2005

Heuristics for memory access optimization in embedded processors

Saravanan Subramanian
Louisiana State University and Agricultural and Mechanical College

HEURISTICS FOR MEMORY ACCESS OPTIMIZATION IN EMBEDDED PROCESSORS

A Thesis
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillments of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

By

Saravanan Subramanian
B.E., University of Madras, India, 2002
December, 2005

Acknowledgements

I would like to thank my major professor Dr. Ramanujam Jagannathan for his support, patience, guidance, and his thought provoking suggestions provided throughout the course of this research. His ability to express the complicated problems as simple ones provided me with the motivation and direction.

I express my gratitude to the members of my committee: Dr. Vaidyanathan Ramachandran and Dr. Seung-Jong Park for their valuable suggestions and advice. Also, I would like to thank Dr. Jinpyo Hong for his inputs and ideas.

I dedicate this work to my family members for their prayers, encouragement and financial support. I would like to thank all my friends at LSU for their assistance, wishes, support and encouragement throughout my research, and also throughout the course of my graduate study at LSU.

Table of Contents

List of Tables

List of Figures

Abstract

Digital signal processors (DSPs) such as the Motorola 56k are equipped with two memory banks that are accessible in parallel in order to offer high memory bandwidth, which is required for high-performance applications. In order to make efficient use of the memory bandwidth offered by two or more memory banks, compilers for such DSPs should be capable of appropriately partitioning the program variables between the two memory banks and scheduling accesses. If two variables can be accessed simultaneously, then it is essential to have these two variables assigned to two different memory banks. Also if these two variables are in different banks, then instead of using two separate instructions for accessing the variables, both the accesses can be encoded into a single instruction, thereby reducing the code size as well. An efficient heuristic for maximizing the parallel accesses in DSPs with dual memory banks is proposed and evaluated. The heuristic is shown to be very effective on several examples.

Architectures like the M3 DSP have a group memory for the single-instruction multiple-data (SIMD) architecture, for which addressing an element of the group means to access all the elements of that group in parallel, so there is no need for separately addressing each element of the group. Given a variable access sequence for a particular code, instead of separately accessing each one of the variables, if the variables are grouped then the number of memory accesses can be reduced as per SIMD paradigm. An efficient way of forming groups can significantly reduce the memory accesses. Two solutions for this problem are presented in this thesis. First, a novel integer linear programming formulation for forming the groups, thereby reducing the number of memory accesses in DSPs with SIMD architecture is presented. Second, a heuristic based

on the solution for optimizing multiple memory bank accesses is presented and evaluated for this problem. Results on several graphs show the effectiveness of the heuristic.

1. Introduction

Digital signal processors (DSPs) such as the Motorola 56k and the Analog Devices ADSP- 210x series are equipped with two memory banks (frequently referred to banks X and Y) that are accessible in parallel in order to offer high memory bandwidth, which is required for high-performance applications. In order to make efficient use of the memory bandwidth offered by two or more memory banks, compilers for such DSPs should be capable of appropriately partitioning the program variables between the two memory banks and scheduling accesses. For instance, consider the line of code

a = b + c;

From this line of code it is clear that variables b and c can be accessed in parallel. If these two variables are assigned to the same memory bank, then the variables b and c must be accessed one after the other (i.e., sequentially), thereby masking the parallelism that exists. Therefore to exploit the parallelism that exists between these two variables, it is essential to have these two variables assigned to two different memory banks. Also if these two variables are in different banks, then instead of using two separate instructions for accessing variables b and c, both the accesses can be encoded into a single instruction, thereby reducing the code size as well.

Dual memory bank DSPs are poorly exploited by compilers due to lack of suitable variable partitioning techniques. So, one objective of this research work is to develop a suitable variable-partitioning technique for dual memory bank DSPs.

DSPs such as M3-DSP are Single-Instruction Multiple-Data (SIMD) architectures**.** The SIMD paradigm refers to the concept of performing the same type of

computation (instructions) simultaneously on different data. In the case of the M3-DSP, there are 16 data path slices. In order to provide an effective use of all the data path slices in parallel, the memory is organized as a group memory. Here addressing one 16-bit data word means addressing an entire group of 16 such words.

The M3-DSP is organized as a very long instruction word (VLIW) architecture. Here, two successive VLIWs can be reused and do not need to be stored as instructions in memory. In order to provide an effective use of this method in loops the M3-DSP also contains an instruction cache for up to four VLIW instructions. In order to achieve high performance and code quality, it is essential that code generation makes use of these special architectural features such as SIMD and VLIW. Lorenz et al. [14] discuss one such technique that makes use of both these architectural features. The technique is subdivided into two phases. In the first phase, referred to as *horizontal address assignment*, the variables are partitioned into groups such that the same group is used as much as possible before using an another group, thereby making use of M3-DSPs SIMD group memory architecture. This reduces the number of memory accesses, and thereby the code size is reduced and the performance is improved. This research work includes a heuristic and an ILP formulation for addressing this optimization problem.

In the second phase the address generation instructions for addressing all the groups are optimized for a given memory layout. A heuristic proposed in [14] is based on the concept of maximizing the reuse of address generation instructions which reside in a VLIW of the instruction cache, thereby making use of the M3-DSPs VLIW instruction cache architecture. This optimizes the address generation instructions for a given memory layout.

1.1 Objectives and Thesis Overview

It has been mentioned in [12] that an efficient partitioning of the variables used in a processor application into two memory banks, results in an increased memory bandwidth utilization and high code quality. However the heuristic proposed in [9] to address this problem, incorporated a rigid assumption that there is an even distribution of variables in the two memory banks, which masks the possibility of computing a more optimal solution if a certain degree of imbalance was tolerated. This possibility has been investigated in [15]. However the heuristic proposed in [15], masks some other possibilities, which could further benefit to the optimal solution. So, one of the objectives of this research work is to investigate those possibilities, thereby computing an even more optimal solution. It has been discussed in Chapter 2.

It has been discussed in [14] that horizontal address assignment problem is a graph partitioning problem involving multiple banks rather than just two banks as in the previous case and finding an efficient partitioning of the variables into multiple banks will reduce the number of memory accesses. So the other research objective is to exploit the benefit and more optimal solution that could be achieved by using the heuristic discussed in previous Chapter 2, for solving this multiple bank partition problem. A heuristic and an ILP formulation which addresses this problem has been discussed in Chapter 3, Sections 3.1 and 3.2 respectively. Chapter 4 presents a summary of the results and points to further possible research.

2. Variable Partitioning Problem

Effective partitioning of the variables of a program is essential to the effective use of high memory bandwidths available in digital signal processors (DSPs) with multiple memory banks. It is well known that the problem of finding an optimal partition of a set of nodes of a weighted graph into two sets X and Y such that the sum of the edges between the sets X and Y is minimum (referred to as the graph bisection or the mincut problem) is NP-complete [4,21]. The same is true for the problem of maximizing the sum of the edges between X and Y.

