Memory Bank and Register Allocation in Software Synthesis for ASIPs

Ashok Sudarsanam Sharad Malik
Department of Electrical Engineering
Princeton University

Abstract—An architectural feature commonly found in digital signal processors (DSPs) is multiple data-memory banks. This feature increases memory bandwidth by permitting multiple memory accesses to occur in parallel when the referenced variables belong to different memory banks and the registers involved are allocated according to a strict set of conditions. Unfortunately, current compiler technology is unable to take advantage of the potential increase in parallelism offered by such architectures. Consequently, most application software for DSP systems is hand-written – a very time-consuming task.

We present an algorithm which attempts to maximize the benefit of this architectural feature. While previous approaches have decoupled the phases of register allocation and memory bank assignment, our algorithm performs these two phases simultaneously. Experimental results demonstrate that our algorithm substantially improves the code quality of many compiler-generated and even hand-written programs.

I. Introduction

An emerging trend in the implementation of DSP systems is the increasing use of programmable processors, such as off-the-shelf or application-specific processors (ASIPs). Another trend, due to cost and power constraints, is the integration of the processor, program RAM and ROM, and ASIC circuitry into a single integrated circuit. Consequently, program ROM size is limited. In this scenario, the code density must be high to keep ROM size low. Additionally, the software must be written so as to meet high-performance constraints, which may include hard real-time constraints. Unfortunately, the code quality of current DSP compilers 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. Therefore, most application software is hand-written – a very time-consuming task.

Our research aims to overcome the limitations of current compilation techniques for DSPs. We would like to identify the individual architectural characteristics which make code generation difficult and provide specific solutions for each of these. Our current research focuses on providing support for *multiple data-memory banks*. This feature, found in several commercial DSPs such as the *Motorola 56000* and *NEC 77016*, increases memory bandwidth by permitting multiple memory accesses to occur in parallel when the referenced variables belong to different memory banks and the registers involved conform to a strict set of conditions. Furthermore, the instruction set of this architecture encodes parallel accesses in a single instruction word, thus assisting in the generation of dense code. We will be using the *Motorola 56000* as our experimental vehicle in this research [4].

We present an algorithm which 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*. Our algorithm is based on *graph labelling*, the objective of which 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 NP-hard, we use *simulated annealing* to find a good labelling.

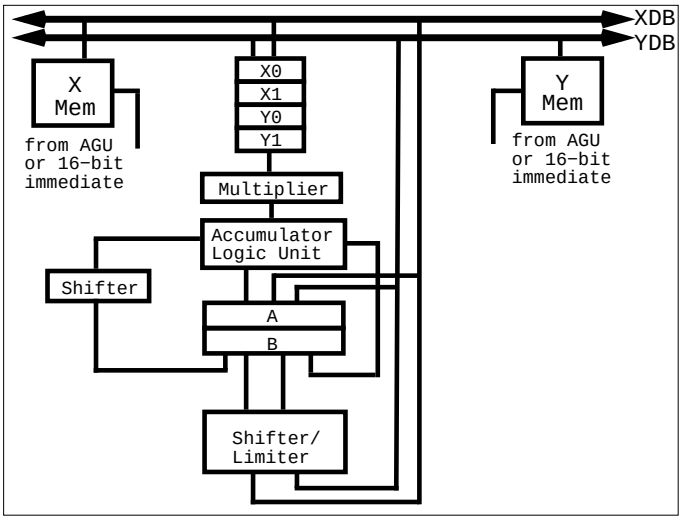

Fig. 1. Data ALU and Memory Banks

This paper is organized as follows: Section II gives an overview of the *Motorola 56000* architecture. Section III describes our graph-labelling algorithm. Section IV provides experimental results and finally, Section V presents our conclusions and future work.

