Louisiana State University

LSU Scholarly Repository

LSU Master's Theses

Graduate School

2005

Heuristics for offset assignment in embedded processors

Satya Swaroop Mahapatra

Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://repository.lsu.edu/gradschool_theses

Part of the Electrical and Computer Engineering Commons

Recommended Citation

Mahapatra, Satya Swaroop, "Heuristics for offset assignment in embedded processors" (2005). *LSU Master's Theses*. 3552.
https://repository.lsu.edu/gradschool_theses/3552

This Thesis is brought to you for free and open access by the Graduate School at LSU Scholarly Repository. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Scholarly Repository. For more information, please contact gradetd@lsu.edu.

HEURISTICS FOR OFFSET ASSIGNMENT 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

Satya Swaroop Mahapatra
B.E., M.S. Ramaiah Institute of Technology, India, 2000
May 2005

ACKNOWLEDGMENTS

I would like to express my immense gratitude towards my research advisor, Professor Dr. Jagannathan Ramanujam, for his continued support, advice, and encouragement to me towards my accomplishing the completion of my research analysis of the problem. His thorough understanding of the many issues fundamental to the area of embedded processor code design, and his attention to the fine details and basics that very often govern the behavior of modern-day systems will always be remembered.

I would like to thank Dr. Jerry Trahan and Dr. Bijaya Karki, members of my research committee for their valuable suggestions and advise. Also, I would like to thank Dr. Jinpyo Hong for his inputs and ideas.

I dedicate this work to my parents and to my sister for their constant advice, prayers, encouragement and moral and financial support throughout the process of my research, as also all through my life.

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. Also, my thanks to the many other people whose faces pass through my mind as I ruminate about the wonderful period spent at LSU. It was a pleasure knowing them all, and that, as much as anything else, made this entire journey of my life worthwhile.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ……………………………………………………….…………………..ii

LIST OF TABLES …………………………………………………………………………v

LIST OF FIGURES ………………………………………………………………………vi

ABSTRACT ……………………………………………………………………………vii

1. INTRODUCTION……………………………………………………………………1
1.1 Processor Architecture ……………………………...…………………….....1
1.2 Background …………………………………………………………………...4
1.3 Research Objective ……………………………...……………………………6
1.4 Organization of This Thesis ………………………………………….……8

2. THE SINGLE OFFSET ASSIGNMENT PROBLEM ……………………………..9
2.1 Background …………… ………………………………………………………..9
2.2 Drawbacks, Comparison and Motivation ……………………….………….13
2.3 Motivating Example …………………………………………………………13
2.4 A New Approach to SOA……….………………………...…………………13
2.4.1 The Heuristic ………………….……………………………………14
2.4.2 Explanation and Working ……………..……………………………14
2.4.3 Example ……………………………………………………………...15
2.4.3.1 Execution Steps ……………………..…………………….16
2.4.3.2 Comparison and Analysis …………………….………..19
2.4.4 Results of the SOA Heuristic Implementation ……………………19
2.4.5 Observation and Analysis of the SOA Results ………….……..20
2.5 Chapter Summary…………………………………………………………21

3. THE GENERAL OFFSET ASSIGNMENT PROBLEM …………………………22
3.1 Leupers and Marwedel's GOA Heuristic …………………..………………22
3.2 Drawbacks of the GOA Heuristic of Leupers and Marwedel…..……………27
3.2.1 Static Nature of the Access Sequence …………..………..……28
3.2.2 Register Utilization ………………….…………………………....30
3.2.3 Element of Randomness …………..……………………………31
3.2.4 Edges with Equal Weights …………………………..……………31
3.3 An Analytical Insight …………………………………………..……...31
3.4 A New GOA Heuristic …..………………………………………….……33
3.4.1 Explanation and Working …………………….…………………35
3.4.2 Example ……………………………………………………………...38
3.4.3 Results ………………………………………………………………..39
3.4.4 Observations and Analysis ……………..…………………………41
3.5 Chapter Summary …..………………………………………………………42

