A MULTI-OBJECTIVE INTEGER LINEAR PROGRAM FOR MEMORY ASSIGNMENT IN THE DSP DOMAIN

G. Gréwal, S. Coros and D. Banerji
Department of Computing and Information Science
University of Guelph,
Guelph, ON, Canada,
N1G 2W1

A. Morton
Department of Electrical and Computer Engineering
University of Waterloo,
Waterloo, ON, Canada
N2L 3G1

ABSTRACT

To increase memory bandwidth, many programmable Digital Signal Processors (DSPs) employ two on-chip data memories. This architectural feature supports higher memory bandwidth by allowing multiple data memory accesses to occur in parallel. Exploiting dual memory banks, however, is a challenging problem for compilers. This, in part, is due to the instruction-level parallelism, small numbers of registers, and highly specialized register capabilities of most DSPs. In this paper, we present a new methodology, based on *Integer Linear Programming (IP)*, for assigning data objects to dual-bank memories. Our approach is global, and integrates several important issues in memory assignment within a single model. Special effort is made to identify those objects that could potentially benefit from an assignment to a specific memory, or perhaps duplication in both memories. Our computational results show that the IP is lightening fast, and is able to achieve a 54% reduction in the number of memory cycles and a reduction in the range of 7% to 42% in the total number of cycles when tested with well-known DSP kernels and applications.

KEY WORDS
integer programming, digital-signal processing, dual-memory assignment

1. Introduction

To increase memory bandwidth, many programmable Digital-Signal Processors (DSPs) employ two data memories. Examples include the Motorola DSP56000, SGS-Thomson 18950, Analog Devices ADSP2106x, and NEC mPD77016, just to name a few. The objective is to improve performance by maximizing the parallelism in the software pipeline. Parallelism is promoted by concurrently loading pairs of operands from opposite memories whenever an operation requires two memory-resident values. Assigning data in the best way to dual memories, however, remains an elusive problem.

Whenever operands must be fetched from the same memory, their load operations must occur on different control steps. Furthermore, it may not be possible to perform all of the required loads in parallel with the existing ALU operations. If even a few consecutive ALU operations pull too many operands from the same memory, additional instructions may be needed just to load values.

The register structure of the target architecture also has a direct effect on memory assignment. On some machines, the operand registers can only receive values from one corresponding memory; on others they are connected only to certain input ports of the ALU; on other machines, both restrictions apply. Such restrictions may occur only when the load operation is done in parallel with other operations, but parallelism is generally preferred for other reasons. The implication is that when a value appears in the "wrong" memory, it must first be fetched, then copied from the register associated with the memory to a register connected to the input terminal of the ALU. Non-commutative operations, like subtraction and division, may make a specific memory the preferred one, while (isolated) commutative operations only prefer that opposite memories be employed.

The problem can have additional constraints, which include: having certain objects pre-assigned to a specific memory; requiring certain objects to be either in the same memory or in opposite memories as certain others; and limiting the total number (or size) of all objects assigned to either memory. Another type of constraint can restrict the number of accesses to either memory in the most time-critical parts of the program. Furthermore, it may improve

performance if a few frequently used objects are replicated in both memories.

Unfortunately, compilers capable of exploiting the memory bandwidth increase offered by dual memory banks are not generally available. Most of the research for optimizing compilers has been oriented towards general-purpose processors with less instruction-level parallelism and a friendlier ensemble of registers.

The aim of this paper is to present a new optimization algorithm capable of maximizing the benefit of dual memory banks. Our model is based on *Integer Linear Programming (IP)*. An IP formulation of any problem is simply a formal, algebraic way of expressing the problem, for which off-the-shelf solution procedures are available. We model all of the issues mentioned above in terms of linear equalities, whose combined effect is to specify the correctness criteria of any feasible solution. Since all the constraints are considered together by the IP, any solution can relate and trade off issues that arise, thus providing an integrated result.

An important idea is the way we approach memory assignment. Like other approaches [1-8] we assume that all data objects to be assigned to memory are known. However, we use detailed frequency information, provided by the front end, to favor objects appearing in blocks in the critical path. Our approach is global, considering all data objects in all blocks simultaneously. Issues relating to commutativity, or the irregular register structure of the machine, are taken into account when assigning objects to particular memories. Finally, mechanisms exist for dealing with pre-assigned objects, and for balancing the load between memories.

