node is only moved from the first set to the second if its removal results in a net decrease in cost. This is shown in Figure 5(c), where node C is moved to the second set. This process continues until it is no longer possible to reduce cost, at which point the data is partitioned. This, again, is shown in Figure 5(c).

Upon partitioning the program's data, the variables corresponding to the nodes in the two sets are stored in separate memory banks. Assigning global variables is fairly straightforward and requires only minor program changes involving memory-bank assembly directives. For local variables, some small changes to the instruction sequence are required. To allow parallel accesses to local variables, we allocate two program stacks, one for each memory bank, each with its own stack and frame pointers. Prior to partitioning, all local variables are allocated to one stack. After partitioning, both stacks are in use and thus the code must be modified to reflect this. For example, the code that is initially used to allocate space for local variables at the beginning of a function is replaced by code that allocates space for the two stacks. Moreover, the offsets used to calculate local variable addresses must be remapped to new values that reflect the variables' new locations on the different stacks.

In addition to the parallelism in accessing program variables, there is also parallelism in saving and restoring registers when entering and exiting functions. To exploit this parallelism, we augmented our algorithm to assign successive save/restore operations to alternating memory banks, thus partitioning the save and restore operations in a mechanical manner that exploits the two stacks.

Once all variables have been assigned to a memory bank, each memory operation is tagged with the bank that stores the data it is accessing. This information is later used by the operation-compaction pass for packing operations into long machine instructions.

Because we are using a compaction algorithm to construct the interference graph, we call our algorithm compaction-based (CB) data partitioning. The time complexity for constructing the interference graph is $O(B.n^2)$, where n is the number of operations in

```
for (n = 1; n < r; n++) {
  R[n] += signal[n] * signal[n+m];
}
```

Figure 6: C loop for calculating an autocorrelation vector

a basic block and B is the number of basic blocks in a program. The time complexity of partitioning the interference graph using the greedy algorithm is $O(v^2)$, where v is the number of nodes in the graph, which is also the number of variables and arrays in the program. Finally, the time complexity of assigning variables and arrays to their corresponding memory banks is O(v). Thus, the time complexity of our CB data partitioning algorithm is $O(B.n^2+v^2)$.

3.2. Partial Data Duplication

Although many of our benchmark programs approached ideal performance using compaction-based partitioning, some did not. Initially, we thought the poor performance was due to the use of poorly approximated weights for the edges in the interference graph. Recall that the weights are used to determine which pairs of variables should be given higher priority while partitioning. However, as we demonstrate in Section 4.1, using profiling to derive more accurate weights had a negligible effect on the performance of our benchmark programs.

Upon further investigation, we determined that the loss in performance was due to data-access patterns like the one shown in Figure 6, where accesses are made to two different elements in the same array. In such a case, the data cannot be split to allow simultaneous accesses because the entire array is stored in one memory bank. There are three possible solutions to this problem: using a low-order interleaved memory system, using dual-ported memory cells, or duplicating the data and storing a copy in each memory bank. We discuss each of these in turn.

Using low-order interleaving, where successive elements in an array are stored in different memory banks, would provide dual parallel access for the example in Figure 6 but only if the value of `m` is odd. Even values of `m` would cause the two references to access the same memory bank. Hence low-order interleaving does not provide a general solution for such situations.

Another solution is to use dual-ported memory cells instead of separate single-ported memory banks. Two elements from the same array could then be accessed simultaneously even when the entire array is stored in one memory bank. Each access would be made through a separate port. Additionally, with dual-ported memory, no special data partitioning is needed to guarantee simultaneous accesses.

However, the major disadvantage to using dual-ported memory is its cost. Adding another port to each cell increases the area as well as the power consumption of the memory system. Moreover, the memory access time may be negatively affected. Finally, the dual-ported nature of such a memory system may not be necessary for all applications. Because most applications can rely on a data partitioning algorithm to obtain the necessary memory bandwidth, providing a dual-ported memory for such applications is not cost-effective.