A heuristic for this problem that uses swapping of nodes has been proposed by Kernighan and Lin [9]. The Kernighan-Lin heuristic assumes that at any given point of time during the execution of the heuristic, the number of nodes in the sets X and Y differ by at most one; this assumption may prevent the possibility of computing a solution perhaps closer to the optimal solution. Allowing a certain degree of imbalance between the sets X and Y could lead to better solutions [4]. However, if the case in which any amount of imbalance between sets X and Y is tolerable were to be valid, then the possibility of a scenario wherein all nodes migrate to a particular set cannot be ruled out, and safeguarding against that scenario needs some thought and consideration. A heuristic that attempts to solve this problem by incorporating the concept of transferring nodes from set X to set Y and vice-versa, rather than swapping nodes on the basis of a one-to-one mapping between the two sets, has been discussed in [15]. This chapter considers and presents several improvements to the heuristic in [15]. The reader is referred to [15] for details of that heuristic.

First, the heuristic in [15] does not consider the following possibility:

Transfer of some of the nodes that are marked without making any transfers, but for violating the size constraint, could still add to the gain of the graph.

We have modified the heuristic in [15] to take care of this issue, by marking a node only when there is a transfer of that node from one partition to an another one.

Secondly, the we have observed the following case:

Some node transfers could possibly still benefit to the overall gain, even though the maximum cumulative gain has become zero.

We have taken care of this, by continuing with the transfers, even though the maximum cumulative gain becomes 0, till the nodes that are being transferred in the current iteration are a subset of nodes that had already been transferred with the maximum cumulative gain being 0.

Thirdly, [15] does not use a tie-breaking function in the case of multiple nodes that are candidates at any point. We have introduced a tie-breaking function, to break a tie, if there are two or more nodes with the same maximum gain among the set of nodes that are ready to be transferred.

2.1 The Improved Heuristic

We use the following definitions. The total cost (Overall Gain) of the two sets X and Y, denoted by T = cost(X, Y), is defined as the sum of the weights of the edges whose end-points belong to distinct partitions. Thus T= $\sum$ {w(e) where e is an edge between some node in X and some node in Y}. The *internal cost* of a node u (denoted I(u)) is defined as the sum of weights of all edges incident at u whose other end-point is in the same set as u. Let I(x) denote the internal cost of x in X; thus, I(x) = $\sum$ w(x, x') for

other x' in the set X. The internal cost I(y) for nodes y in the set Y is defined similarly. The *external cost* of a node u (denoted E(u)) is defined as the sum of weights of all edges incident at u whose other end-point is not in the same set as u. Let E(x) denote the external cost of x in X; thus, E(x) = ∑ w(x, y) for other y not in the set X; with just two sets X and Y, this means that y belongs to the set Y. The external cost E(y) for nodes y in the set Y is defined similarly. We define D(u) for every node u as I(u)-E(u).

With just two sets X and Y, T = ∑ E(x) for x belonging to X; also T = ∑ E(y) for y belonging to Y. Now we consider the effect of transferring a node p from its current partition to the other partition. We say node a is adjacent to node b if there is an edge (a,b) in the graph. There are two cases to consider, which we discuss in detail.

2.1.1 Node p Is Initially in X

In this case, node p is being transferred from X to Y. As a result of this transfer, the cost of the partitions changes; let the resulting cost of the partitions (X - {p},Y + {p}) be referred to as newT. As a result of the transfer, the internal cost of p becomes its external cost, and the external cost of p becomes its internal cost. Thus newT T + I(p) - E(p).= T + D(p); one can view D(p) as the gain due to transferring p referred to as gain(p).

The internal cost I(x) for nodes in X that are adjacent to p are updated as follows: I(x) = I(x)-w(x,p) for all nodes x in the set X such that there is an edge (x,p) in the graph. The external cost E(x) for nodes in X that are adjacent to p are updated as follows: E(x) = E(x)+w(x,p) for all nodes x in the set X such that there is an edge (x,p) in the graph. Thus D(x) for all nodes for all nodes x in the set X that are adjacent to p are updated as D(x)=D(x)-2w(x,p). We can define the effects on nodes y in the set Y analogously. The

internal cost I(y) for nodes in Y that are adjacent to p are updated as follows: I(y) = I(y)+w(y,p) for all nodes y in the set Y such that there is an edge (y,p) in the graph. The external cost E(y) for nodes in Y that are adjacent to p are updated as follows: E(y) = E(y)-w(y,p) for all nodes y in the set Y such that there is an edge (y,p) in the graph. Thus D(y) for all nodes for all nodes y in the set Y that are adjacent to p are updated as D(y)=D(y)+2w(y,p).

2.1.2 Node p Is Initially in Y

In this case, node p is being transferred from Y to X. The analysis in this case is similar to the previous case. As a result of this transfer, the cost of the partitions changes; let the resulting cost of the partitions (X + {p},Y- {p}) be referred to as newT. As a result of the transfer, the internal cost of p becomes its external cost, and the external cost of p becomes its internal cost. Thus newT T + I(p) - E(p).= T + D(p); one can view D(p) as the gain due to transferring p referred to as gain(p). The internal cost I(y) for nodes in Y that are adjacent to p are updated as follows: I(y) = I(y)-w(y,p) for all nodes y in the set Y such that there is an edge (y,p) in the graph.

The external cost E(y) for nodes in Y that are adjacent to p are updated as follows: E(y) = E(y)+w(y,p) for all nodes y in the set Y such that there is an edge (y,p) in the graph. Thus D(y) for all nodes for all nodes y in the set Y that are adjacent to p are updated as D(y)=D(y)-2w(y,p). We can define the effects on nodes x in the set X analogously.

The internal cost I(x) for nodes in X that are adjacent to p are updated as follows: I(x) = I(x)+w(x,p) for all nodes x in the set X such that there is an edge (x,p) in the graph. The external cost E(x) for nodes in X that are adjacent to p are updated as follows: E(x) =

E(x)-w(x,p) for all nodes x in the set X such that there is an edge (x,p) in the graph. Thus D(x) for all nodes for all nodes x in the set X that are adjacent to p are updated as D(x)=D(x)+2w(x,p).

2.1.3 Variations

1. If transferring a node 'p' from one partition to the other partition, say for example from X to Y, exceeds the size limit in the Y partition, then as per the heuristic in [15] all the nodes in the X partition are marked, thereby making them ineligible from being transferred to the Y partition. Thus the nodes in X partition are marked, even though they are not transferred to Y, but for violating the size constraint in Y. This masks the possibility of nodes in X, increasing the Overall Gain, by being transferred to the partition Y, without violating the size constraint. We have modified the heuristic to take care of this possibility, by marking a node only when there is a transfer of that node from one partition to an another one.
2. From the heuristic in [15], it is clear that the repeat until loop will execute as long as the maximum cumulative gain is positive. The loop will terminate when the maximum cumulative gain becomes Zero or negative. But we know that the maximum cumulative gain of Zero will not harm the Overall Gain. So we have considered the possibility of Zero Gain as a special case, by continuing with the transfers even if the maximum cumulative gain is Zero, till the nodes that are being transferred in the current iteration are a subset of nodes that had already been transferred with the maximum cumulative gain = 0.