II. Motorola 56000 DSP Architecture

The *Motorola 56000* architectural units of interest are the Data Arithmetic-Logic Unit (*Data ALU*), Address Generation Unit (*AGU*), and *X and Y data-memory banks*:

- **Data ALU**: This unit, shown in Figure 1, contains hardware that efficiently performs arithmetic and logical operations. The data ALU consists of four 24-bit input registers named *X0*, *X1*, *Y0*, and *Y1*, and two 56-bit accumulators named *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 named *XDB* and *YDB* permit two input registers or accumulators to be read or written in conjunction with the execution of an ALU operation.
- **AGU**: This unit contains two files of 16-bit address registers, one consisting of registers *R0* through *R3*, and the other consisting of *R4* through *R7*. Associated with each file is an ALU which, on each cycle, can increment or decrement a single register from that file. Two multiplexers permit two effective addresses to be generated on each cycle – one address is generated per register file, and these addresses must point to locations in *different* data-memory banks.
- **X and Y Data-Memory Banks**: This unit, also shown in Figure 1, consists of two independent 512 word x 24-bit data-memory banks which permit two memory accesses to occur in parallel. When register-indirect addressing is used, the AGU generates the effective address. Alternatively, a 16-bit constant, generated as part of the instruction word, may be used for absolute addressing.

The *56000* instruction set encodes parallel operations in a single 24-bit word, resulting in dense, high-bandwidth code. In particular, up to two *move* operations can be specified in one instruction word, where a move refers to a *memory access* (load or store), *register transfer* (moving of data from input register to accumulator or vice-versa), or *immediate load* (loading of 24-bit constant into input register or accumulator). Additionally, most data ALU operations can be encoded in a single word with 1 or 2 move operations. However, there are restrictions on which move operations can be performed in parallel:

- A memory access can be performed in parallel with another memory access or register transfer
- A register transfer can be performed in parallel with a memory access or immediate load
- An immediate load can be performed in parallel with a register transfer

Furthermore, two move operations can occur simultaneously only if the associated registers and memory banks are allocated appropriately. Consider the following parallel move:

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

The above parallel memory access is legal because certain register and memory bank constraints have been appropriately satisfied:

- The two moves access data in different data-memory banks.
- The X data-memory access loads into a restricted set of locations, namely *X0*, *X1*, *A* or *B*.
- The Y data-memory access loads into a restricted set of locations, namely *Y0*, *Y1*, *A* or *B*.
- Both memory accesses use register-indirect addressing – the address registers specified belong to *different* AGU files.

Thus, the following parallel memory accesses are NOT permitted:

- MOVE X:(R0), X1 Y:(R4), X0
  The Y data-memory access does not load into the allowable set of locations, namely *Y0*, *Y1*, *A* or *B*
- MOVE X:(R0), X1 X:(R1), X0
  Both memory accesses are to the same bank
- MOVE X:(R0), X1 Y:#0034, Y1
  The Y data- memory access does not use register-indirect addressing
- MOVE X:(R0), X1 Y:(R2), Y1
  The specified address registers belong to the same AGU file

Similar restrictions exist for the other parallel move combinations which this architecture permits.

Thus, given the nature of the *56000* architecture, our intent is to develop an algorithm which takes full advantage of the available parallelism.

III. 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 the instruction selection phase. We assume that the code-generator outputs optimized straight-line code with symbolic register and variable references. One can visualize the code-generator as first producing optimized straight-line code with complete reference allocation, then replacing all physical references with symbolic ones. Our objective is to assign physical registers and memory banks to the symbolic registers and variables, respectively, such that high-quality code is generated.

Previous approaches have *decoupled* the phases of register and memory bank allocation. Wess' technique [10] first generates code in which register allocation, but not memory bank allocation, has been performed. After the code is compacted, memory bank allocation is performed so as to satisfy the parallel move requirements. This entire process is repeated until a valid allocation is found.

The algorithm of Powell *et al.* [8] first performs register allocation on the symbolic assembly code. Thereafter, variables are assigned to memory banks in an alternating fashion – the first variable is assigned to the X data-memory, the second to the Y data-memory, and so forth. Finally, the code is compacted into as few instructions as possible without violating any data-dependency or target machine constraints.

Since the validity of parallel move operations is so highly dependent on *both* register and memory bank allocation, we believe that these two phases should be performed *simultaneously*. By decoupling the two phases, these two algorithms lose all guarantees of optimality before the code is compacted.

A. Assumptions

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

A.1 Global Variables and Parameters

To determine the optimal memory bank assignment for a given global variable, we need to observe its references over *all* procedures. The fact that code is normally generated one procedure at a time prevents us from doing this. A similar argument can be made about parameters, if they are passed as pointers. This problem can be addressed by using *inter-procedural* data-flow analysis. For now, we assume that all global variables and parameters are statically allocated in a single memory bank.