The third solution, which is implemented entirely in software, is to duplicate the data in both memory banks. The immediate cost is the extra memory area required to hold the duplicated data as well as the additional instructions needed to maintain the integrity of the duplicated data. However, with an effective data partitioning algorithm, it is not necessary to duplicate data for all applications nor for all variables in a particular application. Thus, by judiciously choosing which data to duplicate, increased cost would only be incurred when a gain in performance is possible. Moreover, although memory size is fixed, the use of it remains flexible when using data duplication in this manner. Memory is "wasted" for duplication only when necessary; otherwise it can be used as additional storage for those applications that do not require duplication.

To study the impact of data duplication on performance as well as on cost, we augmented the CB data-partitioning algorithm to replicate all arrays that could be accessed simultaneously. Such arrays can easily be identified while constructing the interference graph. Recall from Figure 3 that adding an edge to the graph requires a pair of memory operations to be checked to determine if they access the same array. If they do, the array must be duplicated to allow parallel access. Since we are able to determine at compile-time which arrays need to be duplicated, we call this approach *partial data duplication.*

To avoid fragmenting memory, we first allocate duplicated variables to both banks before other variables. For global variables, this enables the same address to be used to access the same variable from both data banks. For local variables, which are stored on the two stacks, this enables the same offset to be used for calculating the address that accesses the duplicated variable from both data banks.

We also modified the algorithm to maintain the data integrity of duplicated variables: each time the algorithm identifies a store operation to a duplicated variable, it adds another store operation to the program to keep the data in both memory banks current. For local variables, an additional stack operation is also needed to ensure that the additional store operation accesses the correct stack location.

Duplicating data may not always result in an increase in performance. This is because the additional store and stack operations could potentially degrade performance. This would happen if the compaction pass could not pack these additional operations into existing VLIW instructions and would have to then create additional VLIW instructions. If this were to occur in critical portions of the code that account for a significant portion of the execution time, it is possible that any performance gains due to parallel memory access could be negated by the performance degradations due to the additional operations, with the overall effect being either no change in performance or even possibly a decrease in performance.

Finally, duplication could complicate interrupt handling. Because stores to different copies of duplicated data may be scheduled in different instructions, it is possible that an interrupt may occur after the instruction containing a store to one copy of duplicated data and before the instruction containing the store to the other copy. It is further possible that the interrupt may update the duplicated data, a common occurrence in embedded systems where external data is being fed to the system on a continuous basis. To ensure that both copies are the same even in the presence of interrupts, a special pair of store operations could be used for updating duplicated data. The first update would use a *store-lock* operation which would prevent interrupts from occurring and the second update would use a *store-unlock* operation to enable interrupts again. Additionally, the interrupt handling routine would need to know that there are two copies of the data to update.

4. Performance and Cost Evaluation

To evaluate the effectiveness of our algorithms, we compiled a suite of DSP programs, executed the generated code on the instruction-set simulator of our model DSP architecture, and measured the performance gain over code that allocates all data to one memory bank. We also investigated the additional storage requirements when using duplication to determine whether this technique would be cost-effective.

The benchmark suite we used consists of two types of programs:

Kernel	Description
fft_1024	Radix-2, in-place, decimation-in-time fast Fourier transform (FFT)
fft_256	
fir_256_64	Finite Impulse Response (FIR) filter
fir_32_1	
iir_4_64	Infinite Impulse Response (IIR) filter
iir_1_1	
latnrm_32_64	Normalized lattice filter
latnrm_8_1	
lmsfir_32_64	Least-mean-squared (LMS) adaptive FIR filter
lmsfir_8_1	
mult_10_10	Matrix multiplication
mult_4_4	

Table 1: DSP Kernel Benchmarks

Application	Description
adpcm	Adaptive, Differential, Pulse-Code Modulation speech encoder
lpc	Linear Predictive Coding speech encoder
spectral	Spectral analysis using periodogram averaging
edge_detect	Edge detection using 2D convolution and Sobel operators
compress	Image compression using Discrete Cosine Transform
histogram	Image enhancement using histogram equalization
V32encode	V.32 modem encoder
G721MLencode	Various implementations of CCITT G.721 ADPCM speech encoder
G721MLdecode	
G721WFencode	
trellis	Trellis decoder

Table 2: DSP Application Benchmarks

kernels and applications. These are listed in Tables 1 and 2. In the embedded community, DSPs are typically evaluated using loops that form the core of many common signal processing algorithms. Fast execution of these loops is important particularly in real-time environments. Hence we need to demonstrate that our compiler algorithms are able to generate fast code for such loops. However, as noted with benchmarking efforts in the general-purpose community, loops do not accurately characterize the execution of full programs [10]. Not surprisingly, the same is true for embedded applications [27]. Hence, our benchmark suite also includes complete programs that better represent typical embedded applications.