These two variations have been implemented and tested and we obtained significant improvement in the Overall Gain. The heuristic with these two variations is given next.

2.1.4 Heuristic

1. Compute the Cost for the Initial partition X and Y
2. terminateList =NULL, terminate = # of nodes;
3. repeat
   {
4. Unmark and unticket all the nodes u in XUY
5. Compute the gain D(u) for all nodes u in XUY
6. while(# of unmarked and unticketed nodes != 0)
   {
7. Find an unmarked and unticketed node p_i in X U Y maximizing the gain.
8. ***If (transferring p_i from current partition to the other partition exceeds size constraint in the other partition)***
   {
9. ***set ticket = 1 for all nodes in the current partition.***
   }
10. else
   {
11. ticket = 0 for all nodes in the other partition.
12. Mark the node p_i.
13. Update D(u) for all the unmarked nodes u as though p_i had been transferred, and save gain(p_i) and p_i.
   }
14. }endwhile;

15. Pick 'k' for which Max.Cumulative.Gain = $\sum$gain(p_t) for t =1 to k is maximum where 0<=k<=i;

16. ***If (Max. Cumulative.Gain == 0)***
   {
17. ***terminate = k;***
18. ***for (t =1 to k)***
   {
19. ***if (p_t $\in$ terminateList)***
   {
20. ***terminate = terminate -1 ;***
   }
21. ***else***
   {
22. ***terminateList = terminateList + p_t***
   }
23. ***}***
   }

Figure 2.1 Heuristic for Variable Partitioning (fig. cont'd.)

```
24.     if (terminate != 0 and  Max.Cumulative.Gain >= 0)
                {
25.                     Update set X with new nodes.
26.                     Update set Y with new nodes.
27.                     T = T + Max.Cumulative.Gain;
28.             }
```

29. } ***until (Max.Cumulative.Gain <0 or terminate == 0)***

If two nodes A, B have the same gain, then the following Tie Breaking function is used:

```
TieBreaking function (A,B)
     {
30.  if( gain > 0 )
31   {
32     if ((E(A) > E(B))
33     return A;
34     else return B.
     }
35  else
    {
36     if ((E(A) > E(B))
37     return B;
38     else return A.
39   }
40   }
```

2.1.5 Explanation

For the initial partition, the Initial Cost is determined by adding the weights of the edges whose end-points are in different partitions. In line 5, the node gains D(u) are determined using the expression I(u) - E(u) and the node with the maximum gain value D(u) (say p) is chosen. If transferring the node p from current partition to other partition leads to exceeding the size limit in the other partition, then all the nodes in the current partition are set with a flag named “ticket” (lines 8-9). This disables those nodes from being transferred.

On the other hand, if there is no size limit violation on transferring the node p from current partition to the other partition, then the ticket flag is reset (ticket = 0) for all

the nodes in other partition (line 11). This makes the nodes in the other partition eligible for being transferred. Then the node p is set with the marker flag (marker =1) (line 12).

In line 13, D values for those unmarked nodes are updated using the formula as though p has been transferred. Gain of the node p and the node p are saved in line 13.The whole process of finding the node with maximum gain, checking for size violation, setting the ticket flag, marking the node, updating the D value are repeated till there exist some node which is unmarked and the ticket flag is not set (lines 6-14).

When there are no more unmarked and unticketed nodes, find the set of nodes whose transfer gives the maximum gain (Max.Cumulative.Gain) (line 15). If the maximum cumulative gain is positive then the sets X and Y are updated with those set of nodes and the new Overall Gain (T) value is obtained.

If the maximum cumulative gain is equal to zero, it is checked if the nodes that must be transferred to get the maximum cumulative gain are already in the terminateList. If so then a variable "terminate" is decremented, or else that node is added to the terminateList. If all the nodes to be transferred (i.e p_t's) are in the list then the "terminate" variable will have value Zero (lines 16- 23) and the while loop (line 3) terminates. This means that the nodes that are already transferred for no improvement in the gain are to be transferred again, and the subsequent passes will require the same nodes to be transferred again, which marks the end of the execution and the while loop terminates.

If the "terminate" variable value is not zero, then there are some new nodes that are to be transferred ,which might possibly increase the Overall Gain (T) value and the partition sets X and Y are updated with the new nodes and the T value is also updated (lines 24 – 28).

If two nodes have the same gain and if the gain is positive, then the node with higher external Cost is transferred, or else the node with lower external cost is transferred (lines 30–40).

2.2 Examples

2.2.1 Example Showing Improvement in the Overall Gain due to the Variation.1

Consider the Interference Graph given below in Figure.2.2. Let the partition Size be 3 and initial partition be partition_1 = {a,b,c} and partition_2 = {d,e}.

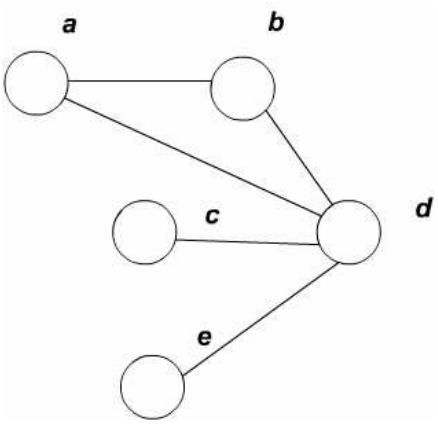

Figure 2. 2: Example Interference Graph to Show Improvement in the Overall Gain due to the Variation.1

Initial Cost for this partition is 3, as per the definition described before. For every node initial gain value D(u) is computed and is shown below.

Table 2. 1: Initial Gain Computation of all Nodes in Figure 2.2

Node 'u'	Internal Cost I(u)	External Cost E(u)	D(u)
a	1	1	0
b	1	1	0
c	0	1	-1
d	1	3	-2
e	1	0	1

The node with the maximum gain is ‘e’. Now, transferring node ‘e’ to the other partition exceeds size constraint and so all the nodes in Y partition are set with ticket flag (line 9). The next node with the maximum gain value is ‘a’ and it is marked (line 12). Assuming that node ‘a’ has been transferred the remaining node gains are updated as per the definition.

Table 2. 2: Updated Gain Computation for Transferring Node ‘a’

NewD(b)	D(b) – 2*w(b,a) = 0 - 2*1 = -2
NewD(c)	D(c) - 2 *w(c,a) = -1 – 2*0 = -1
NewD(d)	D(d) + 2*w(d,a) = -2 + 2*1 = 0
NewD(e)	D(e) + 2 *w(e,a) =1 +2*0 = 1

Now the next node with maximum gain is ‘e’. Once again, the gains of the remaining nodes are updated and the whole process is repeated till there is some unmarked and unticketed node. When this loop terminates, the nodes whose transfers yield the maximum cumulative gain is determined and those nodes are transferred, and the Overall Gain (T) is updated. This process continues till the maximum cumulative gain becomes negative or when the same set of nodes is transferred in subsequent passes for no improvement in the Overall Gain.