A.2 Aggregate Variables

Arrays and other aggregate variables also pose problems for us, particularly during the address register allocation phase. For now, we simply treat each array element as a distinct variable. We are currently working on methods which will deal with globals, parameters, and aggregate variables in a more sophisticated fashion.

Given the assumptions surrounding our code generation technique, we now give a detailed account of each step in our algorithm.

B. Step I: Compaction

The first step in our algorithm *compacts* the symbolic straight-line code. Two move operations are compacted into a single instruction if no data dependencies are violated and the parallel move combination is permitted by the architecture. Additionally, an ALU operation is compacted with up to two move operations if no data dependencies are violated and it is permissible for this ALU operation to be specified in conjunction with parallel moves.

Our current compaction method is rather conservative in that it does not permit out-of-order execution of operations. Our results demonstrate, however, that a significant improvement in code-quality is achieved. We are currently looking into more aggressive compaction techniques.

C. Step II: Constraint Graph Creation

We view reference allocation as a process of *constraint graph labelling*. A constraint graph has a vertex for each symbolic register and variable, and edges which represent constraints that need to be satisfied when labelling the vertices. Each register-vertex must be labelled *X0*, *X1*, *Y0*, *Y1*, *A* or *B*, while each variable-vertex must be labelled either *X-MEM* or *Y-MEM*. Associated with each edge is a *cost*, or penalty, which is incurred when the corresponding constraints are violated. A similar technique is used in register allocation for homogeneous register architectures [2]. The next few sections will describe each such edge.

C.1 Red (Interference) Edges

A **red edge** is added between vertices reg_x and reg_y, corresponding to symbolic registers *x* and *y*, if and only if *x* and *y* are simultaneously *live* [1]. This edge specifies that nodes reg_x and reg_y must be labelled differently, or analogously, registers *x* and *y* must be assigned to different physical registers. Otherwise, *spill code* will be required to save one of these values to memory and to retrieve it upon each use.

The amount of spill code required due to an unsatisfied red edge varies, depending on how many times the spilled value is used subsequently. We currently take a conservative approach and assign the spill cost a constant value of *10*. This represents an *estimate* of the additional amount of code required to implement a spill. Due to the conservative nature of our compaction scheme, it is not necessary to compute the *actual* spill cost. Since compaction does not result in out-of-order execution of operations, the set of variables live at each point in the program does not change. Assuming that the symbolic uncompacted code contains all necessary spill code, a register allocation exists which requires *no* additional spilling. Thus, by assigning the spill cost a sufficiently large value, we reduce the probability that two vertices connected by a red edge will be labelled identically.

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

C.2 Green Edges

A **green edge** is added to the constraint graph for each parallel move corresponding to a dual memory access. In particular, given a parallel move of the form

MOVE	var_i, reg_i	var_j, reg_j
MOVE	var_i, reg_i	reg_j, var_j
MOVE	reg_i, var_i	var_j, reg_j
MOVE	reg_i, var_i	reg_j, var_j

we add a green edge between vertices reg_i and reg_j. The green edge also includes pointers to vertices var_i and var_j. These pointers specify that reg_i and var_i constitute a single move operation, while reg_j and var_j constitute another.

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

- Vertices var_i and var_j must be labelled differently.
- If vertex var_i is labelled *X-MEM* (*Y-MEM*), vertex reg_i must be labelled *X0*, *X1*, *A* or *B* (*Y0*, *Y1*, *A*, or *B*).
- If vertex var_j is labelled *X-MEM* (*Y-MEM*), vertex reg_j must be labelled *X0*, *X1*, *A* or *B* (*Y0*, *Y1*, *A*, or *B*).

If the current procedure consists of a single basic block, then each unsatisfied green edge incurs a cost of *1*. This represents the resulting increase in instruction words and cycles: a parallel move instruction corresponding to an unsatisfied green edge must be separated into two individual move operations.

If the current procedure is composed of multiple basic blocks, then each unsatisfied green edge incurs a cost of *1 * basic-block-frequency*, where basic-block-frequency is an estimate of the number of times the current basic block is executed per procedure invocation. This information is obtained from profiling analysis.