The kernel benchmarks are based on six algorithms commonly used in embedded applications. For each algorithm, we implemented two kernels to operate on large-size and small-size data respectively. For example, **fir_256_64** is a 256-tap FIR filter processing 64 samples, and **fir_32_1** is a 32-tap FIR filter processing a single sample. The application benchmarks consist of eleven applications commonly used in the areas of speech processing, image processing, and data communication.

4.1. Performance Results

Figure 7 shows the performance results for the kernels while Figure 8 shows the results for the applications. The results are given as the percentage increase in performance over the case that uses neither partitioning nor duplication. To produce the performance data for this latter case, we compiled the programs with the data allocation pass disabled but with all other optimizations enabled. Thus data is stored in only one memory bank. Performance is measured by the number of cycles executed.

The Ideal data is the performance gain if a dual-ported memory were used. Because accesses can occur in parallel without being constrained by the placement of data in memory, such a memory will provide the best possible performance, albeit at the expense of using dual-ported memory cells. A goal of our compiler research is to achieve Ideal performance by using only single-ported memory banks.

When only compaction-based partitioning and no duplication is used (labeled CB in Figures 7 and 8), Ideal performance is obtained for all but one of the kernels. Even for the one kernel **iir_4_64** (k5 in Figure 7), using CB partitioning provides a 31% performance improvement, which is only three percentage points less than the gain achieved with dual-ported memory. Note that partitioning improves performance for all the kernels. Performance gains range from 13% to 49% with an average of 29%.

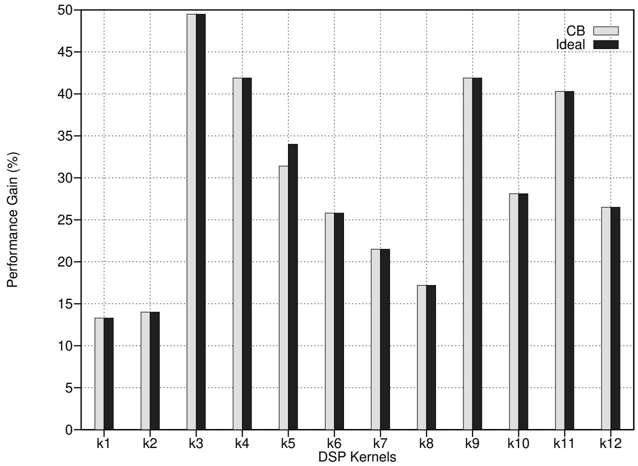
In contrast, the performance gains for the applications are less pronounced even when using an Ideal memory system. The reason for this is because the kernels consist of loops with large amounts of parallelism and several memory operations. Exploiting load/store parallelism inside these loops has a significant impact on overall performance. On the other hand, applications also consist of other sections of code, such as control code or the intervening code between loops. These sections of code generally contain less parallelism but also contain many memory operations. Exploiting memory parallelism in these sections of code has little impact on performance.

The complete absence of memory parallelism is evident in the four applications – **histogram** (a6 in Figure 8) and the three **G721** programs (a8, a9, and a10) – that do not benefit at all from a dual-ported memory; hence any partitioning algorithm is also unlikely to improve the performance of these programs. Thus, other means must first be applied to expose more memory parallelism in these programs.

Nonetheless, applications that do have some amounts of memory parallelism do benefit from using CB partitioning. When performance gains are possible (i.e., Ideal improvement is greater than 0%), the improvement in performance when using CB partitioning ranges from 3% to 15%, with an average of 8% when excluding those programs whose performance cannot be improved (average of 5% over all applications).