The Overall Gain for this graph computed by this heuristic is 4 and the optimal partition obtained is `bce` in one partition and `ad` in the other. But the Overall Gain that is obtained without these variations in the heuristic is only 3.

2.2.2 Example Showing Improvement in the Overall Gain due to the Variation.2

Now let us consider an example in Figure.2.3 for which the Variation.2 produces significant results. In Figure 2.3 the outputs of the operators numbered 7,8,9,10,11,12,13,14,15,16,17 are represented by the variables A,B,C,D,E,F,G,H,I,J,K.

From this data dependency graph, a folded interference graph is constructed which is not shown here due to space constraints. The heuristic was implemented on this interference graph and it has been observed that the maximum cumulative gain value increases, even after taking a value zero.

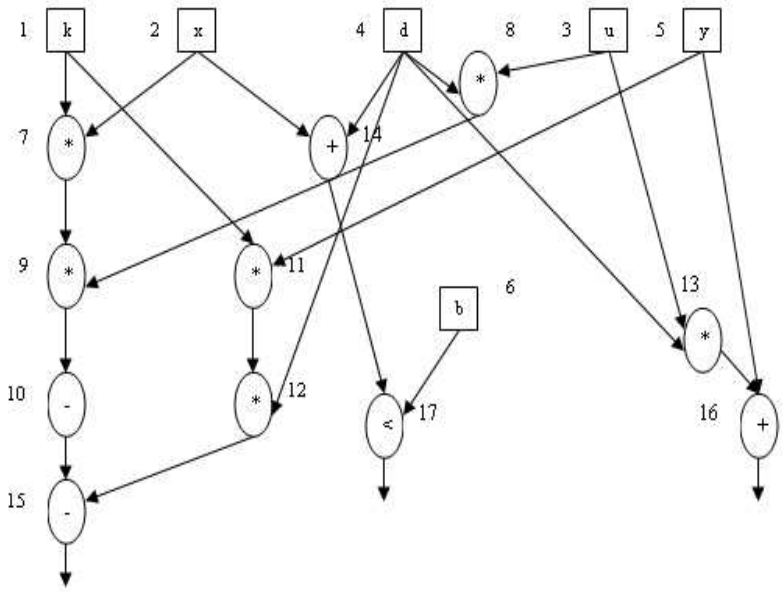

Figure 2. 3: Example Data Flow Graph to Show Improvement in the Overall Gain due to the Variation.2

Table 2.3 shows the maximum cumulative gain value and the nodes that are to be transferred in each of the iteration. It has been observed that at iteration.4 maximum cumulative gain value is 2, even after being 0 at iteration.2, making the Overall Gain to

Table 2. 3: Max.Cumulative Gain value for each Iteration of the Heuristic on the Interference Graph of Figure 2.3

Iteration	Max. Cumulative Gain	Nodes to be transferred
1	14	dyCJHu
2	0	IdbB
3	0	k
4	2	uKkHd
5	0	E

be 59. If the execution had stopped at iteration 2 because the maximum cumulative gain being 0, then we would have ignored this possibility of improvement in the gain and Overall Gain would have been just 57. After the 4th iteration, there is no improvement in the gain and the same set of nodes are transferred again and again and thereby the execution terminates at the iteration.6.

2.3 Experimental Results

Consider the Interference graph below with 5 nodes with initial partitions partition_1 = {a,b,c} and partition_2 = {d,e}. The heuristic was implemented on this graph and the results are tabulated as given below.

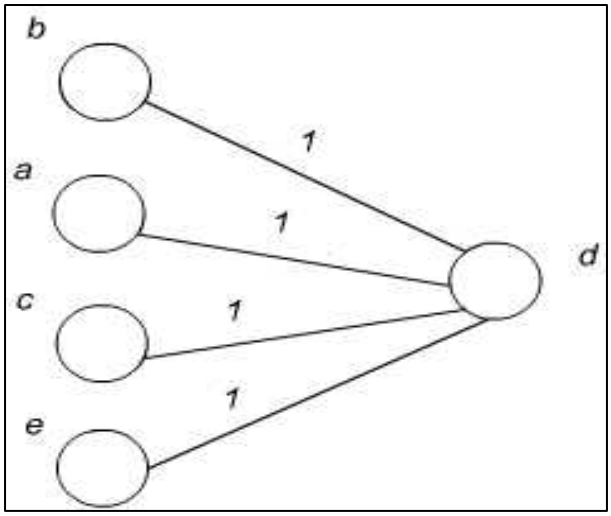

Figure 2. 4: Example Interference Graph

Table 2. 4:Overall Gain and the Final Partition for Various values of S of Figure 2.4

S.No	S	Overall Gain	Final partition
1	3	3	eab, dc
2	4	4	abec,d
3	5	4	abec,d

The table above shows the Overall Gain and the Final partition for various S. It can be observed that once the Maximum possible Overall Gain is achieved, any further increase in the S value will not have any effect on the Overall Gain.

Consider the Interference graph below with 10 nodes with initial partitions partition_1={B,C,A,b,x} and partition_2 ={k,d,y,u,D}.Let the weights of all the edges be 1. The heuristic was implemented on this graph and the results are tabulated (Table 2.5).

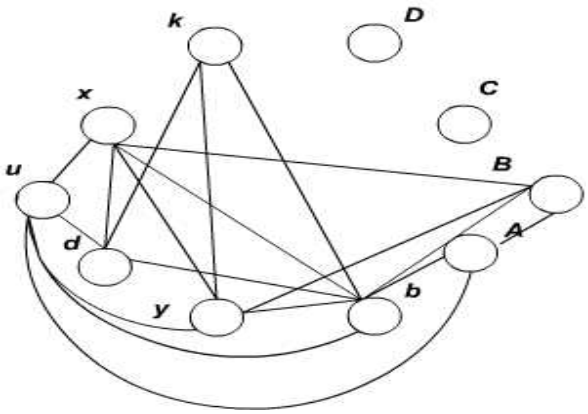

Figure 2.5: Example Interference Graph

Table 2. 5: Overall Gain and Final Partitions for Various values of S of Figure 2.5

S.No	S	Overall Gain	Final partition
1	6	13	AxydD kuBCb
2	7	13	AxydD kuBCb
3	8	13	AxydD kuBCb

The table above shows the Overall Gain and the Final partition for various S. It can be observed that after achieving a maximum possible Overall Gain of 13, any increase in S still maintains the same Overall Gain. Also the final partition remains the same in this

case, this is because once the max. Overall Gain of 13 is achieved, there is not any further improvement in the Overall Gain and thereby the partitions are not updated.

Consider the graph in Figure.2.3 with 17 nodes which shows the node numbers. It is assumed that the outputs of the operators numbered 7,8,9,10,11,12,13,14,15,16 and 17 are stored in temporary variables A,B,C,D,E,F,G,H,I,J and K. From this graph a folded Interference graph is constructed and the heuristic was implemented with initial partitions being partition_1 = {A,B,C,D,E,F,G,H,I} and partition_2 = {b,x,k,d,y,u,J,K}, and the results are tabulated below. Table 2.6 shows the Overall Gain and the Final partition for various S. In this case, the maximum possible Overall Gain as shown in the table is 59.

