Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs

ASHOK SUDARSANAM and SHARAD MALIK
Princeton University

We address the problem of code generation for DSP systems on a chip. In such systems, the amount of silicon devoted to program ROM is limited, so application software must be sufficiently dense. Additionally, the software must be written so as to meet various high-performance constraints, which may include hard real-time constraints. Unfortunately, current compiler technology is unable to generate high-quality code for DSPs, whose architectures are highly irregular. Thus, designers often resort to programming application software in assembly—a time-consuming task.

In this paper, we focus on providing support for one architectural feature of DSPs that makes code generation difficult, namely multiple data memory banks. This feature increases memory bandwidth by permitting multiple data memory accesses to occur in parallel when the referenced variables belong to different data memory banks and the registers involved conform to a strict set of conditions. We present an algorithm that attempts to maximize the benefit of this architectural feature. While previous approaches have decoupled the phases of register allocation and memory bank assignment, thereby compromising code quality, our algorithm performs these two phases simultaneously. Experimental results demonstrate that our algorithm not only generates high-quality compiled code, but also improves the quality of completely-referenced code.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation*; *Compilers*; *Optimization*

General Terms: Design, Languages

Additional Key Words and Phrases: Code generation, code optimization, graph labelling, memory bank assignment, register allocation

A version of this paper was presented in Sudarsanam and Malik [1995] at the IEEE International Conference on Computer-Aided Design, San Jose, CA, November 5–9, 1995.
Authors' address: Department of Electrical Engineering, Princeton University, Princeton, NJ 08544.

1. INTRODUCTION

It is a well-known fact that the quality of compilers for embedded DSP systems is generally unacceptable with respect to code density and performance—compilation techniques for general-purpose architectures do not adapt well to the irregularity of DSP architectures. Consequently, most application software for such systems is hand-written—a very time-consuming, error-prone, and non-portable task.

Our research aims to overcome the limitations of current compilation techniques for DSPs [Liao et al. 1996]. We would like to identify all architectural characteristics of DSPs that make code generation difficult and provide specific solutions for each of these. We would then like to integrate these solutions so as to provide a high-quality, retargetable code synthesis environment. In this paper, we focus on providing support for one particular architectural feature, namely *multiple data memory banks*. This feature, found in the *Motorola DSP56000* and *NEC 77016* DSPs, increases memory bandwidth by permitting multiple data memory accesses to occur in parallel when the referenced variables belong to different banks and the registers involved conform to a strict set of conditions. Furthermore, the instruction set architectures (ISAs) of these DSPs require the programmer to encode in a limited number of long instruction words all data memory accesses that are to be performed in parallel, thus assisting in the generation of dense code. We will use the DSP56000 [Motorola 1990] as our experimental vehicle in this research.

We present an algorithm that attempts to maximize the benefit of this architectural feature. While previous approaches decoupled the phases of *register allocation* and *memory bank assignment*, thereby compromising code quality, our algorithm performs these two phases simultaneously. Our algorithm is based on *graph labelling*, whose objective is to find an optimal labelling of a *constraint graph* representing conditions on the register and memory bank allocation. Since optimal labelling of this graph is an NP-hard problem, we use the well-known hill-climbing algorithm *simulated annealing* to find a good labelling. Although simulated annealing is computationally expensive, we find its use completely acceptable because of the high quality of results obtained from it.

This paper is organized as follows: Section 2 gives an overview of the DSP56000 architecture; Section 3 describes previous work in this area; Section 4 describes our graph labelling algorithm; Section 5 provides experimental results; finally, we present our conclusions in Section 6.

2. MOTOROLA DSP56000 ARCHITECTURE

The DSP56000 architectural units of interest are the Data Arithmetic/Logic Unit (*Data ALU*), Address Generation Unit (*AGU*), and *X*/*Y data memory banks*:

—Data ALU: This unit, shown in Figure 1, contains hardware specialized for performing fast multiply-accumulate operations. Other arithmetic

and logical operations can also be performed efficiently. The data ALU consists of four 24-bit input registers called *X0*, *X1*, *Y0*, and *Y1*, and two 56-bit accumulators, *A* and *B*. The source operands for all ALU operations must be input registers or accumulators, and the destination operand must always be an accumulator. Two 24-bit buses *XDB* and *YDB* permit two input registers or accumulators to be read or written in conjunction with the execution of an ALU operation. Hence, three operations may be executed simultaneously in one instruction cycle.

—AGU: This unit, shown in Figure 2, contains two sets of 16-bit register files, one consisting of address registers *R0* through *R3* and offset registers *N0* through *N3*, and the other consisting of address registers *R4* through *R7* and offset registers *N4* through *N7*. Associated with each set is an address ALU which, on each cycle, can post-increment or post-decrement a single address register Ri by (i) the constant one or (ii) the contents of the corresponding offset register Ni. Two multiplexers permit two effective addresses to be generated each cycle—at most one address per set may be generated, and these addresses must point to locations in different data memory banks. The source of each effective address may be an address register (for register-indirect addressing) or the output of the address ALU (for indexed addressing). The AGU also features two files of 16-bit modifier registers (not shown) which may be used to perform modulo arithmetic.

—X/Y data memory banks: This unit, also shown in Figure 2, consists of two 512-word x 24-bit data memory banks, which allow a total of two data memory accesses to occur in parallel. When indirect or indexed addressing is used, the effective addresses are generated by the AGU. Alternatively, a 16-bit immediate that is generated as part of the instruction word may be used for absolute addressing.

The DSP56000 ISA assists in the generation of dense, high-bandwidth code by requiring the programmer to encode in either one or two 24-bit instruction words all operations that are to execute in parallel during each instruction cycle. Specifically, up to two *move* operations and one Data ALU operation may be encoded in these words, where a move refers to a memory access (load or store), register transfer (moving of data from an input register to an accumulator, or vice-versa), or immediate load (loading of a 24-bit constant into an input register or accumulator). However, due to the nature of the DSP56000 microarchitecture, only the following pairs of move operations may be performed in parallel:

—two memory accesses may be performed in parallel.

—a memory access and register transfer may be performed in parallel.

—a register transfer and immediate load may be performed in parallel.

Additionally, several Data ALU operations may not be encoded in the same instruction word(s) with move operations. Furthermore, two move

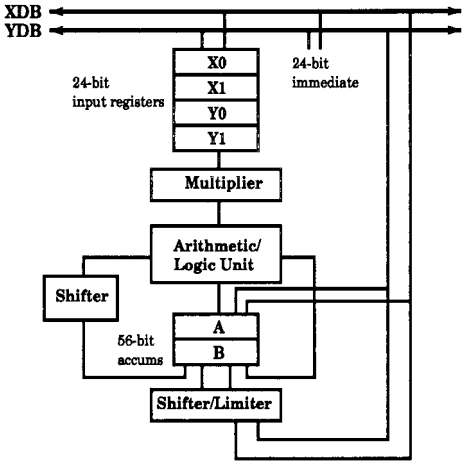

Fig. 1. Data arithmetic/logic unit.