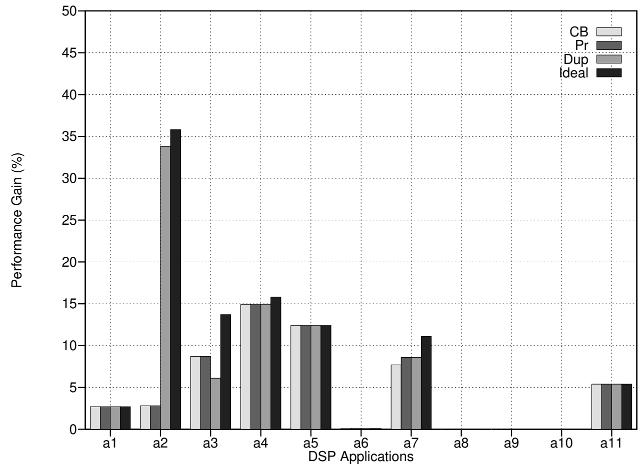
In comparison, Ideal improvement for the applications ranges from 3% to 36% with an average of 14% (9% over all applications), indicating that CB partitioning is missing significant amounts of parallelism in some applications. This is most evident in the **lpc** (a2) and **spectral** (a3) applications, where Ideal shows gains of 36% and 14% versus 3% and 9% respectively for the CB algorithm.

Comparing the code generated for the Ideal case and when using the CB algorithm, we found that the higher Ideal performance was due to exploiting memory parallelism in loops whose lengths are unknown at compile-time. As we mentioned earlier, we hypothesized that CB partitioning was not giving the variables in these loops high enough priority due to the simple heuristic of



k1 = fft_1024 k2 = fft_256 k3 = fir_256_64 k4 = fir_32_1
k5 = iir_4_64 k6 = iir_1_1 k7 = latnrm_32_64 k8 = latnrm_8_1
k9 = lmsfir_32_64 k10 = lmsfir_8_1 k11 = mult_10_10 k12 = mult_4_4

Figure 7: Performance Gain for DSP Kernels



a1 = adpcm a2 = lpc a3 = spectral a4 = edge_detect
a5 = compress a6 = histogram a7 = V32encode a8= G721MLencode
a9 = G721MLdecode a10 = G721WFencode a11 = trellis

Figure 8: Performance Gain for DSP Applications

using loop nesting depth for the edge weights in the interference graph. This led to using profiling to determine the edge weights more accurately. However, the results (labeled Pr in Figure 8) were not as expected.

The use of profile-driven edge weights resulted in different data partitionings for only a few benchmarks. In these benchmarks, we observed slight changes in the code schedules due to the new partitioning. These changes, however, resulted in performance improvements comparable to those of the original CB partitioning, thus suggesting that our heuristic edge weights are satisfactory for our benchmarks and that there is no need for profiling to build the interference graph. It also indicates that the inability of CB partitioning to exploit memory parallelism in some loops is due to other factors.

Upon further inspection, it became clear that the performance degradations were due to simultaneous accesses to the same array, which led to the addition of partial data duplication. Partial duplication was used for three applications – **`lpc`**, **`spectral`**, and **`V32encode`** – since these were the only ones that contained simultaneous accesses to the same array. As Figure 8 shows, partial duplication (labeled Dup) improves the performance of **`lpc`** (a2) by 34% in comparison to a mere 3% gain with CB partitioning and is close to the Ideal performance improvement. On the other hand, partial duplication is marginally better than CB partitioning in improving the performance of **`V32encode`** (a7) and produces a smaller performance gain for **`spectral`** (a3). For these two applications, the execution of the additional store and stack operations required to maintain the integrity of the duplicated data offsets the performance that would otherwise be gained from parallel memory accesses.