Table 2. 6: Overall Gain and Final Partitions for Various values of S for the Interference Graph of Figure 2.3

S.No.	S	Overall Gain	Final partition
1	9	59	BuJGdHKb CxADkIEFy
2	10	59	ACDIkxBu bdJKHGyFE
3	11	59	ACDIkxBu bdJKHGyFE
4	12	59	ACDIkxBu bdJKHGyFE
	13	59	ACDIkxBu bdJKHGyFE

Table 2. 7: Overall Gain and Final Partitions for Various Initial Partitions for the Interference Graph of Figure 2.3

S.No.	Initial partition	Final partition	Overall Gain
1	ABCDEFGHI bxkdyuJK	ACDIkxBu bdJKHGyFE	59
2	BCEFGHI AbxkdyuJKD	FGyJbKHd AxDCIBukE	59
3	BCHIkdyu AbxJKDEFG	BCIkuDAxE bJKFGyHd	59
4	JKDEFBCu AbxGHIkdy	DBCuxIAkE byFKHJGd	59

For the same graph in Figure 2.3, the table below shows the Final partition and Overall Gain value for various Initial partitions. It can be observed from the Table 2.7, that there are several final partitions yielding the same Overall Gain value of 59 depending on the choice of the Initial partition.

2.4 Complexity Analysis

Let N be the number of nodes. In the calculation of the initial cost for each of the node in one partition, all the nodes in the other partition are considered, which requires $O(N^2)$ time. Each pass requires the following computations:

At the beginning of each pass, unmarking and unticketing each of the node requires O(N) time.

1. Each D(u) computation for nodes in one partition requires consideration of all the nodes in the same partition (for computing the internal cost) and also all the nodes in the other partition (for computing the external cost). This requires $O(N^2)$ time.
2. Computing the # of unmarked and unticketed nodes requires considering all the nodes and hence requires O(N) time.
3. Finding the node with the maximum gain requires O(N) time.
4. If the size constraint is violated then ticketing the nodes requires O(N) time.
5. Updating the D values for the unmarked nodes requires consideration of all the unmarked nodes, which requires O(N) time.
6. Finding the maximum cumulative gain requires O(N) time.
7. If the maximum cumulative gain becomes equal to zero, then the for-loop requires consideration of all the nodes in the worst case and this requires O(N) time.
8. Updating the sets X and Y require O(N) time. Updating T value requires O(1) time.

If the total pass is ‘P’ then the total time required is $P*(O(N^2) + O(N) + O(1)) = O(PN^2)$. In practice, ‘P’ is observed to be independent of N.

2.5 Chapter Summary

In this chapter, we presented several improvements to a heuristic for partitioning the variables of a graph. Results on several graphs demonstrate the effectiveness of our improvements.

3. Horizontal Address Assignment

Horizontal address assignment is the problem of assigning the variables into groups such that the number of memory accesses is reduced. The memory accesses are represented by the variable access sequence (VAS). Two variables v_i and v_j are neighbors if there are successive accesses in the VAS to these variables. Two variables v_i and v_j have an unexploited neighbor relation if they are neighbors in the VAS but are not members of the same group [14].

The main concept of minimizing the number of unexploited neighbor relations to minimize the number of memory accesses as explained in [14] is as follows: "Load the group containing the required data and work on these data as long as possible without further memory accesses. If another group should be loaded into the group register and the currently loaded group is modified, it is necessary to store the current group back to memory." Thus the number of memory accesses can be minimized by minimizing the number of unexploited neighbor relations. Now we will represent the horizontal address assignment problem as a graph partitioning problem as explained below.

Definition: The Variable Access Graph VAG = (V,E) is an undirected graph with node set V, where each node v_i in V represent a variable (v_i) in VAS and the edge set E contains set of edges between the nodes v_i and v_j if the corresponding variables v_i and v_j of VAS are neighbors. An edge which represents an unexploited neighbor relation is called an external edge otherwise this edge is called internal edge. The weights w_{ij} on the edge represent the number of the times the two variables v_i and v_j are neighbors.

Definition: The cost of an undirected graph, denoted by Cost (G) is defined as the sum of weights of edges whose end-points are not both in the same partition.

Now with this graph representation and definitions, the problem can be redefined as partitioning the variables (nodes) of the VAG such that the Cost of the VAG is minimized. To address this problem we have discussed an ILP formulation which maximizes the sum of the weights of internal edge, and a heuristic approach which uses the heuristic for variable partitioning into two memory banks, that has been discussed in the previous chapter.

3.1 ILP Formulation for Variable Partitioning into Multiple Groups

For a given VAG = (V, E), we use an integer-linear programming (ILP) approach to solve this optimization problem. Let k represent the groups and the total number of groups be P, i.e., 1<=k<=P. The ILP model comprises the following variables.

For every element v_i of V:

$X_{ik} = 1$ if variable v_i is assigned to group k

0 otherwise

For every pair of variables v_i, v_j and every group k:

$S_{ijk} = 1$ if $X_{ik} = X_{jk} = 1$

0 otherwise

Let w_{ij} be the weight of the edge (v_i, v_j). The variable X_{ik} indicates if variable v_i is assigned to group k. The variable S_{ijk} indicates whether the variables v_i and v_j are assigned to the same group k. Using these variables the internal cost between the variables v_i and v_j if both are assigned to group k is computed using the expression $S_{ijk} . w_{ij}$. If $S_{ijk} = 1$, then the internal cost between the variables v_i and v_j if they are both assigned to group k will be w_{ij}. Thus the sum of $S_{ijk} . w_{ij}$ over all v_i,v_j and k needs to be maximized in order to maximize the internal

cost of the graph. That is the objective function will be

$$\text{Maximize} \sum_k \sum_i \sum_j S_{ijk} w_{ij}$$

The requirement that $S_{ijk} = 1$ if and only if $X_{ik} = X_{jk} = 1$ is enforced by the following constraints in conjunction:

- $S_{ijk} >= X_{ik} + X_{jk} - 1$
- $S_{ijk} <= X_{ik}$
- $S_{ijk} <= X_{jk}$

The ILP formulation should also take care of the memory-width constraint, i.e., the maximum size of each group. This is enforced by the following constraints:

For all groups k: $\sum_i X_{ik} \leq S$ where S is the memory width of each group k.

Also each variable should be assigned to only one of the group. This is enforced the following constraint:

$\sum_k X_{ik} = 1$ for all the variables v_i in V.

3.1.1 Illustration

For the variable access graph given below in Figure 3.1, the ILP formulation can be illustrated as follows.