The remainder of the paper is organized as follows. In Section 2, we explore the problem in greater depth. Examples throughout this section will be based on two commercial DSPs: The Motorola DSP56000 (M56K) and the highly irregular SGS-Thomson ST18950 (SG950). In Section 3 we describe the related work. In Section 4 we show how the memory assignment problem can be modeled as an integer linear program. In Section 5, we present a series of simple examples to illustrate the flexibility of our approach, before presenting results for standard benchmarks in Section 6. Conclusions are drawn in Section 7 along with directions for future work.

2. A Closer Look at the Problem

The general memory assignment problem can be stated as the task of assigning data objects to one or both memories so that the performance of the final program is optimized and all constraints are satisfied. Though simple to state, a number of interrelated issues make finding a solution non-trivial. To understand these issues, the following series of brief examples are presented.

```
double FIR(in double A[], in double B[], in int tap)
int i;
double sum=0;

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

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

1. Example of Software Pipelining

A. Software Pipelining

Figure 1 shows C-like code for implementing an N-Tap FIR filter as well as a portion of the resulting assembly-language code for the M56K. Notice that instructions 1 and 3 are parallel DSP instructions. For example, instruction 3 multiplies the contents of registers X0 and Y0 and adds the resulting product to the contents of accumulator A, retrieves new operands from memories X and Y into registers X0 and Y0, respectively, and updates address registers R0 and R4 using auto-increment for each. The assembly-language is characteristic of the software pipelining necessary for producing the fast, dense code required by DSP applications. Also notice that instruction 3 constitutes the entire loop body. Two operands are concurrently fetched from two memories on the iteration prior to their actual use. This implementation is possible only because array A and array B were assigned to opposite memories. Assigning both arrays to the same memory would have doubled the loop body. Therefore, whenever possible, the compiler should ensure that data objects are assigned to memory in a way that best facilitates the type of software pipelining just described.

B. Register Connectivity and Operator Commutativity

Non-commutative instructions require their operands to appear at specific ports of the ALU and are especially affected by the degree to which registers connect specific ports of functional units and memories. Register connectivity varies from machine to machine. Some architectures are relatively symmetric in that every operand and accumulator register can access both memories and both sides of the ALU. This flexibility greatly simplifies the task of memory assignment, since operands can be readily

retrieved from either memory to appear at either port of the ALU. However, memory assignment is harder for more irregular architectures (like the ST950 and M56K).

For example, consider an architecture that has no direct path between both memories and both sides of the ALU. Now, consider the two non-commutative instructions, `A-B` and `C-A`, where variables `A`, `B`, and `C` reside in memory. If variable `A` is initially assigned to the right memory, it must first be loaded into one of the registers associated with the right memory, and then copied into one of the registers associated with the left side of the ALU. A similar problem exists if variable `A` is originally assigned to the left memory. In both cases, the only way to avoid an extra copy operation (and possibly an extra control step) is to duplicate A in both memories.

A comparable dilemma can also arise with commutative operations, if the machine has restricted register connectivity. For example, the expressions `A+B`, `B+C`, `A+C`, exhibit a "circular" dependency that makes it impossible to place single copies of `A`, `B`, and `C` in two memories so that operand pairs are always drawn from opposite memories. The rearrangement permitted by commutativity does not resolve the problem.

C. Data Duplication