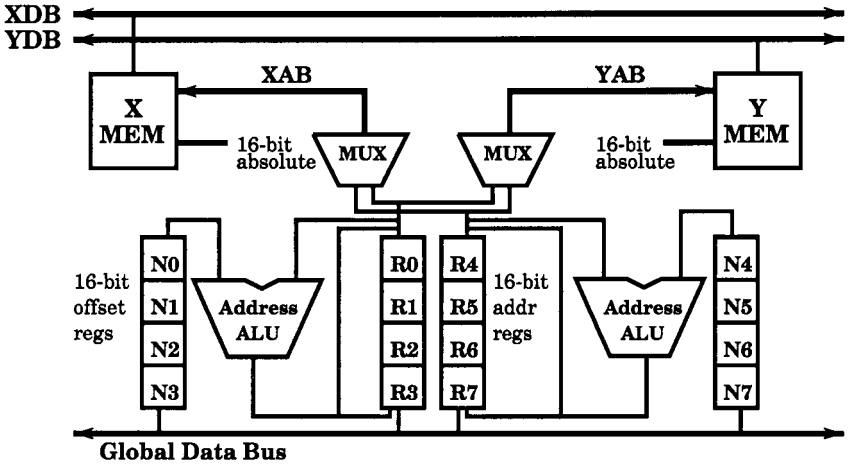

Fig. 2. Address generation unit and data memory banks.

operations may execute simultaneously only if the allocation of the associated registers and memory banks meets a set of requirements. Consider the following parallel move specification which simultaneously (i) loads into X1 a datum from the X memory bank, whose address is stored in R0 and (ii) loads into Y1 a datum from the Y memory bank, whose address is stored in R4:

```
MOVE X:(R0), X1  Y:(R4), Y1
```

This parallel memory access specification is legal because certain register and memory bank allocation constraints have been appropriately satisfied:

—the two operations access data in different data memory banks.

—the destination registers of the two moves are different.

—the X memory access loads into a restricted set of locations: X0, X1, A or B.

—the Y memory access loads into a restricted set of locations: Y0, Y1, A or B.

—both data memory accesses employ register-indirect addressing only, in which the address registers specified belong to different AGU register file sets.

Thus, the following parallel memory access specifications are not permitted:

—MOVE X:(R0), X1 Y:(R4)+, X0
the Y memory access does not load into the allowable set of locations.

—MOVE X:(R0), X1 X:(R1)+, X0
both memory accesses are to the same bank.

—MOVE X:(R0), X1 Y:(R4+N4), Y1
the second memory access uses indexed, rather than register-indirect addressing.

—MOVE X:(R0), X1 Y:(R2), Y1
the specified address registers belong to the same AGU register file set.

Similar allocation restrictions exist for the other parallel move combinations that are permitted by this architecture. Thus, given the nature of the DSP56000 architecture, our intent is to develop an algorithm that takes full advantage of the available parallelism.

3. PREVIOUS WORK

Much work has been done in the area of code generation (instruction selection, register allocation, instruction scheduling) and optimization for DSPs. Code is traditionally generated per basic block, where each basic block is represented as a *directed acyclic graph* (DAG). Optimal code generation for DAGs is a well-known NP-complete problem, and is generally tackled by dismantling the DAG into *expression trees*, thus potentially sacrificing optimality, and then generating code for each tree. In Araujo and Malik [1995], it is shown that optimal instruction selection, register allocation, and instruction scheduling for trees can be performed in linear time for a class of machines that satisfy a certain architectural property—the Texas Instruments TMS320C25 [Texas Instruments 1993], but not the DSP56000, satisfies this property. The basis of this optimal code generation

algorithm is a combination of efficient tree-matching and dynamic programming techniques [Aho et al. 1989; Fraser et al. 1992] that perform optimal instruction selection for expression trees in linear time. Optimal instruction selection for expression trees can also be performed in linear time using *trellis diagrams* [Wess 1992]. Alternatively, instruction selection for expression DAGs may be performed exactly by means of a *binate covering* formulation [Liao et al. 1995].

Various compaction algorithms exist that attempt to exploit available *instruction-level parallelism* (ILP) in DSPs. The algorithm of Leupers and Marwedel [1996] performs local compaction for those architectures in which available ILP is encoded in the opcode itself (e.g., TMS320C25). The method of Kafka [1990] performs global compaction for those microcoded architectures in which available ILP is encoded in horizontal, rather than vertical, instruction formats.

There has also been an increasing amount of work in *retargetable* DSP code generation [Laneer et al. 1995; Paulin et al. 1994; Leupers et al. 1994]. Although these works discuss various code optimizations, the emphasis is on techniques that facilitate retargetability. Our own efforts at developing a retargetable optimizing DSP compiler are described in Liao et al. [1996].

Very few previous works have targeted code generation for the DSP56000, and each of these works has *decoupled* the phases of register allocation and memory bank assignment. In Wess [1991], dynamic programming is used to translate the *signal flow graph* (SFG) representation of the source program into uncompacted code. Only register allocation is performed on this code, so all variable references are symbolic. This code is then compacted into as few instructions as possible so as to not violate any data-dependency or target machine constraints. Finally, the symbolic variables are assigned to memory banks and address registers based on the current register allocation and compaction. If this results in the violation of one or more parallel moves, then various code generation parameters are modified, and the entire process is repeated, beginning with the generation of uncompacted code. This sequence of steps is repeated until no further improvements are possible.

In Powell et al. [1992], each block of the source program SFG maps onto a segment of hand-optimized *meta-assembly* code, in which all register and variable references are symbolic. Register allocation is first performed on each block of code, then variables are assigned to memory banks in an alternating fashion. Finally, the code is compacted into as few instructions as possible without violating any constraints.

We find one major limitation of these two approaches: since the validity of parallel moves is so highly dependent on both the register and memory bank allocation, we believe that these two phases should be performed *simultaneously*. By decoupling these phases, optimality is potentially compromised before code compaction occurs—due to target machine constraints, performing register and memory bank allocation prior to compaction may lead to suboptimal compacted code.

Code generation for multiple memory bank architectures is also addressed in Saghir et al. [1996]. However, a synthetic DSP model featuring a large general-purpose register file is assumed. Consequently, register and memory bank allocation are independent code generation phases for this architecture. A greedy partitioning technique is used to perform memory bank assignment, while conventional graph coloring techniques [Chaitin et al. 1981] are used to perform register allocation.

In the parallel processing community, related work has focused on the distribution of program data among the local memories of the individual processing elements. In particular, a distribution of arrays and structures among the memories is desired such that the total amount of interprocessor communication required is minimized [Anderson and Lam 1993; Li and Chen 1991].

4. TECHNIQUE FOR REGISTER AND MEMORY BANK ALLOCATION

Our algorithm for register and memory bank allocation, or *reference allocation*, is a post-pass optimization that occurs after instruction selection. We assume the code generator generates optimized uncompacted assembly code in which all register and variable references are symbolic (i.e., meta-assembly code). Our objective is to assign the symbolic registers to physical registers and the symbolic variables to data memory banks such that the benefit of dual data memory banks is maximized.