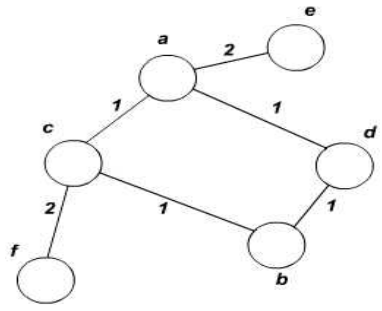

Figure 3. 1: Example Variable Access Graph to Illustrate the ILP Formulation

Let the maximum memory width of each group (S) be 3 and the total number of groups (k) be 3. Then the objective function considering all the variables and all the groups, will be

Maximize: $S_{ac1}+ S_{ad1}+2*S_{ae1} + S_{bc1} + S_{bd1}+ 2*S_{cf1} + S_{ac2}+ S_{ad2}+2*S_{ae2} + S_{bc2} + S_{bd2} + 2*S_{cf2}+ S_{ac3}+ S_{ad3}+2*S_{ae3} + S_{bc3} + S_{bd3} + 2*S_{cf3}$;

The setting $S_{ijk} = 1$, if and only if $X_{ik} = X_{jk} = 1$ for all the variables of the graph, is enforced by the following constraints.

For group k =1:

$S_{ab1} \geq X_{a1} + X_{b1} - 1;$ $S_{ab1} \leq X_{a1};$ $S_{ab1} \leq X_{b1};$

$S_{ac1} \geq X_{a1} + X_{c1} - 1;$ $S_{ac1} \leq X_{a1};$ $S_{ac1} \leq X_{c1};$

$S_{ad1} \geq X_{a1} + X_{d1} - 1;$ $S_{ad1} \leq X_{a1};$ $S_{ad1} \leq X_{d1};$

$S_{ae1} \geq X_{a1} + X_{e1} - 1;$ $S_{ae1} \leq X_{a1};$ $S_{ae1} \leq X_{e1};$

$S_{af1} \geq X_{a1} + X_{f1} - 1;$ $S_{af1} \leq X_{a1};$ $S_{af1} \leq X_{f1};$

$S_{bc1} \geq X_{b1} + X_{c1} - 1;$ $S_{bc1} \leq X_{b1};$ $S_{bc1} \leq X_{c1};$

$S_{bd1} \geq X_{b1} + X_{d1} - 1;$ $S_{bd1} \leq X_{b1};$ $S_{bd1} \leq X_{d1};$

$S_{be1} \geq X_{b1} + X_{e1} - 1;$ $S_{be1} \leq X_{b1};$ $S_{be1} \leq X_{e1};$

$S_{bf1} \geq X_{b1} + X_{f1} - 1;$ $S_{bf1} \leq X_{b1};$ $S_{bf1} \leq X_{f1};$

$S_{cd1} \geq X_{c1} + X_{d1} -1;$ $S_{cd1} \leq X_{c1};$ $S_{cd1} \leq X_{d1};$

$S_{ce1} \geq X_{c1} + X_{e1} - 1;$ $S_{ce1} \leq X_{c1};$ $S_{ce1} \leq X_{e1};$

$S_{cf1} \geq X_{c1}+ X_{f1} - 1;$ $S_{cf1} \leq X_{c1};$ $S_{cf1} \leq X_{f1};$

$S_{de1} \geq X_{d1} + X_{e1} -1;$ $S_{de1} \leq X_{d1};$ $S_{de1} \leq X_{e1};$

$S_{df1} \geq X_{d1} + X_{f1} - 1;$ $S_{df1} \leq X_{d1};$ $S_{df1} \leq X_{f1};$

$S_{ef1} \geq X_{e1} + X_{f1} - 1;$ $S_{ef1} \leq X_{e1};$ $S_{ef1} \leq X_{f1};$

Similarly the constraints for other groups (k=2 and k=3) are written.

The memory-width constraint for k =1 is written as

$$X_{a1} + X_{b1} + X_{c1} + X_{d1} + X_{e1} + X_{f1} \leq 3$$

The constraints for the other groups are written in a similar fashion. The uniqueness constraint for variable a is written as

$$X_{a1} + X_{a2} + X_{a3} = 1$$

Again, the constraints for other variables are written. All these constraints are given as the input the ILP Solver and the objective function value (the sum of weights of the internal edges) was found to be 6. The partitioning of the variables as determined by the solver is shown in Table 3.1 for total groups P = 3.

Table 3. 1: Final Partition obtained by the ILP Solver for the Graph in Figure 3.1

k	Final Partition obtained by the ILP Solver
1	-
2	bcf
3	aed

3.1.2 Experimental Results

Consider the graph in Figure 3.1 with 6 nodes. ILP formulation was applied to this Graph with the P = 2. Table 3.2 shows the Objective function value for various S values. It can be observed that as the S value increases the Objective function value increases. Table 3.3 shows the ILP results on the same Graph in Figure 3.1, with P =3.

Table 3. 2: Objective Function value for various S values for the Graph in Figure 3.1 with P = 2

No.	S	Objective function value
1	3	6
2	4	6
3	5	6
4	6	8

The following table shows the ILP results on the same Graph in Figure 3.1, with P =3.

Table 3. 3: Objective function value for various S value for the Graph in Figure 3.1 with P = 3

No.	S	Objective function value
1	2	5
2	3	6
3	4	6
4	5	6
5	6	8

Consider the graph in Figure 3.2 with 10 nodes. Let the weights of all the edges be 1 and let P =2. ILP formulation was applied for this graph and the results are tabulated below. Table 3.4 shows the Objective function value for various S values. It can be observed that as the S value increases the Objective function value increases. Table 3.5 shows the increase in the Objective function value with increase in S value for the same Graph in Figure 3.2, with P = 3.

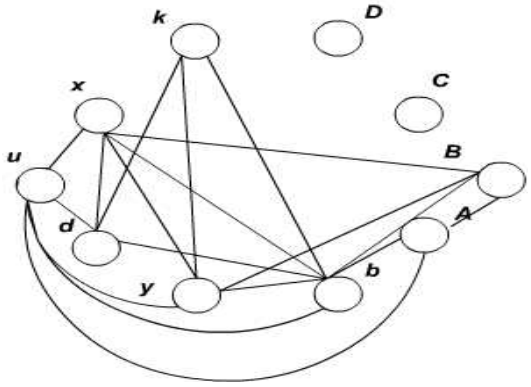

Figure 3. 2: Example Variable Access Graph on which the ILP Formulation was Implemented.

Table 3. 4: ILP Results for the Graph in Figure 3.2 with P = 2

S.No	S	Objective function value
1	6	13
2	7	15
3	8	18

Table 3. 5 ILP Results for the Graph in Figure 3.2 with P = 3

S.No	S	Objective function value
1	5	10
2	6	13
3	7	15

3.2 Heuristic for Variable Partitioning into Multiple Groups

This section discusses an heuristic for partitioning the variables of the VAG such that the Cost (VAG) is minimized. Let P be the total number of partitions or groups. Let S be the memory width of each group. If S < | V | / P, then it would be impossible to accommodate all

the | V | variables in P partitions. Therefore the value of S given as input should be greater than $\lceil |V|/P \rceil$.