Consider the high-level statement: `F[i] = F[i-1] + F[i-2]`. Notice how all three array references are to different elements of the same array. If the array is assigned to a single memory, each operand must be retrieved sequentially on separate steps - possibly increasing schedule length. Duplicating the array in both memories, however, allows both operands to be accessed in parallel on the same control step.

Unfortunately, duplication is expensive, as it requires twice as much memory to store both copies of the array. In practice, the limited memory capacity of most commercial DSPs limits the amount of data that can be duplicated. Furthermore, if the array is updated, as is the case here, an extra store operation is required to update both copies of the array. This may result in an increase in schedule length if the extra store cannot be performed in parallel with another ALU operation.

In general, duplication should only be performed when it leads to an overall improvement in performance.

D. Other Considerations

Constraints relating to the physical size of each memory may also exist. In order to keep performance high, designers of embedded systems, where DSPs are employed, favor the use of on-chip memory and avoid external memory as much as possible. This necessitates the judicious assignment of data if the capacity of each on-chip memory is not to be exceeded. Furthermore, certain program variables (usually arrays) may have to be assigned a priori to specific memories. In fact, some variables may have to be placed at specific locations that are accessed by other hardware components within the embedded system. At the lowest level, such decisions have nothing to do with the instruction set of the target machine.

For code blocks in critical paths, it is wise to balance the load between both memories. Each memory usually has a limited number of dedicated address and offset registers. Too many references to one memory may exhaust a memory's register resources and may even extend the schedule to accommodate one memory's activity. High address register usage may require frequent re-initializations of the memory's address and offset registers. These "extra" instructions also lengthen the schedule and degrade performance. By finding a balance between both memories this undesirable behavior can be curtailed to some degree.

E. Summary

To summarize, program variables should be assigned to memory in order to support simultaneous accesses. However, the use of duplication should only be performed when the performance improvement justifies the additional memory cost. If the register structure of the target machine is irregular, operands of non-commutative operations should be assigned to memory in a way that minimizes the number of additional copy instructions required, because of data appearing in the "wrong" memory. Finally, the on-chip memory capacities of the DSP must be balanced and cannot be exceeded, and some variables may already be assigned to, or prohibited from, specific memories.

3. Related Work

To date, relatively little work has been published on the dual-bank memory assignment problem. Most approaches employ a *graph-oriented* model of the problem. For example, in [6] [7] Sudarsanum and Malik develop an approach based on simulated annealing that is sensitive to irregular register structures. The algorithm labels the nodes of a constraint graph where each node represents a symbolic register and memory preference, and each edge represents dependencies and constraints between the references. Although this approach is able to produce good results, high run times can be an issue due to the nature of simulated annealing.

A simpler version of the problem is solved by Saghir et. al. [4] [5]. The approach assumes that no restrictions apply to the registers that can be used, and operates only on basic blocks. An undirected interference graph is formed where nodes represent program variables and edges indicate pairs of variables that should be assigned to opposite memories. Once constructed, the interference graph is searched using a greedy algorithm to partition the nodes into two sets, one for each memory, having minimal

dependence (fewest edges) among nodes within each set.

A limitation of using an interference graph is that it does not always provide a complete picture of the memory accesses that can occur in parallel during scheduling. This shortcoming is addressed in [8], where Zhuge et. al. employ a novel variable independence graph that is refined (by removing edges between memory accesses that cannot be scheduled in the same control step) to show potential parallel accesses that may occur later during scheduling.