4. VARIABLE ASSIGNMENT INTO MEMORY BANKS…………………………44
4.1 Overview……………………….…….……………………………45
4.1.1 Example………………..……………………………………46
4.2 The Variable Partitioning Problem…….……………………………….48
4.3 A General Approach………………………………………………48
4.3.1 Heuristic…………………………………………………..52
4.3.2 Explanation………………………………………………….53
4.3.2.1 Illustration………..…………………………………54
4.3.2.2 Results……………..………………………………..55
4.3.3 Complexity Analysis…………..………………………………56
4.4 Chapter Summary …………………..………………………………57

5. CONCLUSION AND SCOPE FOR FUTURE WORK…………………………..59
5.1 Possibilities for Further Work on SOA and GOA …….……………..….60
5.1.1 Introduction of Other Tie-Break Parameters……………….………...60
5.1.2 Sequencing of Tie-Break Criteria……………….………………61
5.1.3 Formulation and Heuristics for Higher Orders……….…………61
5.2 Possibilities for Further Work on the Variable Partitioning Problem ……..62

REFERENCES AND BIBLIOGRAPHY…………….……………………….....63

VITA……………………………………………………………………………66

LIST OF TABLES

2.1: Initial edge weights and T-values for the access sequence of Section 2.1…..….......16

2.2: Updated T-values after addition of edge (e, a) ……………………………………..17

2.3: Updated T-values after addition of edge (a, d) ……………………………………..17

2.4: Updated T-values after addition of edge (d, e) ……………………………………..18

2.5: Updated T-values after addition of edge (d, b) ……………………………………..18

2.6: Results of the SOA implementation ………………………………………………..20

3.1: Results GOA on sample access sequences…………………...……………………….39

3.2: Results of the GOA heuristic on randomly generated access sequences assuming the availability of k = 16 address registers …………………………………………..…...40

3.3: Results of the GOA heuristic on randomly generated access sequences assuming the availability of k = 8 address registers …………………………………………...…...40

3.4: Results of the GOA heuristic on randomly generated access sequences assuming the availability of k = 4 address registers ……………………………………..……..…...41

3.5: Results of the GOA heuristic on randomly generated access sequences assuming the availability of k = 2 address registers ……………………………………..……..…...41

4.1: Initial computations of the costs associated with the nodes of Figure 4.3…………..54

LIST OF FIGURES

1.1: A generic Address Generation Unit (AGU) model [15]…………………………….….2

2.1: Access graph for the access sequence aadeaeadccdbcdeedbda …………………9

2.2: SOA cost computation for the graph of Figure 2.1…………………………...…..10

2.3: Pseudo code for SOA Heuristic ………………………………………………….14

2.4: Final selection of edges for the MWPC ………………………………………….19

3.1: Access graph for the access sequence afagaeadacababcdefgh ……………...24

3.2: Updated access graph after the selection of the edge (a, b) ………………………...25

3.3: Updated access graph after the selection of the edge (c, d)...………………………26

3.4: Access graph after the selection of the edge (g, h) ………..…………………………26

3.5: Access graph from the updated sub-access sequence "fgedccdefgh"………..…29

3.6: Pseudo code for the new GOA heuristic…………………………………………… 34

4.1: Black box model of the Gepard DSP architecture [16]……………………………..44

4.2 Example data dependency graph ……………………………………………………47

4.3 Benefit graph for Figure 4.2 …………………………………………………………47

4.4: A heuristic for an initial partition ……………………………………………………..49

4.5: Pseudo-code for the general variable partitioning heuristic …………………...…...52

4.6: Data dependency graph for the differential equation problem……………….……..56

ABSTRACT

This thesis deals with the optimization of program size and performance in current generation embedded digital signal processors (DSPs) by the design of optimal memory layouts for data. Given the tight constraints on the size, power consumption, cost and performance of these processors, the minimization of the code size in terms of the number of instructions required and the associated reduction in execution time are important. Several DSPs provide limited addressing modes and the layout of data, known as offset assignment, plays a critical role in determining the code size and performance. Even the simplest variant of the offset assignment problem is NP-complete. Research effort in this area has focused on the design, implementation and evaluation of effective heuristics for several variants of the offset assignment problem.