C.3 Blue, Brown, and Yellow Edges

In a similar manner, we add other edges to the constraint graph based on the type of parallel move instruction encountered in the compacted code. Each of these edges has a set of constraints which must be satisfied when labelling the graph, plus an associated cost.

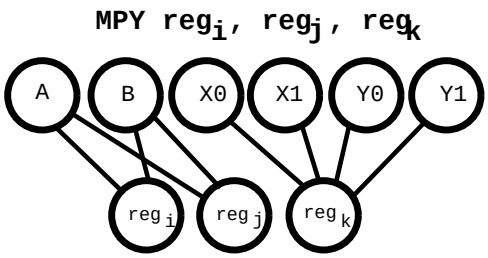

Fig. 2. Example of black edge construction

- **Blue Edges:** added for each parallel move corresponding to a memory load and register transfer.
- **Brown Edges:** added for each parallel move corresponding to an immediate load and register transfer.
- **Yellow Edges:** added for each parallel move corresponding to a memory store and register transfer.

C.4 Black Edges

Each ALU operation in the *56000* instruction set 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 more restrictive. Consider the *multiply* operation:

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

The *56000* architecture restricts registers *i* and *j* to be input registers only. Register *k* must then be an accumulator. We enforce these constraints in our constraint graph by introducing **black edges** and **global vertices**, which correspond to input registers and accumulators. A black edge between register-vertex *x* and global-vertex *y* implies that vertex *x* can *not* be labelled with the register associated with vertex *y*. Thus, in order to enforce the constraint that register *i* must be an input register, we add black edges between vertex reg_i and the global vertices corresponding to the two accumulators. Figure 2 shows the complete set of black edges needed for the multiply operation.

Each unsatisfied black edge has an associated cost of ∞ since the register allocation implied by an illegally-labelled black edge can not be supported by ALU hardware.

C.5 Address Register Allocation

Given some memory bank allocation, we now need to determine the form of addressing used in each memory access. Each instruction containing two memory accesses *must* use register-indirect addressing. Instructions with one memory access may use either register-indirect or absolute addressing. Each use of absolute addressing incurs a one word penalty, however, since an extra word is needed to store the 16-bit immediate. Since one of our goals is to minimize the total number of instruction words, we choose to use register-indirect addressing *exclusively*.

Now each memory access can be performed only if an address register is available which contains the appropriate address. Recall that the *56000* architecture features two files of address registers which point to locations in the data memories. Associated with these registers are auto-increment and auto-decrement update modes which can be used to efficiently access the various variables in memory. Rather than explicitly initializing an address register before each access, an intelligent placement of variables within the memory banks could make use of

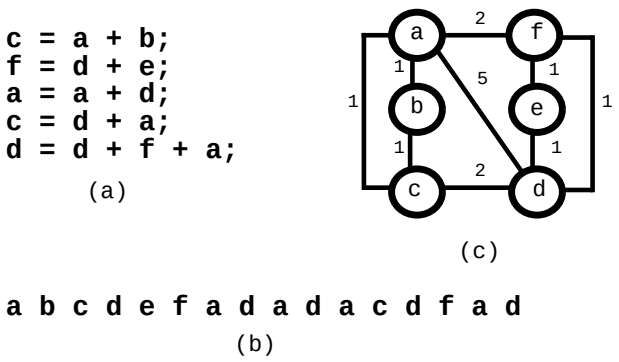

Fig. 3. (a) Original code sequence (b) Access sequence (c) Access graph

these modes and thus minimize the required number of set-up instructions. This is precisely what the *Offset Assignment Problem* attempts to do. The following description summarizes the work presented in [7].

There are two flavors of offset assignment: *Simple* Offset Assignment (*SOA*) assumes that one address register is available for memory accesses, while *General* Offset Assignment (*GOA*) assumes that multiple address registers are available. Since parallel memory accesses must employ address registers from different AGU files, we arbitrarily assign registers *R0* through *R3* to the X data-memory bank and registers *R4* through *R7* to the Y data-memory bank. We must therefore perform *GOA* on each memory bank.