Finally, in the work of Cho [1] et. al., traditional graph coloring and minimum-spanning tree algorithms with special constraints added to deal with issues arising from the non-orthogonal nature of the DSP are used to find a good memory assignment.

In general, the work presented here differs from the previous works in the following ways. First, a global optimization is performed for all blocks of code and all objects simultaneously. Detailed frequency information, provided by the front end, is used to favor objects appearing in blocks in critical paths. Issues relating to commutativity, or the irregular register structure of the machine, are taken into account when assigning objects to particular memories. In addition, mechanisms exist for dealing with preassigned objects, and for balancing the load between memories.

Unlike previous works, our model is not graph based. Rather, we model the problem of assigning objects to dual memories as an Integer Linear Program (IP). The primary advantage of the IP is that it allows different, and often conflicting, issues to be integrated and explored within a unified model. Our model integrates not one, but several different memory-assignment issues into the IP's objective function, all of which are optimized simultaneously. Moreover, the IP is lightening fast, requiring at most a few seconds to find an optimal solution to very large memory-assignment problems.

4. The Model

A. Operational Assumptions

Our model assumes that an analysis has been done by the front end to reveal the global histories (lifetimes and activities) of all data objects, both simple and complex. The information provided by the front end includes measures on how frequently a variable appears on the left (or right) side of a non-commutative (or unary) operator; the frequency with which a variable is updated; and, how frequently two variables appear together as operands of the same commutative operation. This frequency information is used to favor variables appearing in blocks in critical paths.

Determining the frequency information is made easier by the fact that most of the algorithms that use programmable chips have regular, well-understood behavior. Moreover, simulation, earlier experience with the algorithm, or allowing for the inclusion of branch probabilities and estimates of expected or maximum loop repetitions in the original source program may provide a precise performance profile.

B. Formulation

We model the dual memory assignment problem as an optimization problem with linear equality and inequality constraints. In our model, solution variables are indexed by i and j and are defined as follows:

Variable	Type	Meaning
x_i	0-1	if 1, variable i is assigned to the X memory
y_i	0-1	if 1, variable i is assigned to the Y memory
b_i	0-1	if 1, variable i is assigned to both X and Y memories
a_{ij}	0-1	if 1, both i and j are assigned to memory X only; 0 otherwise
b_{ij}	0-1	if 1, both i and j are assigned to memory Y only; 0 otherwise

The following are index sets of scalar objects:

Index Set	Meaning
N_i	number of "space units" (e.g., bytes, words, etc.) occupied by variable i. The units depend on the smallest data unit and most confined memory access mode of the machine.
L_i	total expected frequency of non-commutative (or unary) operators that access operand i on the left side
R_i	total expected frequency of non-commutative (or unary) operators that access operand i on the right side
U_i	total expected frequency of operations that update variable i
P_{ij}	total expected frequency of commutative binary operations that use operands i and j as paired operands
Max_x	capacity (in bytes) of X memory
Max_y	capacity (in bytes) of Y memory

L_i, R_i, U_i, and P_{ij} are constants derived directly from the usage of the variables in the original program and represent frequency-dependent numbers taken from the entire scope of the variable. Note that for operations that either consume or update a "variable" of size N(e.g., say an array) the effect or importance of the operation can be captured by multiplying the simple frequency by N_i when computing L_i, R_i, U_i, and P_{ij}.

C. Basic Constraints

In the constraints that follow, we refer to the "left" memory as the X memory and the "right" memory as the Y memory. Collectively, constraints 2 through 5 relate a_{ij} and b_{ij} to the other variables and ensure that a_{ij} and b_{ij} cannot both be equal to 1 at the same time. If