4.0.1 Generation of Uncompacted Code

Given the DAG for a basic block, we first dismantle it into expression trees, transform each of these into an *intermediate representation* (IR), and then translate each IR tree into uncompacted symbolic assembly code using the *Olive* code-generator generator [Liao et al. 1996]. Olive takes as input a grammar-based description of the target machine ISA, then automatically constructs a code generator that performs, in linear time, optimal instruction selection for expression trees. An issue naturally arises as to how the subtrees of an IR tree should be ordered for code generation: Does one ordering result in better code than another? The answer lies in the fact that upon request, a *new* symbolic register is generated to store a variable or constant. Hence, all subtrees of an IR tree are pair-wise data-independent with respect to symbolic registers, since no two define or use the same one. Symbolic variables may only be used within subtrees—they may be defined only at the root of the entire IR tree. Thus, all subtrees are pair-wise data-independent with respect to symbolic variables also.

The *data-dependency graph* (DDG) is the primary data structure used by our compaction algorithm (see Section 4). Due to the manner in which they are constructed, the DDGs representing each possible subtree ordering are isomorphic, since all subtrees are pair-wise data-independent. Hence, the quality of the compacted code is not affected by the order in which subtrees are translated.

As previously mentioned, DAG dismantling generally results in loss of optimality; it is possible that some DSP compiler users may wish to sacrifice low compilation times and generate code directly from DAGs. However, we strongly believe that Olive-based code generation techniques should be employed for two reasons: first, the grammar-based input file to Olive can be easily modified to generate code for a new architecture. We find it acceptable to introduce some code inefficiencies due to dismantling if retargetability is facilitated. Second, it has been shown that for a large sample of DSP benchmarks, the majority of DAGs are in fact trees [Araujo et al. 1996]. This result also favors tree-based code generation.

4.0.2 Reference Allocation Before or After Code Compaction

In Figure 3, we demonstrate why we delay reference allocation until after the symbolic assembly code has been compacted. Consider the C and corresponding uncompacted symbolic assembly code shown in Figure 3(a). Figure 3(b) shows the compacted code that results when reference allocation is performed on this initial code sequence using the following *greedy* approach. The memory bank assigned to a symbolic variable is the opposite of the bank that was assigned to the symbolic variable last referenced. Furthermore, if a load operation is encountered in which the referenced variable has been allocated to the X (Y) bank, then preference is given to X0 and X1 (Y0 and Y1) when allocating the corresponding symbolic register. Once reference allocation has been performed, the uncompacted code is compacted into as few instructions as possible such that no constraints are violated, and address registers are appropriately allocated to variables. For sake of clarity, uses of register-indirect addressing are not shown in this and subsequent figures, but should be assumed. Since variable *f* has been allocated to the X memory bank by the greedy algorithm, it is not possible to execute the move operation *MOVE f,* reg_5 in parallel with other operations. The resulting compacted code has a total length of 6 instruction words.

Figure 3(c) shows the code sequence that results when reference allocation is performed after the symbolic code has been compacted. The delaying of reference allocation until this stage enables the compiler to first analyze all memory accesses that may potentially be performed in parallel, and then generate an allocation that satisfies a maximum number of these parallel accesses. As one can see, the allocation of variable *f* to the Y bank and the allocation of symbolic register reg_5 to B permits the operation *MOVE f,* reg_5 to execute in parallel with two other operations. The length of the resulting compacted code is now only 5 words.

By intelligently performing reference allocation and instruction selection concurrently, a code generator could produce optimal uncompacted code. For example, by realizing that a required variable *x* already resides in an input register, the code generator could suppress generation of an unnecessary instruction to reload *x*. However, as demonstrated in Figure 3(b), it appears unlikely that even a very clever code generator could produce

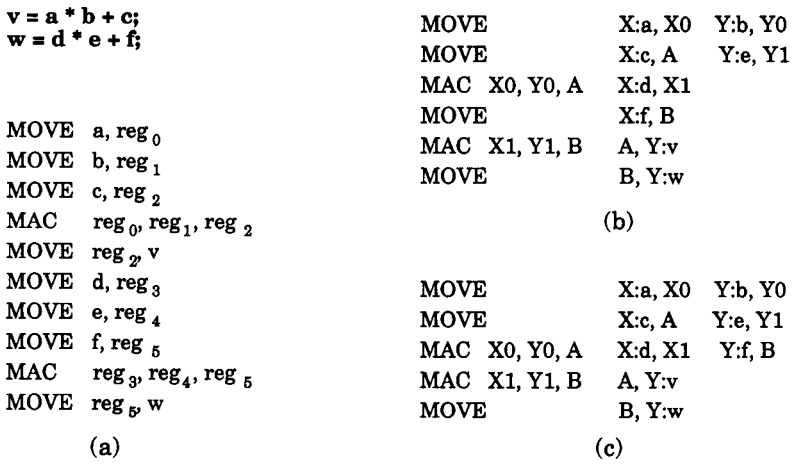

Fig. 3. (a) C and uncompacted assembly code; (b) greedy allocation; (c) delayed allocation.

optimal compacted code, since no prior knowledge of parallel move constraints exists while instruction selection is being performed.

4.0.3 Simultaneous Versus Decoupled Reference Allocation

Figure 4 shows why it is advantageous to perform the phases of reference allocation simultaneously. Consider the C and corresponding uncompacted symbolic assembly code shown in Figure 4(a). The compacted code that results when reference allocation is decoupled using the approach described in Powell et al. [1992] is shown in Figure 4(b). Since this approach allocated variable d to the Y bank and symbolic register reg_2 to X0, the operation *MOVE d,* reg_2 may not execute in parallel with other operations. The resulting compacted code has a total length of 5 words.

The code that results when reference allocation is performed simultaneously after compaction is shown in Figure 4(c). The simultaneous execution of the reference allocation phases enables the compiler to first analyze all interactions between symbolic registers and variables, and subsequently generate an intelligent allocation. Since variable d and symbolic register reg_2 have been allocated to the Y bank and Y0, respectively, the operation *MOVE d,* reg_2 is now able to execute in parallel with two other operations. The length of the resulting code is now only 4 words.

4.1 Assumptions in Reference Allocation

The following sections describe the key assumptions that we make in our approach:

4.1.1 *Allocation of Global Variables*. To determine the optimal memory bank assignment for a given global variable, the compiler would need to observe its references over all procedures in the program. The fact that

```
c = a + b;
f = d + e;

MOVE  a, reg0
MOVE  b, reg1
ADD   reg0, reg1
MOVE  reg1, c
MOVE  d, reg2
MOVE  e, reg3
ADD   reg2, reg3
MOVE  reg3, f
```

(a)

```
MOVE        X:a, X1   Y:b, A
ADD  X1, A  X:e, B
MOVE                  Y:d, X0
ADD  X0, B  A, X:c
MOVE                  B, Y:f
```

(b)

```
MOVE        X:a, X0   Y:b, A
ADD  X0, A  X:e, B    Y:d, Y0
ADD  Y0, B  A, X:c
MOVE                  B, Y:f
```

(c)

Fig. 4. (a) C and uncompacted assembly code; (b) decoupled allocation; (c) simultaneous allocation.

code is normally generated on a per-procedure basis makes this a non-trivial task. Hence, we assume that all global variables are statically allocated in the X data memory bank and absolutely addressed. By performing interprocedural data-flow analysis [Aho et al. 1986], the compiler could determine this optimal memory bank assignment.