One of the most important factors in the determination of the size, and hence, the execution time of a code is the number of instructions required to access the variables stored in the processor memory. The indirect addressing mode common in DSPs requires memory accesses to be realized through address registers that hold the address of the memory location to be accessed. The architecture provides instructions for adding to and subtracting from the values of the address registers to compute the addresses of subsequent data that need to be accessed. In addition, some DSP processors include multiple memory banks that allow increased parallelism in memory access. Proper partitioning of variables across memory banks is critical to effectively using the increased parallelism.

The work reported in this thesis aims to evolve efficient methods for designing memory layouts under the conditions of availability of one address register (SOA) or of

multiple address registers (GOA). It also proposes a novel technique for choosing the assignment of variables to the memory banks. This thesis motivates, proposes and evaluates heuristics for all these three problems. For the SOA and GOA problems, the heuristics are implemented and tested on different random sample inputs, and the results obtained are compared to those obtained by prior heuristics. In addition, this thesis provides some insight into the SOA, GOA and the variable partitioning problems.

1. INTRODUCTION

Modern day systems such as digital cameras and cellular phones are becoming more and more compact, while continuing to provide increasingly complex functionality. Such application-specific systems with processing cores are referred to as *embedded systems*. The processors embedded in these application-specific systems include sufficient memory required to execute the application and store information that is relevant to the features for which the systems are designed. The execution and storage of the application code necessitates the availability of adequate memory in the processor as well as the availability of memory external (or off-chip) to the processor in many cases. In order to attain compactness and complete functionality simultaneously, the objective of achieving a reduction in the system size or area while maintaining the execution time (i.e., required performance) at the specified rate is of paramount importance in the design of application-specific embedded systems.

1.1 Processor Architecture

The sequence of accesses (reads and writes) to data in a program can be represented by an access sequence. An access sequence is the order in which variables in the program are accessed, each access being either a read or a write operation. These variables are typically stored in memory. In several embedded processors such as the TI TMS320C5, the variables are accessed through an address register that contains the address of a particular memory location at any given time. Such addresses are generated by the address generation unit (AGU), the architectural diagram [15] of which is as

shown in Figure 1.1. Embedded processors used in digital signal processing are referred to as Digital Signal Processors (DSPs).

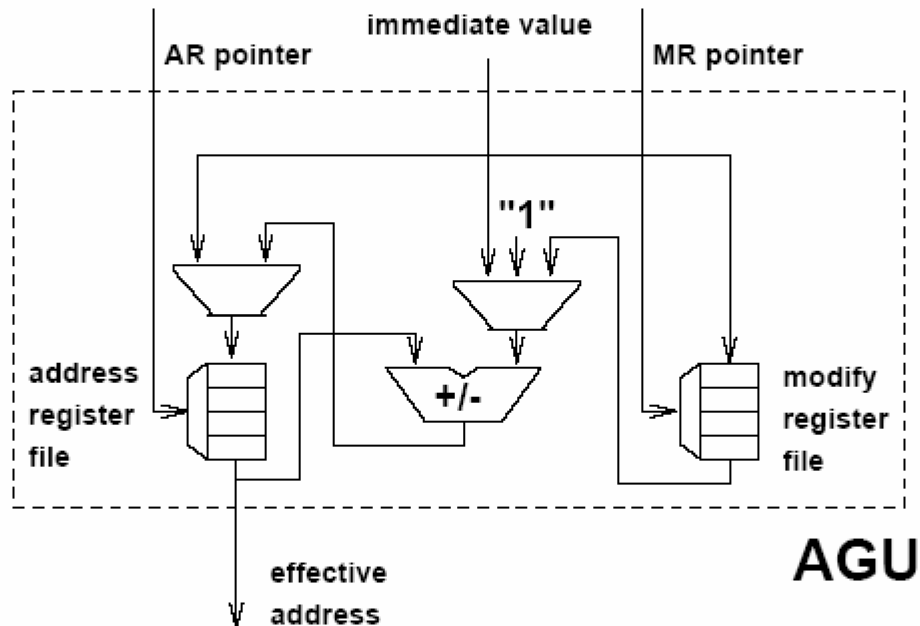

Figure 1.1: A generic Address Generation Unit (AGU) model [15].