$a_{ij} = 1$, both variables i and j are found in the same (X) memory and cannot be drawn from opposite memories – notably on the same control step. A similar observation applies to i and j with regards to the Y memory if $b_{ij} = 1$.

1. A variable i must be assigned either to the X memory, the Y memory or both memories:
   $\forall i : \quad x_i + y_i + b_i = 1$
2. If variable i is assigned to the Y memory or both memories, it is never assigned *exclusively* to the X memory along with any other variable j
   $\forall i, j : \quad 2(1 - a_{ij}) \geq y_i + b_i + y_j + b_j$
3. If variable i is assigned to the X memory or both memories, it is not assigned to the Y memory:
   $\forall i, j : \quad 2(1 - b_{ij}) \geq x_i + b_i + x_j + b_j$
4. If variable i and variable j are both assigned to the X memory, they must appear together only in the X memory, forcing $a_{ij} = 1$:
   $\forall i, j \ where \ i < j \ : \quad x_i + x_j \leq 1 + a_{ij}$
5. If variable i and variable j are both assigned to the Y memory, they must appear together only in the Y memory, forcing $b_{ij} = 1$:
   $\forall i, j \ where \ i < j \ : \quad y_i + y_j \leq 1 + b_{ij}$
6. The total size of the variables assigned to a memory cannot exceed the size of the on-chip data memories:
   $\sum_i N_i(x_i + b_i) \leqslant Max_x$
   $\sum_i N_i(y_i + b_i) \leqslant Max_y$

D. Objective Function

$$min\left[\sum_i y_i \cdot L_i + \sum_i x_i \cdot R_i + \sum_i b_i \cdot U_i + \sum_{i,j} (a_{ij} + b_{ij}) \cdot P_{ij}\right]$$

The value of the objective function reflects the cost of having to insert extra register copy instructions or to delay processing in order to move data to or from memory. More specifically, the objective function seeks to minimize a weighted sum that reflects the cost of having an operand appear in the "wrong" memory possibly requiring an extra control step due to a register copy (first two terms), the cost of having to update an operand twice if it appears in both memories possibly an extra control step due to the additional store (third term), and, finally, the cost associated with fetching paired operands on separate control steps whenever those operands are only assigned to the same memory (an extra control step).

E. Additional Constraints

The memory assignment problem can have additional constraints, and are handled as follows:

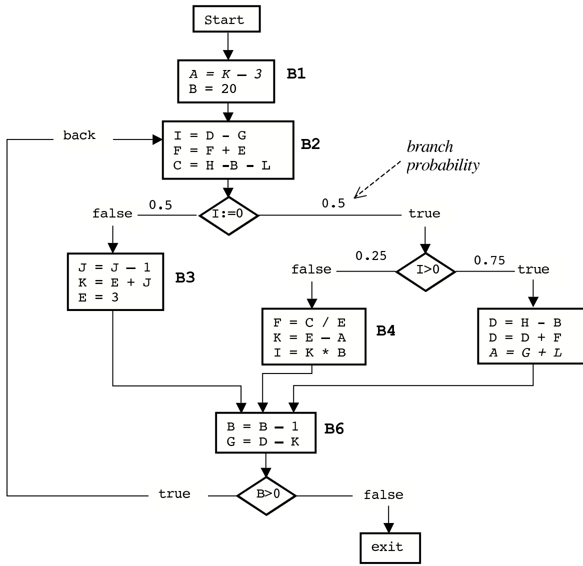
7. If variable i must be assigned a priori to the X memory:
   $y_i + b_i = 1$
8. If variable i must be assigned to the Y memory:
   $x_i + b_i = 1$
9. If variable i cannot be assigned to the X memory:
   $y_i = 1$
10. If variable i cannot be assigned to the Y memory:
    $x_i = 1$
11. A variable i cannot be duplicated in both memories:
    $y_i + x_i = 1$
12. Variables i and j must appear in the same memory:
    $b_i + b_j + a_{ij} + b_{ij} \geq 1$
13. Variables i and j must appear in opposite memories:
    $a_{ij} + b_{ij} = 0$

5. An Example

Before presenting results for standard benchmarks, we introduce a few small examples to illustrate the subtle ways in which the IP is able to trade-off the various memory assignment issues, all within a single integrated model. The examples that follow are based on the simple control-flow graph shown in Fig. 2. The nodes in the graph contain the text of statements, while the edges have been annotated with relative branch probabilities, and are used to estimate the execution frequency of each block in the program. These estimates, in turn, identify critical paths to favor variables appearing in blocks on critical paths. For example, variable `A` is updated in blocks 1 and 5 (i.e., U_A = 2). Block 1 appears outside the loop and executes once. If the loop in the control-flow graph executes 20 times, and the probabilities that the two if statements' conditions are true are 0.5 and 0.75, respectively, block 5 will execute approximately 7.5 times (i.e., 20 $\times$ 0.5 $\times$ 0.75). This information can now be used to reflect the importance of updating variable `A` with regards to the entire scope of the variable. For every block in which variable `A` is updated (in this case, blocks 1 and 5), we simply sum the expected execution frequencies of the individual blocks (1 + 7.5 = 8.5). This leads to a new value of U_A = 8.5, reflecting the relative importance of variable `A` in any IP solution. A similar update can be made for all i variables and constants (i.e., L_i, R_i, U_i, and P_i) that are part of the IP model.