Figure 3 shows the main structures used by *SOA* and *GOA*. Assume we are given a code sequence of variable references, as shown in Figure 3(a). We first form an *access sequence*, shown in Figure 3(b), by linearizing the variable references. We next form an *access graph* in which each vertex corresponds to a distinct variable, and edge (var_i, var_j) with weight *w* exists if and only if variables *i* and *j* are adjacent to each other *w* times in the access sequence. This graph, shown in Figure 3(c), conveys the relative benefits of assigning each pair of variables to adjacent locations in memory. With respect to the access sequence, the cost of an assignment is equal to the number of adjacent accesses of variables which are not assigned to adjacent memory locations. With respect to the access graph, the cost of an assignment is equal to the sum of the weights of all edges connecting variables which are not assigned to adjacent memory locations.

The idea behind *Offset Assignment* is as follows: if two variables are accessed together frequently (i.e. the edge connecting the two variables is of high weight), then they should be assigned to adjacent memory locations. Consequently, an address register can efficiently move between these two locations with the help of the auto-increment and auto-decrement modes. Thus, for each high-weighted edge in the access graph, we need to assign the corresponding variables to adjacent memory locations. Since this may not be possible for all such edges, variables should be assigned in such a way that the sum of the weights of all edges connecting variables not assigned to adjacent memory locations is *minimized*. This is equivalent to finding a *maximum-weighted path covering* (MWPC) of the access graph. Since MWPC is NP-hard, heuristics are used to find an approximate solution, i.e. a good, though not necessarily optimal, arrangement of variables in memory. *GOA* is tackled by extending the heuristic solutions of *SOA*.

Recall that each array element is treated as a distinct variable. The reason for this is that offset assignment currently works for scalar variables only. Work is currently being done on offset assignment which will permit aggregates to be treated accordingly.

D. Step III: Constraint Graph Labelling

The cost of a particular constraint graph labelling is equal to the unsatisfied edge costs plus the *GOA* cost. We wish to find a labelling of *least* cost since such a labelling represents a code sequence with the least deviation from the schedule formed by compaction. It is easy to see that optimal labelling of the constraint graph is NP-hard because it subsumes three other NP-complete/NP-hard problems [3]:

- **Graph K-Colorability Problem**: This problem is encountered in register allocation for homogeneous architectures.
- **Maximum-Weighted Path Covering** (*MWPC*): This problem is encountered in *Offset Assignment.*
- **Maximum Bipartite Subgraph Problem**: This problem is encountered if we just consider plain memory bank allocation without additional constraints on register allocation.

Therefore, a heuristic is needed which will instead generate a *low-cost* labelling of the constraint graph. A number of important observations can be made about constraint graph labelling:

- **Complex cost function**: The cost function is complex with respect to optimizing it within the solution space.
- **Large solution space with hills and valleys**: Given *m* register and *n* variable vertices, the total number of possible labellings is $6^m * 2^n$. 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 in turn to determine whether or not its constraints have been satisfied and the cost is updated appropriately. *GOA* heuristics are applied so as to determine the additional cost of allocating address registers. With an intelligent choice of data structures, determining the cost of a solution can be done efficiently.
- **Easy to generate new solution**: A vertex is randomly chosen and its label is changed.

These observations suggest that finding a low-cost labelling of the constraint graph is especially well-suited for *simulated annealing* [5]. Simulated annealing is a probabilistic hill-climbing algorithm which has been used successfully in several design automation and combinatorial optimization problems. As our results demonstrate in the next section, our implementation of simulated annealing does a good job of finding a low-cost labelling of the constraint graph.

IV. Experiments and Results

We have implemented a program which accepts uncompacted, symbolic assembly code and generates high-quality compacted code.

We obtained two sets of benchmarks for purposes of experimentation. The first set came from the DSPstone benchmark suite [9] and contained compiled code for various DSP kernels: *complex multiply*, *complex update*, *real update*, *iir biquad*, *convolution*, *fir*, and *lms* (least-mean square). The second set was obtained from Motorola's public bulletin board and contained hand-written code for a few DSP algorithms: *rvb* (reverberation) and *adpcm* (speech encoding). We obtained two versions of the reverberation program – *rvb1* was unoptimized while *rvb2* was optimized.

Each benchmark was first rewritten into a symbolic format and then fed as input to our program. Table I demonstrates how our algorithm performed on both sets of benchmarks. The first two columns of Table I convey the sizes of the resulting constraint graphs: the graphs for the DSPstone benchmarks were quite small, while the graphs for the hand-written benchmarks were much larger.