4.1.2 *Allocation of Local Variables*. In the C programming language, local variables may be classified as being either *automatic* or *static*. Automatic variables are variables that come into existence on procedure entry and disappear on procedure exit. Static variables are local variables that remain in existence throughout program execution. Static variables are treated similarly to global variables—they are statically allocated and absolutely addressed. However, we do not restrict static variables to be allocated to the X bank, since the compiler can determine the optimal memory bank assignment for them by performing an intraprocedural analysis.

Meanwhile, we assume that automatic variables are dynamically allocated on a stack frame. We also assume that the stack pointer and frame pointer point to corresponding locations in the two data memory banks, so that at any given point in time, two stack frames of identical size are active. For the sake of simplicity, we assume that parameters are pushed onto the stack frame of the X bank only. By analyzing the reference patterns of the parameters in the callee procedure, the compiler could optimally distribute the parameters across both frames. However, this would again require the non-trivial task of interprocedural analysis. In Section 4, we will discuss in detail the allocation of non-parameter automatic variables.

Given the assumptions surrounding our technique, we now give a detailed account of each phase of our algorithm. We begin by describing the local compaction phase.

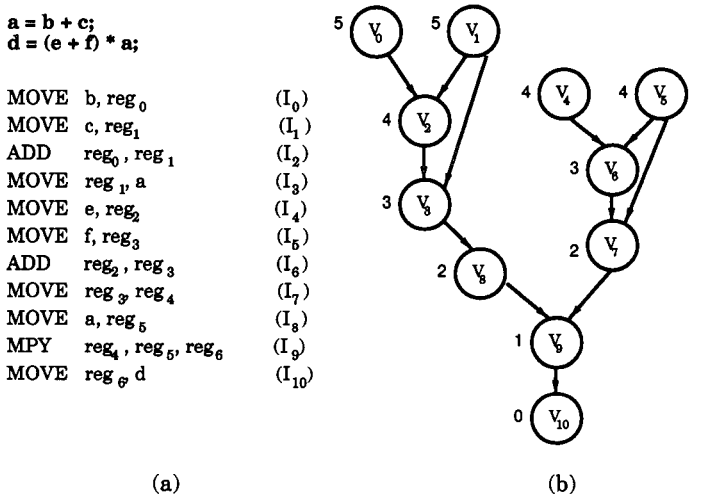

Fig. 5. (a) Original C and symbolic assembly code; (b) data dependency graph.

4.2 Step I: Local Compaction

The first phase of our algorithm compacts the symbolic uncompacted code. Two move operations are scheduled into one or two long instruction words if no data dependencies are violated and the parallel move combination is permitted by the ISA. Additionally, an ALU operation O is scheduled with up to two move operations if no data dependencies are violated and it is permissible for O to be encoded in the same instruction word(s) with moves. We use the *list scheduling* algorithm [Landskov et al. 1980] to perform code compaction within basic blocks.

The first step in the list scheduling algorithm is to construct a *data-dependency graph* (DDG) for each basic block. Each node V_x of this directed graph corresponds to instruction I_x in the uncompacted code, and edge $e\langle V_i, V_j\rangle$ exists if I_j has a data dependency on I_i. This edge specifies that in the final compacted code, I_j must not be scheduled before I_i; otherwise, program semantics may be adversely affected. Consider the C code sequence and corresponding uncompacted assembly code shown in Figure 5(a). The DDG for this code sequence is shown in Figure 5(b). The number to the left of each node represents the *priority* of the node—this attribute is required since nodes are scheduled in order of decreasing priority. The priority function that has been implemented is distance-from-sink. By definition, the sink node has a priority of zero, which implies that it will be scheduled last.

Finally, the nodes of the DDG are scheduled as follows: until all nodes have been scheduled, schedule into the current instruction word the node whose priority is maximum and whose predecessors have already been scheduled. Let us define an unscheduled DDG node to be *ready* if all of its predecessors have been scheduled.

```
MOVE                          b, reg 0    c, reg 1
ADD    reg 0, reg 1           e, reg 2    f, reg 3
ADD    reg 2, reg 3           reg 1, a
MOVE                          a, reg 5    reg 3, reg 4
MPY    reg 4, reg 5, reg 6
MOVE                          reg 6 , d
```

Fig. 6. Compacted code generated by list scheduling..

Since register allocation has yet to be performed on the symbolic code, we must exercise caution during compaction. We say a symbolic register R is *live* at a given point in the program if it is subsequently used before being redefined. The set of symbolic registers that are live at each point in the program can be computed using the *live-variable analysis* data-flow equations [Aho et al. 1986]. Two symbolic registers that are simultaneously live must be assigned to different physical registers, otherwise *spill code* must be generated to save one of these values to memory after each definition and to retrieve it from memory before each use. If the high-priority DDG nodes correspond to operations that define symbolic registers, then it is highly probable that at some point(s) in the compacted code, the number of symbolic registers that are simultaneously live will exceed the number of available physical registers. In this scenario, spill code must inevitably be generated, hence, we have developed some scheduling heuristics that attempt to reduce the register pressure at each point in the compacted code. This may reduce the amount of spill code that must be generated. Our modified list scheduling algorithm schedules nodes in the following order:

—a ready node whose corresponding operation contains one or more symbolic register source operands. By scheduling such a node, we can only decrease the number of points in the program at which these symbolic registers are live. This possibly decreases register pressure and reduces the amount of spill code required.

—a ready node of highest priority which, if scheduled, causes a node that meets the above criteria for becoming ready.

—a ready node of highest priority.

Figure 6 shows the final compacted schedule that results when DDG nodes are scheduled using our modified list scheduling algorithm. We also perform a final list scheduling pass after reference allocation has been completed. Although certain parallel move operations in the symbolic compacted code may not be satisfied by the reference allocation, other opportunities for compaction may arise. This final pass attempts to exploit these opportunities.

4.3 Step II: Constraint Graph Creation

The process of performing reference allocation may be viewed as the process of finding an appropriate *labelling* of an undirected *constraint*

graph. For each symbolic register reg_x in the compacted code, there exists a constraint graph vertex R_x. Similarly, for each symbolic variable var_y in the code, there exists a constraint graph vertex V_y. Each symbolic register vertex R_i must be labelled X0, X1, Y0, Y1, A, or B, while each symbolic variable vertex V_j must be labelled either X or Y. Also contained in a constraint graph are different classes of edges, which represent constraints on the labelling of vertices. Associated with each edge is a *cost*, or penalty, that is incurred when the constraints represented by the edge are not satisfied. A constraint graph is constructed on a per-procedure basis. This construction proceeds with the insertion of red edges, which are now described.