Tables 1 and Table 2 summarize the activity of all of the variables present in the control-flow graph of Fig. 1. In Table 1, column 1 identifies each variable by name; column 2 identifies the size of each variable in bytes; columns 3 and 4 indicate the relative frequency with which the left (X) or right (Y) memory is the preferred one for a variable; while column 5 indicates the relative frequency with which a variable is updated.

We begin in example 1 with a fixed memory size of 22 bytes per memory. Figure 3(a) shows the final memory assignment produce by the IP. The grey shading shows instances where a variable has been duplicated and appears



2. Simple control-flow graph.

Variable	N_i (bytes)	L_i	R_i	U_i
A	2	0	2.5	8.5
B	4	20	27.5	21
C	2	2.5	0	20
D	4	40	0	15
E	1	2.5	2.5	10
F	1	0	0	22.5
G	2	0	20	20
H	7	27.5	0	0
I	1	0	0	22.5
J	2	10	0	10
K	4	1	20	12.5
L	1	0	20	0

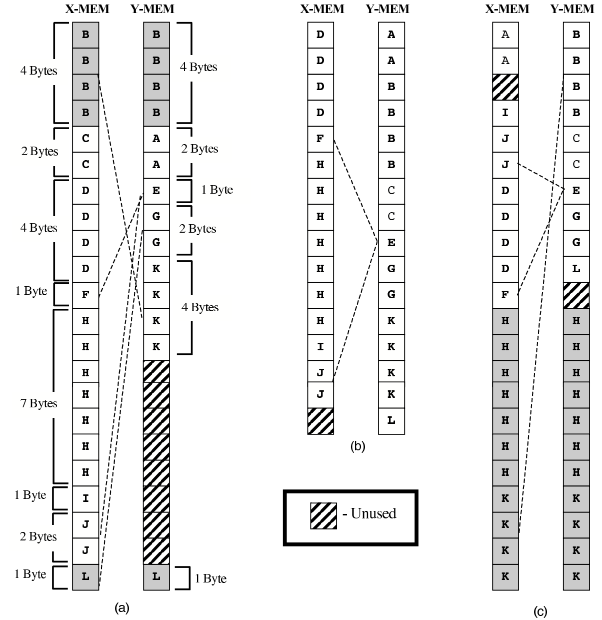
1. Variable sizes and activities.

in both memories. The bold variables correspond to those variables with memory preferences and highlight the IP's ability to satisfy as many of the preferences as possible. Finally, the dashed lines show where joint-operand pairs were successfully mapped to opposite memories.

Close examinations of Fig. 3(a) and Table 1 reveal that variable B is placed in both memories because it appears on the left and right side of a relatively large number of non-commutative operations. The reason behind the IP's choice to duplicate variable L, however, is more subtle. Variable L has a strong preference to appear in the right (Y) memory, as it appears on the right-hand side of a non-commutative operator in block 2. Along with variable G, it also appears on the right-hand side of a commutative operator in block 5. The Variable G, however, also has a strong preference for appearing in the right-hand memory as it appears on the right-hand side of a non-commutative operator in block 2. Clearly, assigning variable G or variable L to both memories can only satisfy the dilemma satisfactorily. However, since variable L is not updated (U_L=0) and variable G is updated in a block that executes frequently (U_G=20), the IP judiciously chose

Variable i	Variable j	P_{ij}
E	F	20
E	J	10
D	F	7.5
B	K	2.5
G	L	7.5

2. Pairs of variables involved in commutative operations.



3. Results for IP memory assignment.

to duplicate variable L.

In example 2, we reduce the size of the left memory to 15 bytes, and the size of the right memory to 16 bytes. This precludes any duplication of variables, as shown in Fig. 3(b). Notice that it is simply not possible to satisfy all of the memory preferences without duplication.

In example 3, we demonstrate the IP's ability to handle additional constraints imposed by the user. More specifically, we impose the following requirements on any IP solution:

- variable H must appear in the left (X) memory;
- variable A is not to appear in the right (Y) memory;
- variable K is to be duplicated;
- variable L must not be duplicated; and,
- variables F and E must appear in opposite memories.

Figure 3(c) shows that all of the previous constraints are satisfied by the solution found by the IP. Moreover, note that operand pairs E−F, E−J and B−K occur in different memories allowing them to be read simultaneously.

6. IP Execution Time

Solving an IP may take considerable time due to the fact that integer linear programming is NP-complete [2]. In

this Section, we report on experiments involving a large number of randomly generated problem instances and show that the proposed IP can be solved to find optimal solutions for large problems in very small amounts of time.

In total, the IP was tested using 48 randomly generated problem instances. To simplify the analysis, the various problem instances were divided into 3 classes, denoted by A, B, and C, according to their sizes. All problems were generated at random and subject only to the constraints in Section 4.3. In class A, each problem has 50 variables. The number of variables where the left memory is preferred is either 25% or 50%. Similarly, the number of variables where the left memory is preferred is either 25% or 50%. Similarly, the number of variables where the right memory is preferred, and the number of variables that require updating is either 25% or 50%. Finally, the number of commutative binary operations that use operands i and j as paired operands is either 50% or 75%. As all combinations exist, class A contains 16 problem instances. Classes B and C are similar to class A, but contain more variables. Class B contains 1000 variables, while class C contains 5000 variables.