4.2. Performance/Cost Trade-Offs of Data Duplication

As mentioned earlier, performance gains from duplication incur cost in the form of increased memory requirements for storing the extra copy of data as well as for storing additional operations. To assess the impact of data duplication on system costs, we developed a first-order cost model that equates cost to memory requirements:

$$Cost = X + Y + 2 \times S + I$$

In this equation, *X*, *Y*, *S*, and *I* represent the memory sizes, in words, of X memory, Y memory, the stack, and the instruction-memory bank respectively. The stack size, *S*, is multiplied by two since we use both data-memory banks. For the results shown here we assumed that instructions are the same size as data. In general, the data costs are more important so any differences between data and instruction sizes will only have minor effects on the results.

We compare the memory cost of using duplication to that of using CB partitioning, the Ideal case, and full duplication. Even for the Ideal case, we only use storage requirement as the measure of cost. In reality this would not be true. Recall that the Ideal case corresponds to using a dual-ported memory bank, which does not require additional storage but instead increases cost in terms of larger chip area, higher power consumption, and possibly longer access times. Hence the Ideal case provides a performance/cost trade-off that is better than what dual-ported memory provides. We include full duplication in the comparison to demonstrate the significant cost savings of duplicating only those arrays that would result in a performance gain when used in conjunction with partitioning.

Table 3 shows the Performance Gain (PG) of each technique relative to the unoptimized case. Interestingly, full duplication does not always improve performance as much as the other techniques do. There is at least one application for which another technique performs at least as well if not better than full duplication. This is because the additional bookkeeping operations offset the performance gains of duplicating all data. Consequently, the average performance gain due to full duplication is less than the averages for partial duplication and the Ideal case.

Table 3 also shows the Cost Increase (CI) of each technique, due to changes in storage requirements, relative to the case when no memory parallelism is exploited. Here, it is important to note that changes in storage requirements include the effects on both instruction and data memories.

Not surprisingly, full duplication incurs a large increase in cost: on average 62% more memory is needed. In contrast, the average increase in cost when using partial duplication with partitioning is only 1%, demonstrating the importance of using partitioning to keep cost increases minimal while improving performance. In some cases, the cost difference is actually a *decrease* in cost. This is because parallel memory accesses are packed into fewer instructions.

Finally, Table 3 lists the Performance/Cost Ratio (PCR) of each technique. PCR is the ratio of the Performance Gain to the Cost

Increase and indicates the goodness of the performance/cost trade-off of each technique. A Performance/Cost Ratio greater than 1 indicates that the improvement in performance outweighs the increase in cost. When using full duplication, the increase in memory requirements is always greater than any gains in performance. When compared to partial duplication or CB partitioning, it is clear that full duplication is never cost-effective.

For techniques other than full duplication, PCR is greater than 1 for all applications. However, this does not always imply that the gain in performance is achieved cost-effectively. When using partial duplication or CB partitioning, the PCR for those applications that do not require duplication is equal to the PCR of the Ideal case indicating that for these applications the performance gain is cost-effective. For those applications requiring duplication, the trade-off is not so consistent. For **lpc**, the PCR when using duplication (1.20) is much greater than the PCR when using CB partitioning only (1.04), suggesting that applying partial duplication is advantageous. For **V32encode**, the two PCRs are comparable (1.10 and 1.09) and the relative importance of performance and cost would need to be used to dictate whether duplicating is worth the additional storage requirements. Finally for **spectral**, partial duplication's PCR (1.01) is less than CB partitioning's PCR (1.11) indicating that duplication is not cost-effective for this application.

While PCR could be used as a compile-time metric for determining the cost-effectiveness of data duplication, more information is required from the DSP application designer to determine the acceptable trade-offs. Depending upon the real-time performance constraints and area budget for the system, a high PCR or a low PCR may be acceptable. By considering maximum execution times and storage capacities provided by the designer, the compiler could be more selective in duplicating data to minimize storage while meeting the performance requirements. Note that profiling information would be needed to estimate performance at compile-time, while determining the cost of duplication is straightforward because the compiler can easily keep track of the additional memory needed to store duplicated data and extra instructions.