4.3.1 *Red Edges*. A *red edge* is inserted between vertices R_x and R_y if reg_x and reg_y are simultaneously live. This edge specifies that R_x and R_y must be labelled differently, or analogously, reg_x and reg_y must be assigned to different physical registers. Otherwise, a definition of reg_x will overwrite the value of reg_y, or vice versa, in which case spill code must be generated.

The amount of spill code required due to an unsatisfied red edge is directly proportional to the number of times the spilled symbolic register is defined and used. This number represents the cost, or penalty, of the corresponding red edge.

For each compacted instruction containing two move operations, we add a non-red edge to the constraint graph that specifies how the associated symbolic register and variable nodes should be labelled in order for the parallel move to be legal.

4.3.2 *Green Edges*. A *green edge* is inserted into the constraint graph for each parallel move corresponding to a dual memory access. Assume that symbolic register reg_i and symbolic variable var_i are involved in one memory access, while symbolic register reg_j and symbolic variable var_j are involved in the other. We then insert a green edge between vertices R_i and R_j. This green edge includes *pointers* to vertices V_i and V_j. These pointers specify that reg_i and var_i constitute one move operation, while reg_j and var_j constitute the other.

Several constraints must be satisfied in order for this green edge to be labelled appropriately and hence, for the parallel memory access to be legal:

—V_i and V_j must be labelled differently.

—if V_i is labelled X (Y), then R_i must be labelled X0, X1, A or B (Y0, Y1, A, B).

—if V_j is labelled X (Y), then R_j must be labelled X0, X1, A or B (Y0, Y1, A, B).

If the current procedure consists of a single basic block, then each unsatisfied green edge incurs a cost of *one*. This represents the resulting increase

in instruction words and cycles: a parallel move corresponding to an unsatisfied green edge must be decomposed into two individual move operations. If the current procedure is composed of multiple basic blocks, then each unsatisfied green edge incurs a cost of *basic-block-frequency*, where basic-block-frequency is an estimate of the number of times the corresponding basic block is executed per procedure invocation—this information is obtained from profiling analysis.

Since we perform a post-reference allocation compaction pass, the penalties for unsatisfied green edges are actually conservative estimates. Suppose green edges e_1 and e_2 are both unsatisfied after reference allocation. It is possible that this final pass may compact a move operation of e_1 with one of e_2. Thus, the cost contributed by these edges may be less than the original cost of *2*basic-block-frequency*.

4.3.3 *Blue, Brown, and Yellow Edges*. In a similar manner, we insert other edges into the constraint graph based on the type of parallel move operation encountered in the compacted code. Each of these edges has a set of constraints that must be satisfied when labelling the graph, plus an associated cost.

—*Blue edges:* inserted for each memory load and register transfer.

—*Brown edges:* inserted for each immediate load and register transfer.

—*Yellow edges:* inserted for each memory store and register transfer.

4.3.4 *Black Edges*. Each ALU operation in the DSP56000 ISA imposes certain constraints on its operands. We already know that the source operands of all ALU operations must be input registers or accumulators, and the destination operand must always be an accumulator. However, in most cases, the constraints are even more restrictive. Consider the multiplication operation:

$$\text{MPY } reg_i, reg_j, reg_k$$

The DSP56000 ISA restricts symbolic registers reg_i and reg_j to be input registers only. Symbolic register reg_k must be an accumulator. We enforce these restrictions in our constraint graph by introducing *black edges*, and *data-path vertices* corresponding to the input registers and accumulators in the Data ALU (see Section 2). A black edge between register vertex R_x and data-path vertex DP_y implies that R_x *cannot* be labelled with the physical register associated with DP_y . To enforce the restriction that reg_i must be an input register, we add black edges between R_i and the data-path vertices corresponding to the two accumulators. Figure 7 shows the complete set of black edges required for this operation.

Each unsatisfied black edge incurs a cost of ∞, since the register allocation implied by an illegally-labelled black edge cannot be supported by ALU hardware.

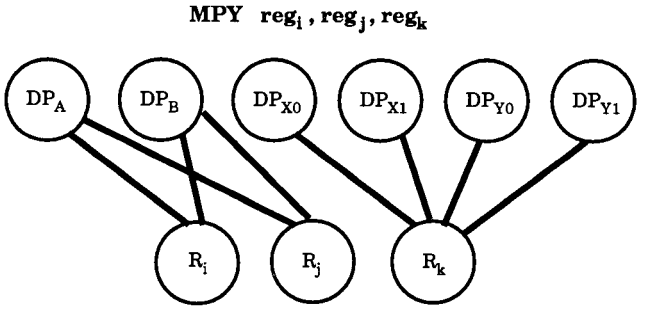

Fig. 7. Black edge construction for multiply operation.

4.3.5 *Address Register Allocation*. Given a memory bank allocation, we must now determine the form of effective addressing used to access the automatic local variables. Each compacted instruction containing two memory references must use register-indirect addressing only. Compacted instructions containing a single memory reference may use indirect, absolute, or indexed addressing. Each use of absolute addressing incurs a one word penalty, however, since an extra word is required to store the 16-bit address. Each use of indexed addressing may possibly require an additional instruction to initialize the corresponding offset register. Since one of our objectives is to generate high-density code, we choose to use register-indirect addressing exclusively. Thus, all automatic variables are allocated on a stack frame and accessed indirectly at an offset from the frame pointer. We have allocated address registers R0 and R4 to be the frame and stack pointers, respectively.

Now, each memory access may be performed only if an address register is available that points to the correct memory location. Recall that associated with the DSP56000 AGU are post-increment/decrement-by-one update modes which may be used to efficiently move address registers between adjacent storage locations. Rather than allocating an address register and performing *address arithmetic* prior to each memory access, an intelligent placement of variables within the stack frames of the two memory banks could make use of this *auto-increment arithmetic* and thus, minimize the number of required address arithmetic instructions. This is precisely what the *Offset Assignment Problem* [Liao et al. 1995] (OA) attempts to perform.

The idea behind OA is that two automatic variables that are accessed together frequently should be assigned to *adjacent* locations on the stack frame. Consequently, an address register may move between these two locations for free using auto-increment arithmetic. However, this may not be possible for all pairs of variables. Hence, an attempt is made to assign variables to stack locations in such a way that the total cost of the assignment is minimized, where an assignment cost is equal to the number of consecutive accesses of variables that are not assigned to adjacent stack locations. In the event that consecutive accesses are to be made to non-adjacent variables, address arithmetic instructions must be generated to

set an address register to point to the correct location. In the DSP56000, this is accomplished by initializing an offset register and adding it to the corresponding address register. Due to pipelining restrictions, a two-cycle delay is required between the definition and use of an offset register. Thus, following the post-reference allocation compaction pass, it may be necessary to insert *no-op* instructions into the compacted assembly code so as to satisfy these restrictions.