The columns labelled *Initial Code Size* and *Code Size After Annealing* demonstrate that our algorithm was able to significantly reduce the code size of each benchmark. In the DSPstone set, our algorithm

Benchmark	Constraint Graph Nodes	Constraint Graph Edges	Initial Code Size	Code Size After Annealing	Annealing Time (s)	Code Size After Greedy
complex multiply	14	39	22	12	6.9	12
complex update	23	58	38	28	33.5	28
real update	9	14	12	9	1.8	9
iir biquad	25	73	35	21	61.3	23
convolution	7	18	10	8	1.4	8
fir	15	41	20	15	6.8	15
lms	30	76	48	35	40.7	35
rvb1	76	186	120	51	1872.4	52
rvb2	76	214	88	41	1329.0	44
adpcm	171	579	275	196	10268.7	200

TABLE I
SIMULATED ANNEALING VS. GREEDY ALLOCATION

reduced the size of *iir biquad* from *35* words to *21* words, a 40 percent reduction. In the Motorola set, our algorithm reduced the size of *rvb1* from *120* words to *51* words, a 58 percent reduction.

The substantial improvement in code size of the DSPstone benchmarks can be attributed to the naive reference allocation performed by the *56000* compiler: all variables were placed in the Y data-memory bank and all memory references used absolute addressing. The hand-written benchmarks did make use of both data-memory banks, however, but absolute addressing was still heavily employed. It is conceivable that the large number of variables in these benchmarks forced the assembly programmer to use absolute addressing for sake of clarity, at the expense of code density.

The amount of CPU time (in seconds) required to perform simulated annealing on each benchmark is given under the column labelled *Annealing Time*. These measurements were taken on a *Silicon Graphics Indigo R4000* with 32 MB of RAM. The times for the DSPstone benchmarks were quite low, which can be attributed to the small constraint graph sizes. In contrast, the times for the Motorola benchmarks were quite high, obviously due to the large constraint graph sizes. Although simulated annealing is computationally-expensive, we find its use completely acceptable because it yields high quality results.

For each benchmark, we compared our method with *greedy allocation*, which works as follows: the memory bank assigned to a given variable is the opposite of that assigned to the variable last referenced. Moreover, if a variable is assigned to the X (Y) data-memory bank, then preference is given to registers *X0* and *X1* (*Y0* and *Y1*) when allocating the corresponding register. One can see, from the column labelled *Code Size After Greedy*, that there was very little difference in the quality of code generated by both techniques. However, our approach has the advantage that it can be easily extended to more complex architectures, whereas the generality of greedy allocation is unknown. For instance, there exist architectures with *four* data-memory banks [6], allowing a total of four data-memory accesses to occur in parallel. By simply modifying the compaction and annealing routines, our approach can easily be extended to support this architecture, whereas it is not clear at all how greedy allocation would perform. Hence, the important issue of scalability strongly favors our approach over greedy allocation.

V. CONCLUSIONS AND FUTURE WORK

We have presented an algorithm which attempts to maximize the benefit of multiple data-memory banks. *Simulated annealing* is used to find a low-cost labelling of the *constraint graph*. Experimental results demonstrate that our algorithm provides significant improvements over compiler-generated code, and even large hand-written programs.

We are currently looking into ways of further improving code quality. We are investigating more aggressive compaction schemes, which will significantly increase the amount of compaction, and inter-procedural data-flow analysis, which will allow us to find appropriate register and memory bank allocations for global variables and parameters.

VI. ACKNOWLEDGEMENTS

This research was supported by an NSF NYI award (grant MIP 9457396).

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, (6):47–57, 1981.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.

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

[5] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. *Science*, 220:671–680, 1983.

[6] E.A. Lee. Programmable DSP Architectures: Part I. IEEE *ASSP Magazine*, pages 4–19, Oct. 1988.

[7] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage Assignment to Decrease Code Size. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 186–195, June 1995.

[8] D.B. Powell, E.A. Lee, and W.C. Newman. Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams. *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, 5:553–556, 1992.

[9] V. Živojnović, J. Martínez Velarde, and C. Schläger. DSPstone: A DSP-oriented Benchmarking Methodology. Technical report, Aachen University of Technology, August 1994.

[10] B. Wess. Automatic code generation for integrated digital signal processors. *Proceedings International Symposium on Circuits and Systems*, pages 33–36, 1991.