5. Summary

Digital signal processors (DSPs) have become significant for their use in embedded applications where high performance, low cost, and high-volume are important criteria. One feature commonly found in DSPs is the use of dual data-memory banks to double the bandwidth of the memory system. To use dual memory banks effectively, most programmers resort to assembly language programming or use high-level languages that have special features to direct the use of memory by the compiler. In this paper, we have demonstrated that it is possible to use a standard programming language, such as C, and effectively allocate the data to the two memories without adding any pragmas or other features. Two complementary algorithms, called compaction-based (CB) data partitioning and partial data duplication, were presented. They are implemented as part of a data allocation pass in our experimental optimizing C compiler. Although dual-ported memories would obviate the need for these algorithms, the added cost in memory area is not justified, particularly when good performance can be achieved using single-ported, dual memory banks.

Our CB data partitioning algorithm allocates program variables to different memory banks so that as much parallelism, and hence performance, can be obtained. The algorithm is based on partitioning the nodes of an interference graph whose nodes represent variables and whose edges represent potential parallel accesses to pairs of variables. To build the graph, we use an operation-compaction algorithm to suggest pairs of memory operations that may be executed in parallel and use this information to add edges to the graph. We also assign a weight to each edge to represent the degradation in performance if the corresponding variables are not accessed in parallel. A greedy algorithm is used to partition the nodes into two sets, corresponding to the two memory banks.

When simultaneous accesses to the same array occur, data partitioning is not sufficient and it is necessary to duplicate the data in both memory banks to exploit the available parallelism. However, data duplication increases memory requirements, and hence system costs, due to the extra space needed to store duplicated data and the extra operations needed to ensure the integrity of duplicated data. Executing the extra operations may also degrade performance. Thus, partial duplication should only be applied if the performance gain justifies the cost increase. We have reported results where all arrays with simultaneous accesses are duplicated. If the Performance/Cost Ratio is too low, a further refinement is to determine whether some of these arrays do not have to be duplicated because doing so would not significantly affect performance.

Our performance results indicate that CB partitioning is an

Application Benchmarks	Full Duplication			Partial Duplication			CB Partitioning			Ideal Dual-Ported Memory		
	PG	CI	PCR	PG	CI	PCR	PG	CI	PCR	PG	CI	PCR
adpcm	1.03	1.30	0.79	1.03	0.99	1.04	1.03	0.99	1.04	1.03	0.99	1.04
lpc	1.33	1.56	0.85	1.34	1.12	1.20	1.03	0.99	1.04	1.36	0.99	1.38
spectral	1.09	1.28	0.86	1.06	1.05	1.01	1.09	0.98	1.11	1.14	0.98	1.16
edge_detect	1.16	1.98	0.59	1.15	1.00	1.15	1.15	1.00	1.15	1.16	1.00	1.16
compress	1.11	1.93	0.58	1.12	1.00	1.12	1.12	1.00	1.12	1.12	1.00	1.12
histogram	1.00	1.94	0.52	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
V32encode	1.04	1.35	0.77	1.09	0.99	1.10	1.08	0.98	1.09	1.11	0.98	1.13
G721MLencode	1.00	1.70	0.59	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
G721MLdecode	1.00	1.70	0.59	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
G721WFencode	1.00	1.70	0.59	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
trellis	1.05	1.33	0.79	1.05	0.98	1.07	1.05	0.98	1.07	1.05	0.98	1.07
Arithmetic Mean	1.07	1.62	0.68	1.08	1.01	1.06	1.05	0.99	1.06	1.09	0.99	1.10

Table 3: Performance/Cost Trade-Offs of Exploiting Dual Data-Memory Banks

effective technique for exploiting dual memory banks. This is evident in our kernel benchmarks where CB partitioning improves performance by 13%–49%. In addition, we use a dual-ported memory as an ideal reference for how well our algorithms perform. For all kernels, the performance gains are identical, or nearly identical for one kernel, to those attained in the ideal case. CB partitioning also improves the performance of our application benchmarks by 3%–15% while for the ideal case, the performance gain is 3%–36%. For one application, **`lpc`**, CB partitioning improves performance by only 3%. By adding partial data duplication, the performance improvement is 34%, which is much closer to the ideal improvement of 36%.

6. Acknowledgments

We would like to acknowledge the Information Technology Research Centre, a Centre of Excellence supported by Technology Ontario, for funding this research. We would also like to thank the reviewers for their valuable comments.