Various OA heuristics exist that allow variables on a stack frame to be accessed using multiple address registers. Since parallel memory accesses in the DSP56000 must employ address registers from different AGU file sets, we have arbitrarily allocated registers R1 through R3 to the automatic variables of the X-memory stack frame, and registers R5 through R7 to those of the Y-memory stack frame. Offset assignment currently supports scalar variables only. In Araujo et al. [1996], an algorithm is presented that describes how auto-increment arithmetic may also be used to efficiently access arrays.

4.4 Step III: Constraint Graph Labelling

The cost of a particular constraint graph labelling is equal to the cost due to unsatisfied edges, plus the cost due to OA. A labelling of least cost is desired, since such a labelling corresponds to a reference allocation that maximally satisfies the constraints imposed by the compacted code. However, determining the optimal labelling of a constraint graph is an NP-hard problem, because it subsumes three other problems that are either NP-complete or NP-hard [Garey and Johnson 1979]:

—*Graph K-Colorability Problem*: this problem arises in optimal register allocation for architectures that feature a homogeneous register set [Chaitin et al. 1981].

—*Maximum-Weighted Path Covering Problem*: this problem arises in optimal address register allocation for local variable accesses [Liao et al. 1995].

—*Maximum Bipartite Subgraph Problem*: this problem arises in optimal vanilla memory bank allocation (i.e., no interactions exist between register and memory bank allocation) [Garey and Johnson 1979].

Thus, a heuristic is required that will generate a low-cost labelling of the constraint graph. Several observations can be made about constraint graph labelling:

—*Complex cost function*: the cost function is difficult to optimize.

—*Large solution space with hills and valleys*: given m symbolic register and n symbolic variable vertices, the total number of possible labellings is $6^m * 2^n$, since each symbolic register vertex may be labelled in one of six ways and each symbolic variable vertex may be labelled in one of two ways. By changing the label of a single vertex, one can cause the cost

function to change drastically in either direction, leading to many hills and valleys in the solution space.

—*Easy to determine solution cost*: each edge is examined to determine whether or not its constraints have been satisfied and the cost is updated appropriately. Offset assignment heuristics are applied so as to determine the additional cost of allocating address registers to the automatic variables. With an intelligent choice of data structures, computing the cost of a solution can be done efficiently.

—*Easy to generate new solution*: the label of a randomly-chosen vertex is changed.

These observations suggest that finding a low-cost labelling of the constraint graph is especially well-suited for *simulated annealing* [Kirkpatrick et al. 1983]. Simulated annealing is a probabilistic hill-climbing algorithm that has been used successfully in several design automation and combinatorial optimization problems. As our results demonstrate in the next section, simulated annealing does an excellent job of finding a low-cost labelling of the constraint graph.

5. EXPERIMENTAL RESULTS

5.1 Improving the Quality of Compiled Code

We have implemented our code generation algorithm in the *SPAM compiler*, which is a retargetable code generation framework for embedded DSP processors (Note: the SPAM compiler may be downloaded from *http://www.ee.princeton.edu/spam*). For purposes of experimentation, we selected a set of ANSI C benchmarks from the *DSPstone* benchmark suite [Zivojnović et al. 1994]. The first set of benchmarks (*adapt_quant*, *adpt_predict_1*, *iadpt_quant*, *scale_factor_1*, *speed_control_2*, and *tone_detector_1*) are procedures belonging to the *adpcm* application, which is a large speech-encoding algorithm. The second set of benchmarks—*complex_multiply*, *convolution*, *fir*, *iir_biquad*, *least mean square*, and *matrix_multiply*—are small kernels representative of typical DSP algorithms.

All experiments were performed on a Sun Microsystems Ultra Enterprise featuring eight UltraSPARC processors and 1016 MB of RAM. In the results that follow, it should be noted that our algorithm was executed on only one of these processors. Table I conveys the constraint graph sizes for each of the benchmarks. The first column lists the benchmarks; the next two columns specify the number of symbolic register and variable nodes, respectively; finally, the last six columns specify the number of red, green, blue, brown, yellow, and black edges, respectively.

In general, the constraint graphs for the *adpcm* benchmarks were much larger than those for the kernel benchmarks. The largest constraint graph belonged to *speed_control_2* and was composed of 130 vertices and 338 edges. In comparison, the largest constraint graph for the kernel benchmarks belonged to *iir_biquad* and was composed of 69 vertices and 243

Table I. Constraint Graph Attributes for DSPstone Benchmarks

	Vertices		Edges					
DSPstone Benchmark	Reg	Var	R	G	B1	Br	Y	Bk
adapt_quant	89	16	45	5	0	0	0	166
adpt_predict_1	92	23	66	13	1	0	0	197
iadpt_quant	30	7	24	3	0	0	0	66
scale_factor_1	27	7	24	2	0	0	0	62
speed_control_2	109	21	108	10	0	0	0	220
tone_detector_1	29	8	22	3	0	0	0	57
complex_multiply	10	6	14	0	0	0	0	28
convolution	19	8	18	5	0	0	0	39
fir	75	22	63	13	0	0	0	132
iir_biquad	53	16	120	13	0	0	0	110
least_mean_square	51	18	78	17	0	0	0	106
matrix_multiply	36	12	30	5	0	0	0	68

edges. Large constraint graph sizes directly translated into long simulated annealing times, as the next table will show.

The impact of simultaneous reference allocation on code size is shown in Table II: the benchmarks are listed in the first column; the second column specifies the size (in instruction words) of the initial uncompacted assembly code that was generated by the code generator; the third column specifies the size (in instruction words) of the code that resulted from the application of simultaneous reference allocation to the initial code; the fourth column specifies the percentage improvement of each number in the third column over the corresponding number in the second column; the fifth column specifies the time (in seconds) that was required to perform simulated annealing on the corresponding constraint graph; the final column specifies the size (in instruction words) of the corresponding hand-written reference code that was provided with DSPstone—no reference code was available for *adpcm*. A fixed set of parameters was employed by the simulated annealing algorithm: first, the temperature, which was initialized to 2000 degrees, was decreased by 10% each iteration of the outer-most loop; second, a total of vertex-count*80 moves were generated and evaluated at each temperature; third, the outer-most loop was terminated when the solution cost did not change for three consecutive iterations.

It is apparent that the simulated annealing times were much higher for those benchmarks that had large constraint graphs than for those that had smaller ones. For instance, the benchmarks with the two largest constraint graphs, *adpt_predict_1* and *speed_control_2*, required 8,105 and 5,217 seconds, respectively. The benchmarks with the two smallest constraint graphs, *complex_multiply* and *convolution*, required 2 and 308 seconds, respectively. Although some of these times are quite high, we find the use of simulated annealing completely acceptable because of our desire to synthesize code of the highest quality.