The IP was solved using the LINDO [10] solver which ran on a Windows 2000 system with 2.4-GHz Intel Pentium 4 Processor and 512 bytes of RAM.

Tables 3-5 show the run times in seconds for each problem instance in class A, B, and C, respectively. The results show that the run times are uniformly low, even for very large problem instances with thousands of variables. Although longer compile times for embedded applications are often tolerated, the speed with which optimal solutions for the proposed model can be found is important consideration. In general, the time required to find solutions for linear programs grows exponentially with respect to the number of variables int the problem formulation and linearly with respect to the number of problem constraints. As varying |L|, |R|, |U|, and |P| does not affect the number of variables in the original formulation the times for problems in each class (table) remain the same or differ only by a small amount. During further experimentation, we observed that this trend continued for even larger problems with ten's of thousands of variables, thus speaking to the efficiency of the proposed IP model.

A. Results Found Using Actual Benchmarks

The effectiveness of the IP model was also tested using various kernels from the DSPstone benchmark suite [9], along with some applications. These benchmarks are also used when establishing the effectiveness of the models proposed in [1-8]. In the embedded systems community, DSPs are typically evaluated using loops that form the core of many common signal-processing algorithms. Each kernel contains one or more loops with operations on two or more global arrays. For each kernel (and application), three variants were considered. In the first version, data objects were assigned explicitly only to one (i.e., the X) memory bank. In the second version, data objects were partitioned between the X and Y memories. Finally, in the third version all data objects were duplicated in both (X and Y) memories.

Based on the memory assignment information, code was produced for the Motorola DSP56000. Table 6 shows the performance results obtained for the selected DSPstone kernels (and applications). For each version, the first column shows the total number of clock cycles executed (T_C); the second and third columns show the number of clock cycles resulting from accesses to the X (X_C) and Y (Y_C) memory banks, respectively. It can be observed that the best overall performance gain is achieved when the IP is used to partition the data into different memories. The number of memory access cycles for both the X and Y memory are reduced by approximately 54%, while the reduction in the total number of clock cycles ranges from 7% to 42%. These results are optimal [3], and are a direct consequence of the fact that most of the kernels (and applications) provide ample opportunities to exploit instruction-level parallelism through judicious memory assignments. Nonetheless, Table 6 indicates that for some programs (e.g., fib2) the best software solution for providing parallel accesses is to duplicate the data in

Problem	$\|variables\|$	$\|L\|$	$\|R\|$	$\|U\|$	$\|P\|$	Run Time
A-1	50	12	12	12	25	0.1s
A-2	50	25	12	12	25	0.1s
A-3	50	12	25	12	25	0.1s
A-4	50	25	12	12	25	0.1s
A-5	50	12	12	25	25	0.1s
A-6	50	25	25	25	25	0.1s
A-7	50	12	12	25	25	0.1s
A-8	50	25	12	25	25	0.1s
A-9	50	12	25	12	37	0.1s
A-10	50	25	12	12	37	0.1s
A-11	50	12	12	12	37	0.1s
A-12	50	25	25	12	37	0.1s
A-13	50	12	12	25	37	0.1s
A-14	50	25	12	25	37	0.1s
A-15	50	12	25	25	37	0.1s
A-16	50	25	25	25	37	0.1s

3. Characteristics of Class A problems.

Problem	$\|variables\|$	$\|L\|$	$\|R\|$	$\|U\|$	$\|P\|$	Run Time
B-1	100	25	25	25	50	1.0s
B-2	100	25	25	25	75	2.0s
B-3	100	25	25	50	50	2.0s
B-4	100	25	25	50	75	1.0s
B-5	100	25	50	25	50	1.0s
B-6	100	25	50	25	75	2.0s
B-7	100	25	50	50	50	2.0s
B-8	100	25	50	50	75	1.0s
B-9	100	50	25	25	50	1.0s
B-10	100	50	25	25	75	2.0s
B-11	100	50	25	50	50	1.0s
B-12	100	50	25	50	75	1.0s
B-13	100	50	50	25	50	2.0s
B-14	100	50	50	25	75	2.0s
B-15	100	50	50	50	50	1.0s
B-16	100	50	50	50	75	1.0s

4. Characteristics of Class B problems.

Problem	$\|variables\|$	$\|L\|$	$\|R\|$	$\|U\|$	$\|P\|$	Run Time
C-1	200	50	50	50	100	7.0s
C-2	200	100	50	50	100	9.0s
C-3	200	50	100	50	100	6.0s
C-4	200	100	50	50	100	9.0s
C-5	200	50	50	100	100	7.0s
C-6	200	100	100	100	100	9.0s
C-7	200	50	50	100	100	8.0s
C-8	200	100	50	100	100	12.0s
C-9	200	50	100	50	150	7.0s
C-10	200	100	50	50	150	8.0s