7. Postscript

A postscript copy of this paper can be downloaded from `http://www.eecg.toronto.edu/~saghir/papers/asplos7.ps`.

8. References

[1] Gianluigi Castelli, Guest Editor's Introduction: "The Seemingly Unlimited Market for Microcontroller-Based Embedded Systems," *IEEE Micro*, Vol. 15, No. 5, pp. 6-8, October, 1995.

[2] Bennett Z. Kobb, "Telecommunications," *IEEE Spectrum*, pp. 30-34, January, 1995.

[3] Edward A. Lee, "Programmable DSP Architectures," *IEEE ASSP Magazine*, Part I: pp.4-19, October, 1988; Part II: pp. 4-14, January, 1989.

[4] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee, "Towards Better DSP Architectures and Compilers," *Proceedings of the International Conference on Signal Processing Applications and Technology*, pp. 658-664, DSP Associates, October, 1994.

[5] Ruby B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, Vol. 15, No. 2, pp. 22-32, April, 1995.

[6] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, G. Zyner, "The Visual Instruction Set (VIS) in UltraSPARC," *Proceedings of Compcon'95*, pp. 462-469, March, 1995.

[7] Upcoming issue of *IEEE Micro* on Media Processing, August, 1996.

[8] Recent IC Announcements, *Microprocessor Report*, p. 27, August 21, 1995.

[9] Most Significant Bits, *Microprocessor Report*, pp. 4-5, July 31, 1995.

[10] John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., 1995.

[11] Linley Gwennap, "Improved Cost Model Puts Pentium at $180," *Microprocessor Report*, pp. 14-15, September 12, 1994.

[12] Jim Turley and Phil Lapsley, "New 56301 DSP Doubles 24-Bit Performance," *Microprocessor Report*, pp. 14-15, December 4, 1995.

[13] Jose Luis Pino, Soonhoi Ha, Edward A. Lee, and Joseph T. Buck, "Software Synthesis for DSP Using Ptolemy," *Journal of VLSI Signal Processing*, Vol. 9, No. 1-2, pp. 7-21, January, 1995.

[14] Vojin Zivojnovic, Harald Schraut, M. Willems, and R. Schoenen, "DSPs, GPPs, and Multimedia Applications - An Evaluation Using DSPstone," *Proceedings of the International Conference on Signal Processing Applications and Technology*, pp. 1779-1783, DSP Associates, October, 1995.

[15] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[16] "MIPS Open RISC Technology R10000 Microprocessor Technical Brief", `http://www.mips.com/r10k/r10000_Pr_Info/R10000_Tech_Br_cv.html`, October, 1994.

[17] *DSP56000/DSP56001 Digital Signal Processor User's Manual*, Motorola, 1990.

[18] Monica S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 318-328, ACM, June, 1988.

[19] Carla Procaskey, "Improving Compiled DSP Code Through Language Extensions," *Proceedings of the International Conference on Signal Processing Applications and Technology*, pp. 846-850, DSP Associates, October, 1995.

[20] Ashok Sudarsanam and Sharad Malik, "Memory Bank and Register Allocation in Software Synthesis for ASIPS," *Proceedings of the International Conference on Computer-Aided Design*, pp. 388-392, IEEE/ACM, 1995.

[21] P. G. Lowney, et. al., "The Multiflow Trace Scheduling Compiler," *Journal of Supercomputing*, Vol. 7, Issue 1-2, pp. 51-142, May, 1993.

[22] "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," `http://suif.stanford.edu/suif/suif-overview/suif.html`, 1994.

[23] Joseph Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proceedings of the 10th International Symposium on Computer Architecture*, pp. 140-150, IEEE, 1983.

[24] Richard M. Stallman, *Using and Porting GNU C*, Free Software Foundation, Inc., 1990.

[25] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett, "Local Microcode Compaction Techniques," *Computing Surveys*, 12(3): pp. 261-294, ACM, September, 1980.

[26] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[27] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee, "Application-Driven Design of DSP Architectures and Compilers," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. II-437-440, IEEE, 1994.