The third and sixth columns of Table II show that there is a significant discrepancy in the quality of code generated by simulated annealing and by

Table II. Impact of Simultaneous Reference Allocation on Code Size

DSPstone Benchmark	Size (orig)	Size (sa)	Code Size Improv	Time (sec)	Size (hand)
adapt_quant	235	16227	453.4%	54399	-
adpt_predict_1	231	209	9.5%	138105	-
iadpt_quant	85	81	4.7%	324	-
scale_factor_1	74	66	10.8%	266	-
speed_control_2	277	247	10.8%	5217	-
tone_detector_1	84	75	10.7%	536	-
complex_multiply	33	30	9.1%	2	18
convolution	49	45	8.2%	308	12
fir	178	158	11.2%	5482	25
iir_biquad	132	128	3.0%	1632	32
least_mean_square	115	102	11.3%	2776	49
matrix_multiply	89	85	4.5%	1011	26

hand. For instance, for the *complex multiply* benchmark, the code generated by our algorithm was 1.7 times larger than the corresponding hand-written code. For the *matrix multiply* benchmark, the compiler-generated code was 3.3 times larger than the corresponding hand-generated code. Although these differences in code size are quite large, they are conservative because not all optimizations have been fully integrated into the SPAM compiler. In particular, important machine-independent optimizations such as *global common-subexpression elimination* and *dead-code elimination* [Aho et al. 1986] are not performed. Additionally, pointer variable accesses with post-increment/decrement are not directly translated into register-indirect memory accesses with auto-increment. We are very confident that once these optimizations are fully in place, the differences in code size will be considerably lower.

5.1.1 *Adapting to Other Architectures*. Our approach has the advantage that it can be efficiently adapted to more complex architectures, e.g., those with four data memory banks. It is assumed the compiler writer (henceforth referred to as *developer*) will rewrite the input file that Olive uses to generate uncompacted symbolic assembly code. We now describe how we have generalized the following routines so that efficient code can be generated for many of these architectures:

5.1.1.1 *Compaction Routines*. We have implemented a generalized list scheduling interface that compacts an arbitrary number of ALU and move operations. The developer must provide this interface a *tabular* specification of all operations that may be performed concurrently by the target machine. Each entry in this *compaction table* contains a sequence of opcodes—this *pattern* specifies that it is permissible to compact a set of operations whose opcodes match this sequence, provided that no data-dependency constraints are violated. The first set of patterns must specify all legal ALU operation combinations, while the second set must specify all legal move combinations. Associated with each ALU pattern is a sequence

of *links* to move patterns. Each link specifies that it is legal to compact a set of operations whose opcodes match the ALU and move patterns, provided that no constraints are violated. The list scheduler uses the compaction table to perform local compaction as follows. Suppose the scheduler wishes to schedule a move operation M into an instruction C. First, M is temporarily inserted into C. Next, a hash function is applied to the opcodes of C representing ALU operations so as to find the corresponding ALU pattern P_1 in the table. This function is then applied to the move opcodes of C to find the corresponding move pattern P_2. The scheduler then permanently schedules M into C if P_2 exists and there exists a link from P_1 to P_2. The situation is analogous if M were to represent an ALU operation.

5.1.1.2 *Constraint graph routines*. Construction of a constraint graph proceeds by first generating the set of symbolic register and variable vertices directly from the current procedure's compacted assembly code. We classify the set of constraint graph edges according to their dependency on the target machine. Red edges are machine-*independent* and must exist in any constraint graph — these edges are generated automatically by performing live-variable analysis on the compacted code. Other edges (e.g., parallel move edges, ALU constraint edges, etc.) are machine-*dependent* and are generated as follows. First, the developer must declare and define C^{++} classes corresponding to the various machine-dependent constraint graph edges. Associated with each of these classes must be a *cost method* which, when invoked, checks all involved constraint graph vertices in order to determine whether all labelling constraints associated with the edge have been appropriately satisfied, and then returns the incurred penalty. Second, the developer must define and associate with each link in the compaction table data structure a machine-specific *edge insertion* function. This function must take as input a pointer to the current constraint graph and a pointer to a compacted instruction whose opcodes match the corresponding ALU and move patterns, and insert into the constraint graph the appropriate machine-dependent edges. The compiler constructs the necessary machine-dependent constraint graph edges by analyzing each assembly instruction in the compacted code, determining the link in the compaction table that corresponds to the instruction, and then invoking the associated edge-insertion function.

5.1.1.3 *Simulated annealing routines*. We have implemented a generalized annealing framework that uses the parameters described in Section 5. Two target-specific enumerations must be provided to this framework—these specify the set of valid labellings for the symbolic register and variable nodes, respectively, of the input constraint graph. Additionally, an array must be provided, where the i^{th} element specifies the number of address registers available to access the automatic variables allocated to the i^{th} memory bank. At each step of the annealing process, a constraint graph vertex is chosen at random, and a label from the appropriate enumeration is randomly chosen for it. The cost method associated with

Table III. Impact of Simultaneous Reference Allocation on Hand-Written Benchmarks

Hand-written Benchmark	Nodes	Edges	Initial Size	Final Size	Code Size Improv	Time (s)
rvb1	76	186	120	51	57.5%	1872
rvb2	76	214	88	41	53.4%	1329

each constraint graph edge e is then invoked, and returns the penalty, if any, incurred by e. If a symbolic variable node is relabelled, offset assignment is performed for those two memory banks whose variable allocation has consequently changed.

5.2 Improving the Quality of Completely Referenced Code

We have also implemented our code generation algorithm as a stand-alone program. Its input is an assembly file in which all register and variable references are symbolic, and its output is an assembly file in which these references are physical. This program has been used to improve the quality of various *completely referenced* files, i.e., assembly files for which reference allocation has already been performed.

In particular, we optimized two hand-written non-kernel benchmarks that were obtained from Motorola's public bulletin board: *rvb1* and *rvb2* are unoptimized and optimized versions, respectively, of a reverberation algorithm. The kernel benchmarks from this bulletin board were heavily hand-optimized and left no room for improvement. However, *rvb1* and *rvb2* had much room for improvement—memory references were predominantly addressed absolutely, thus resulting in increased code size. It is possible that due to time constraints in developing these non-kernel benchmarks, the assembly programmer chose to reference variables absolutely, rather than manually performing offset assignment and referencing them indirectly.

We first manually rewrote these two benchmarks so that all references were symbolic, and then provided these files to our stand-alone. The results of our experiments are described in Table III: the first column lists the benchmarks; the next two columns specify the number of vertices and edges, respectively, in the associated constraint graphs; the next two columns specify the size (in instruction words) of the original hand-written assembly code and the final optimized assembly code, respectively; the sixth column specifies the percentage reduction of the numbers in the fifth column over the numbers in the fourth column; the final column specifies the time (in seconds) required to optimize each benchmark. It is apparent that our stand-alone was very successful at improving the code quality of these benchmarks—the code size of *rvb1* was reduced from 120 words to 51 words, a 57.5% reduction, while the size of *rvb2* was reduced from 88 words to 41 words, a 53.4% reduction. This technique can be applied to any completely referenced file that either has been generated by an unoptimizing compiler or has not been hand-optimized thoroughly.