C-11	200	50	50	50	150	7.0s
C-12	200	100	100	50	150	10.0s
C-13	200	50	50	100	150	7.0s
C-14	200	100	50	100	150	10.0s
C-15	200	50	100	100	150	8.0s
C-16	200	100	50	100	150	9.0s

5. Characteristics of Class C problems.

6. Results for DSPstone Kernels and Apps

	One Memory			**Partitioning**			**Duplication**		
Kernel/ Applc'n	T_C	X_C	Y_C	T_C	X_C	Y_C	T_C	X_C	Y_C
fir2dim	4226	1922	0	2426	1011	911	2426	1011	1011
dotproduct	618	201	0	416	101	100	416	101	101
convolution	416	203	0	212	102	101	212	102	102
lms	516	172	0	388	67	105	388	99	106
biquad n sect	516	172	0	388	67	105	440	99	97
n complex up.	2820	800	0	2420	400	400	2420	600	400
mat10x10	6916	2100	0	4516	1100	100	4516	1100	100
mat1x3	56	21	0	38	9	12	38	12	12
fft1024	60407	12306	0	78154	30715	29692	121160	40955	39932
n real up.	228	64	0	212	32	32	212	48	32
app1	80	30	0	60	20	10	60	20	20
fib1	160	70	0	100	40	30	120	50	50
fib2	32	12	0	28	6	6	24	8	8

both memories. However, it is clear from Table 6 that there is no benefit in duplicating all data indiscriminately when comparable or superior performance could be obtained with far less memory by judiciously allocating arrays to the appropriate memory banks. In general, any gain in performance obtained by duplication must be weighed against the increase in memory cost.

7. Conclusions

In this paper, we have presented a novel methodology, based on integer linear programming, for assigning data objects to dual-bank memories. Our approach is global, and special effort is made to identify those objects that could potentially benefit from an assignment to a particular memory, or perhaps both memories. Issues relating to commutativity, or the irregular register structure of the machine, are taken into account when assigning objects to particular memories. In addition, mechanisms exist for dealing with preassigned objects, and for balancing the load between memories. To the best of our knowledge, ours is the only model that we are aware of that optimizes 4 different objectives simultaneously when assigning data to memory. The primary advantages of the IP approach is its speed, ability to produce an optimal solution for the model, and the ease with which different, and often conflicting, issues can be integrated and explored all within a unified model.

Our experimental results show that our method is able to find a good memory assignment. When applied to typical DSP kernels and some applications, we were able to reduce the number of memory cycles by approximately 54% and the total number of cycles by 7% to 42%.

In the future, we plan to extend the existing model to consider architectures with more than two memories. We also plan to integrate scheduling of ALU and load-store operations into the current model.

References

[1] J. Cho, Y. Paek, and W. Whalley, "Efficient Register and Memory Assignment for Non-orthogonal Architectures via Graph Coloring and MST Algorithms", Proc. of the International Conference on LCTES and SCOPES, Berlin, Germany, 2002.

[2] M. Garey and D. Johnson, "Computers and Intractability", A Guide to the Theory of NP-Completeness, New York: W.H. Freeman and Company, 1979.

[3] Benchmark Programs, DSP56000/DSP56001 Digital Signal Processor User's Manual, Motorola INC., Semiconductor Products Sector, DSP Division, Austin Texas.

[4] M. Saghir, P. Chow, and C. Lee, "Exploiting dual data-memory banks in digital signal processors," Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 30(5), pp., 234-243, 1996.

[5] M. Saghir, P. Chow, and C. Lee, "Automatic Data Partitioning for HLL DSP Compilers," Proc. of the 7th International Conference on Signal Processing Applications and Technology, pp., I-658-664, 1995.

[6] A. Sudarsanum and S. Malik, "Memory bank and register allocation in software synthesis for ASIPs," International Conference on Computer-Aided Design, pp., 388-392, 1997.

[7] A. Sudarsanam and S. Malik, "Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs," Journal of ACM Transactions on Automation of Electronic Systems (TOADES), 5, pp., 242-264, 2000.

[8] Q. Zhuge, B. Xiao, and E. Sha, "Variable Partitioning and Scheduling of Multiple Memory Architectures for DSP," Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2002.

[9] V. Zivojnovic, J. Velarde, C. Schaeger, and H. Meyr "DSPstone - A DSP oriented Benchmarking Methodology," In proceedings of the 6th International Conference on Signal Processing Applications and Technology, 1994.

[10] Lindo Systems Inc. 1415 North Dayton Street, Chicago, IL 60622, USA.