ALGORITHM MULTIPLE BANK(VAG(V, E, w), Initial partitions)

```
1.  new VAG (V,E,w) = VAG(V,E,-w)  ; Count = 1;
2.  while ( count !=0)
    {
3.        Count = 0;
4.        for (a=1 to P)
          {
5.            for (b=a+1 to P)
              {
6.            Construct RVAG (partition_a, partition_b):  new VAG with only the
              edges whose both the end-points are in partition_a or partition_b;
7.            Heuristic2Bank(partition_a, partition_b, RVAG);
8.            if (T  > initialCost (partition_a, partition_b)
              {
9.                Count = Count + 1;
10.               Update the partitions;
11.           }
12.         }
13.       }
14.}
```

3.2.1 Explanation

The inputs to this heuristic are Variable Access Graph (VAG), and the initial partitions (partition_a and partition_b). This heuristic addresses the problem of finding an efficient partitioning of the variables such that the Cost (VAG) is minimized. We know that the heuristic discussed in the previous chapter, addresses the problem of partitioning the variables of graph G between 2 Banks, such that the Cost (G) is maximized. So using the heuristic for 2 Banks, for solving this problem requires making all the edge weights negative which constitutes the new Variable Access Graph (new VAG) (line 1). Also since the heuristic for 2 Banks, works on two partitions, for this multiple partitioning problem we

consider two partitions at a time and the best partitioning of the variables between the two partitions is obtained.

For all pairs of partition the following operations are performed. In line 6, a Restricted Variable Access Graph (RVAG) is constructed by including only those edges whose end-point variables are in any of the two partitions under consideration. In line 7, the two partitions and the RVAG are given as input to the heuristic for two banks. The output of this heuristic will be a partitioning of those variables between the two partitions under consideration, such that the cost of the RVAG (Cost (RVAG)), denoted by T is maximized. If T value is greater than the initial Cost of those two partitions, then the new partitions are better than the initial ones, because since all the edge weights are negative the initial Cost will be negative, and T > initial Cost implies that the Cost (VAG) is minimized, which is what is desired. Now the two partitions, i.e., partition_a and partition_b are updated with the variables in the new partitions. All the steps starting from applying the heuristic for two banks for all the partitions, checking for any improvement and updating the partitions constitutes a iteration (line 4 to line 13). This is repeated until in any iteration none of the partition shows any improvement (line.2 to line.14). Thus the final output will be a partition such that Cost (new VAG) is maximized, and thereby the Cost (VAG) is minimized.

3.2.2 Illustration

Let us illustrate this heuristic with a VAG of Figure 3.1. The new VAG with negative edge weights is shown in Figure 3.3. Let P be 3 and the initial partitions partition_1 = ab, partition_2 = ed and partition_3 =cf and the maximum size of each partition S be 4. Table 3.6 shows the two partitions partition_a, partition_b , their initial Cost, final Cost T and the updated partitions for each pair of partitions for each of the iteration.

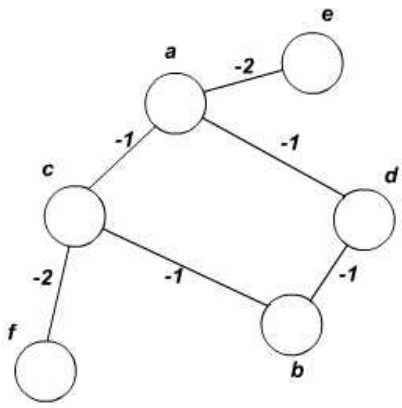

Figure 3. 3: Input Graph for the VAG in Figure 3.1

Table 3. 6 Illustration of the Heuristic with the example Graph in Figure 3.3

Itn.	a	b	partition_a	partition_b	initial Cost	T	Updated partitions		
							Partition_1	Partition_2	Partition_3
1	1	2	ab	ed	-4	0	bdae	-	cf
	1	3	bdae	cf	-2	-2	ae	-	cfbd
	2	3	-	cfbd	0	0	ae	dbcf	-
2	1	2	ae	dbcf	-2	-2	aed	bcf	-
	1	3	aed	-	0	0	-	bcf	aed
	2	3	bcf	aed	-2	-2	-	cf	daeb

From the table it is clear that in the second iteration there is no more improvement in the cost by any of the 3 partition and therefore the program terminates. The final partition as shown in the table is cf, daeb and the Cost of the VAG for this partition is found to be 2.

3.2.3 Experimental Results

Consider the graph in Figure.3.3 (6 nodes). Let the initial partitions be partition_1 = ab, partition_2 = ed and partition_3 =cf. The heuristic was implemented on this graph

and the results are tabulated below. Table 3.7 shows the Final partition and the Cost for various S values. It can be observed that as the S increases the Cost decreases. This is because the heuristic assigns the variables to the same partition as much as possible, and thereby the Cost decreases as the size S increases.

Table 3. 7: Heuristic Results for the Graph in Figure 3.2 with P = 3

S	Final partitions			Cost
	Partition_1	Partition_2	Partition_3	
3	b	dae	cf	3
4	-	cf	daeb	2
5	-	f	eadbc	2
6	-	-	abcdef	0

Now let P = 4. Let the initial partitions be partition_1 = ae, partition_2 = c, partition_3 = f, partition_4 = bd. Table 3.8 shows the variation of the Cost with S value with P = 4.

Table 3. 8 Heuristic Results for the Graph in Figure 3.3 with P = 4

S	Final Partition				Cost
	Partition_1	Partition_2	Partition_3	Partition_4	
3	aec	-	f	bd	4
4	-	-	ae	cfbd	2
5	-	e	fcbda	-	2
6	-	-	abcdef	-	0

Consider the graph in Figure.3.2 (10 nodes). From this Graph, a graph with negative edge weights is constructed and is given as Input to the heuristic.The table 3.9 below shows

the results of the Heuristic on this Graph with the initial partitions being partition_1 = BCAbx and partition_2 = kdyuD.

Table 3. 9 Heuristic Results for the Input Graph of the VAG in Figure 3.2 with P = 2

S	Final partitions		Cost
	Partition_1	Partition_2	
5	BCAbx	kdyuD	9
6	byBAxu	dDCk	5
7	dAubxBy	kDC	3
8	ABbxyudk	DC	0

Table 3.10 shows the final partition and Cost for various S value, for the input graph of Figure 3.2 with the initial partitions being partition_1 = B, partition_2 = CA, partition_3 = bxk, partition_4 = dyuD.

Table 3. 10 Heuristic Results for the Input Graph of the VAG in Figure 3.2 with P=4

S	Final Partitions				Cost
	Partition_1	Partition_2	Partition_3	Partition_4	
4	ABbx	C	duyk	D	9
5	C	BAx	dkbyu	D	8
6	ABbyux	C	dk	D	5
7	dxubABy	k	C	D	3

3.3 Result Comparison ILP Vs Heuristic

Consider the Input Graph of Figure 3.2. The table shows the Cost obtained by the ILP Formulation and the heuristic, for various S value with the initial partitions being BCAbx and kdyuD. It can be observed that the heuristic results are same as that obtained by the

ILP, except for S = 5. This is because, we have exactly 5 variables in each of the two partitions and also the S = 5, so any transfer from any partition will violate the size constraint and thereby the Cost remains the Initial Cost of 9.