6. CONCLUSIONS

Current compiler technology is unable to take advantage of the potential increase in parallelism offered by multiple data memory banks. Consequently, compiler-generated code is much inferior to hand-written code. We have devised a graph labelling-based algorithm that attempts to maximize the benefit of this architectural feature. A constraint graph is constructed with a vertex for each symbolic register and variable in the compacted code, and edges representing constraints that must be satisfied when labelling these vertices. Each labelling of the constraint graph has an associated cost, a lower cost implying a better labelling. Since optimal labelling is NP-hard, we search for heuristics that will find a low-cost labelling.

We choose to use simulated annealing since our problem characteristics match its criteria. Although it is computationally expensive, we find its use acceptable since our intention is to synthesize code of the highest quality. Simulated annealing is not guaranteed to find an optimal solution, but it does work very well in practice. Experimental results demonstrate that our algorithm not only generates high-quality compiled code, but also substantially improves the quality of completely-referenced code. Our algorithm can also be efficiently adapted to more complex architectures.

REFERENCES

AHO, A., GANAPATHI, M., AND TJIANG, S. W. K. 1989. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst. 11*, 4 (Oct. 1989), 491–516.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

ANDERSON, J. M. AND LAM, M. S. 1993. Global optimizations for parallelism and locality on scalable parallel machines. *SIGPLAN Not. 28*, 6 (June 1993), 112–125.

ARAUJO, G. AND MALIK, S. 1995. Optimal code generation for embedded memory non-homogeneous register architectures. In *Proceedings of the Eighth International Symposium on System Synthesis* (Cannes, France, Sept. 13–15, 1995), P. G. Paulin and F. Mavaddat, Eds. ACM Press, New York, NY, 36–41.

ARAUJO, G., MALIK, S., AND LEE, M. T.-C. 1996. Using register-transfer paths in code generation for heterogeneous memory-register architectures. In *Proceedings of the 33rd Annual Conference on Design Automation* (DAC '96, Las Vegas, NV, June 3–7), T. P. Pennino and E. J. Yoffa, Eds. ACM Press, New York, NY, 591–596.

ARAUJO, G., SUDARSANAM, A., AND MALIK, S. 1996. Instruction set design and optimization techniques for address computation in DSP architectures. In *Proceedings of the Ninth International Symposium on System Synthesis*,

CHAITIN, G., AUSLANDER, M., CHANDRA, A., COCKE, J., HOPKINS, M., AND MARKSTEIN, P. 1981. Register allocation via coloring. *Comput. Lang. 6*, 1, 47–57.

FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. 1992. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst. 1*, 3 (Sept. 1992), 213–226.

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY.

KAFKA, S. 1990. An assembly source level global compacter for digital signal processors. In *Proceedings of the International on Acoustics, Speech, and Signal Processing*,

KIRKPATRICK, S., GELATT, C. D., JR., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598 (May), 671–680.

LANDSKOV, D., DAVIDSON, S., SHRIVER, B., AND MALLETT, P. 1980. Local microcode compaction techniques. *ACM Comput. Surv. 12*, 3 (Sept.).

LANNEER, D., PRAET, J., SCHOOFS, K., GEURTS, W., THOEN, F., GOOSSENS, G., AND KIFLI, A. 1995. Chess: Retargetable code generation for embedded DSP rocessors. In *Code Generation for Embedded Processors*, P. Marwedel and J. Goosens, Eds. Kluwer Academic Publishers, Hingham, MA.

LEUPERS, R. AND MARWEDEL, P. 1996. Instruction selection for embedded DSPs with complex instructions. In *Proceedings of the Conference on European Design Automation* (EURO-DAC '96, Geneva, Switzerland, Sept. 16\), G. Symonds and W. Nebel, Eds. IEEE Computer Society Press, Los Alamitos, CA, 200–205.

LEUPERS, R., SCHENK, W., AND MARWEDEL, P. 1994. Retargetable assembly code generation by bootstrapping. In *Proceedings of the Seventh International Symposium on High-Level Synthesis* (ISSS '94, Niagara-on-the-Lake, Ont., Canada, May 18–20), P. G. Paulin, Ed. IEEE Computer Society Press, Los Alamitos, CA, 88–93.

LI, J. AND CHEN, M. 1991. Compiling communication-efficient programs for massively parallel machines. *IEEE Trans. Parallel Distrib. Syst. 2*, 3 (July), 361–376.

LIAO, S., DEVADAS, S., KEUTZER, K., AND TJIANG, S. 1995. Instruction selection using binate covering for code size optimization. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD-95, San Jose, CA, Nov. 5–9), R. Rudell, Ed. IEEE Computer Society Press, Los Alamitos, CA, 393–399.

LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. 1995. Storage assignment to decrease code size. In *Proceedings of the Conference on Programming Language Design and Implementation* (SIGPLAN '95, La Jolla, CA, June 18–21), D. W. Wall, Ed. ACM Press, New York, NY, 186–195.

LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., WANG, A., ARAUJO, G., SUDARSANAM, A., MALIK, S., ZIVOJNOVIĆ, V., AND MEYR, H. 1996. Code generation and optimization techniques for embedded digital signal processors. In *Hardware/Software Co-Design*, G. D. Micheli and M. Sami, Eds. Kluwer Academic Publishers, Hingham, MA, 165–186.

MOTOROLA 1990. DSP56000/DSP56001 Digital Signal Processor User's Manual. Motorola Inc., Phoenix, AZ.

PAULIN, P. G., LIEM, C., MAY, T. C., AND SUTARWALA, S. 1994. CodeSyn: A retargetable code synthesis system (abstract). In *Proceedings of the Seventh International Symposium on High-Level Synthesis* (ISSS '94, Niagara-on-the-Lake, Ont., Canada, May 18–20), P. G. Paulin, Ed. IEEE Computer Society Press, Los Alamitos, CA, 94.

POWELL, D., LEE, E., AND NEWMAN, W. 1992. Direct synthesis of optimized DSP assembly code from signal flow block diagrams. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing 5*, 553–556.

SAGHIR, M. A. R., CHOW, P., AND LEE, C. G. 1996. Exploiting dual data-memory banks in digital signal processors. *ACM SIGOPS Oper. Syst. Rev. 30*, 5, 234–243.

SUDARSANAM, A. AND MALIK, S. 1995. Memory bank and register allocation in software synthesis for ASIPs. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD-95, San Jose, CA, Nov. 5–9), R. Rudell, Ed. IEEE Computer Society Press, Los Alamitos, CA, 388–392.

TEXAS-INSTRUMENTS 1993. TMS320C2x User's Guide. Revision C. Texas Instruments, Austin, TX.

WESS, B. 1991. Automatic code generation for integrated digital signal processors. In *Proceedings of the 1991 IEEE International Symposium on Circuits and Systems* (Singapore, June 11–14), 33–36.

WESS, B. 1992. Automatic instruction code generation based on trellis diagrams. In *Proceedings of the International Conference on Circuits and Systems*, 645–648.

ZIVOJNOVIĆ, V., VELARDE, J. M., AND SCHLÄGER, C. 1994. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the Fifth International Conference on Signal Processing Applications and Technology* (Oct.).

Received: October 1996; revised: February 1997; accepted: July 1998