Table 3. 11 Result Comparison for the Input Graph of Figure 3.2

S.No	S	ILP Cost	Heuristic Cost	Final Partition obtained by the Heuristic
1	5	8	9	BCAbx kdyuD
2	6	5	5	byBAxu dDCk
3	7	3	3	dAubxBy kDC
4	8	0	0	ABbxyudk DC

3.4 Chapter Summary

This chapter presented an integer linear programming formulation for the horizontal address assignment problem for digital signal processors with SIMD memory accesses. In addition, we have developed a heuristic for the same problem. Experimental results on several graphs demonstrate that the heuristic is very effective.

4. Conclusion

Exploiting the memory features in modern DSP architectures remains a significant challenge for optimizing compilers. This thesis presents and evaluates solutions for two important problems in generating efficient code for DSP architectures with multiple memory banks and SIMD DSP architectures.

In Chapter 2, a heuristic was proposed for addressing the problem of variable partitioning between two memory banks such that there is maximum parallel access between the variables, thereby reducing the execution time and the code size. This heuristic incorporates the assumption that at any point of time during the execution, the size of the two banks need not be constant and uses the concept of transferring the node rather than swapping the nodes. This heuristic was an improvement to the heuristic proposed in [15], in terms of the possibilities that have been considered. We have considered two additional possibilities to the latest heuristic proposed in [15], which has shown to produce significant results. We have also demonstrated with an example how the gain improvement is achieved by considering those two possibilities. We have also observed that the same gain can be achieved by a number of final partitions, depending on the choice of the initial partition.

In Chapter 3, we have discussed the variable partitioning problem for DSPs with SIMD architecture to minimize the number of memory accesses. We have discussed how to map this problem to the variable partitioning problem discussed in the previous chapter. To address this problem, we have developed an Integer Linear Programming (ILP) model as well as a heuristic approach. We have demonstrated with an example how

the ILP formulation is used for addressing this problem and have tested this formulation for several graphs. We have observed that as the maximum size of the partition increases, the objective function value of this formulation also increases. In the next section of Chapter 3, we have presented a heuristic for addressing this problem. This heuristic builds on the heuristic developed in Chapter 2. We have demonstrated in detail one iteration of this heuristic for a sample graph and have tabulated the results of execution on a couple of graphs. At the end we have compared the results of the ILP formulation with that obtained by the heuristic and have shown that the heuristic results are promising.

4.1 Future Work

In Chapter 2 we have discussed how to partition the variables between two memory banks such that there is maximum parallel access between the variables. We have not considered the execution time of the operations for this problem. So, one possible future work will be to extend the heuristic for two banks, discussed in Chapter 2, to partition the variables with the execution times of the operations under consideration. A second problem that needs to be addressed is effective code generation dealing with control-flow (i.e., branches) through the use of profiling techniques; note that our work assumes straight-line code. Another possible avenue of research is the exploration of the use of static single assignment (SSA) which allows one to breakdown the live ranges of variables so that a variable could be in different banks during different sections of its live range. In addition, there is scope for exploring the design space of multiple memory banks by getting the compiler in the loop.

References

[1] Analog Devices, Inc., *ADSP-2100 Family User's Manual*, 1993.

[2] S. Atri. *Improved Code Optimization Techniques for Embedded Processors*. M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, December 1999.

[3] S. Atri, J. Ramanujam, and M. Kandemir. Improving variable placement for embedded processors. In *Languages and Compilers for Parallel Computing*, (S. Midkiff et al. Eds.), Lecture Notes in Computer Science, vol. 2017, pp. 158-172, Springer-Verlag, 2001.

[4] C. Fiduccia and R. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the Design Automation Conference*, 1982.

[5] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[6] *GEPARD Familiy of Embedded Software Programmable DSP Cores*, Austria Mikro Systeme International (AMS), Graz, Austria, 2000.

[7] A. Gierlinger, R. Forsyth and E. Ofner. Gepard – A Parameterisable DSP Core for ASICs. In *Proc. Int. Conf. on Signal Processing Applications and Technology (ICSPAT)*, 1997.

[8] J. Hong. *Memory Optimization Techniques for Embedded Systems*, PhD Thesis, Dept. of Electrical and Computer Engineering, Louisiana State University, July 2002.

[9] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49:291-307, Feb. 1970.

[10] R. Leupers. Offset Assignment Showdown: Evaluation of DSP Address Code Optimization Algorithms. In *Proc. Compiler Construction: 12th International Conference (CC 2003)*, pp. 290-302.

[11] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *Proc. International Conference on Computer-Aided Design* (ICCAD), 1996.

[12] R. Leupers and D. Kotte. Variable Partitioning for Dual Memory Bank DSPs. In *Proc. IEEE International Conference on Acoustics Speech and Signal Processing*, pp. 1121-1124, 2001.

[13] S. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, MIT Department of EECS, January 1996.

[14] M. Lorenz, D. Kottmann, S. Bashford, R. Leupers, and P.Marwedel. Optimized Address Assignment for DSPs with SIMD Memory Accesses. In *Proc. Proceedings of the 2001 Conference on Asia South Pacific Design Automation*, pp. 415-420, Yokohama, Japan, 2001.

[15] S. Mahapatra. *Heuristics for Offset Assignment in Embedded Processors*. M.S. Thesis, Louisiana State University, 2005.

[16] Motorola, Inc. *DSP56000 Digital Signal Processor Family Manual*, 1992.

[17] D. Powell, E. Lee, and W. Newman. Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams. In *Proc. ICASSP*, 1992.

[18] M. Saghir, P. Chow, and C. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[19] A. Sudarsanam and S. Malik. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. *ACM Trans. Design Automation of Electronic Systems*, 5(2):242-264, April 2000.

[20] Texas Instruments, Inc. *TMS320C2x User's Guide*, 1993.

[21] K. Yelick. Lecture Handout on Graph Partitioning, Part II, CS 267: Applications of Parallel Computers, University of California, Berkeley, CA. Fall 2001. http://www.cs.berkeley.edu/~yelick/cs267f01/lectures/Lect18-Partition1.ppt.

[22] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr. DSPStone – A DSP-oriented Benchmarking Methodology. In *Proc. Int. Conf. on Signal Processing Applications and Technology* (ICSPAT), 1994.

Vita

Saravanan Subramanian was born in India in 1981. He did his schooling at Alpha Matriculation Higher School, Chennai, and then his Bachelor of Engineering degree at the University of Madras, Chennai. After working at AdventNet Development Centre, Chennai, as a Software Engineer he came to Louisiana State University to pursue his master's degree. During his enrollment in the graduate school, he served as a graduate assistant at the Louisiana State University's College of Basic Sciences. He will graduate with the degree of Master of Science in Electrical Engineering in December 2005.