Figure 1.1 depicts a typical architectural model of an AGU unit, which consists of an address register (AR) file and a modify register (MR) file, the individual registers of which are accessed through AR and MR pointers respectively. The input of "1" into the multiplexer preceding the +/- operator circuit indicates the process of carrying out increment/decrement operations in steps of 1, known as unary increments [27]. In DSPs such as the ADSP-2100 family and TI TMS320C5 that include address generation units that operate in parallel with the data path, the commands that are typically used to perform address register operations in addition to increment/decrement are:

(i) LDAR – which loads a certain value into the address register,

(ii) ADAR – which adds to the value of the address register,

(iii) SBAR – which subtracts a certain value from the address register, and

(iv) LDMR – which loads a certain value into the modify register.

These instructions typically take one hardware clock cycle to execute. In addition, several embedded processors such as the TI TMS320C25 [26], the ADSP-2100 family [1] and the MDSP56000 [21] include the unary auto-increment and auto-decrement operations that can post-increment or post-decrement the address register within the same instruction cycle as a LOAD or ADD instruction [2, 3, 4]. Because of the fact that any modification to the value of the address register other than an automatic unary auto-increment or auto-decrement operation requires an extra machine clock cycle, the problem of reducing the execution time of a program sequence can be translated to the problem of finding a good memory layout for the access sequence, and thus for the program [10]. For a typical AGU architecture [28], this optimal memory layout, which yields the minimum execution time of the program sequence among all the possible combinations of variable ordering in the memory layout, is called the optimal offset assignment [5]. The process of coming up with an optimal offset assignment [5] with unary auto-increment/decrement is simpler to understand, visualize and solve when compared to the case with higher auto-increment/decrements available in some DSPs [25] that increase the design complexity of those processors. Under current design considerations, a cost would have to be paid in terms of time units (clock cycles) for every non-unary increment/decrement operation of the address register, for which the update would have to be performed explicitly using a separate instruction. Unary increment/decrement operations are 'free' in the sense that their update can be performed in parallel with the loading of the contents of the location of the address register into the

accumulator, thus eliminating the need for an additional instruction in the code (affecting code size) and an extra clock-cycle in execution (affecting performance).

1.2 Background

The optimization [18] of the code can be achieved by a reduction either in the execution time (machine clock cycles), the total number of registers used, or simply in the total number of instructions in the code. In this research, we focus on achieving a reduction in the execution time of the code by designing the memory layout (offset assignment) in such a way that the total number of non-unary address register increments necessitated by the access sequence is minimized, and this is particularly useful in codes involving a large amount of array manipulations [13]. Since there is a direct relationship between the execution time of a code and the availability of resources, the analysis of the code sequence with respect to different levels of resource utilization and availability is important in the design of the offset assignment. The simplest [19] of all the cases is called the "single offset assignment" (SOA), wherein it is assumed that only one address register is available in the address generation unit of the target machine. The following are the assumptions [19] made in the offset assignment problem:

1. Variables are stored in memory locations in a linear fashion. In other words, a variable can be surrounded by at the most two other variables.
2. At any given instant of time, only one variable can be pointed at by the address register, and a memory location can correspond to only one variable.

Using the access sequence that is derived from the given code, an undirected graph is constructed as follows: For every distinct variable in the access sequence, a node is drawn. Beginning from the left and ending at the right, the access sequence is scanned

and an edge is introduced between two nodes if their corresponding variables occur side-by-side in the access sequence and the weight of the edge is set to one; if such an edge already exists, then the weight of the edge is incremented by one. At the end of the graph construction, it can be seen that the sum of the weights of all the edges (including self-edges) in the graph would have to be one less than the total length of the access sequence. However, if self-edges are excluded, then the sum of the weights of all the edges in the graph would be identical to the total number of pairs of distinct variables adjacent to each other in the access sequence. In a layout, each variable can be adjacent to at the most two other variables and the two end-points of the layout have exactly one neighbor each. According to [19, 12], the problem of solving the single offset assignment problem for the above graph is equivalent to solving the Maximum Weight Path Cover (MWPC) problem. Since this problem is NP-complete [9], a heuristic is provided in [19] for the SOA problem that selects edges from a list of edges sorted in decreasing order of edge weight, while ensuring that the selected edges do not form a cycle and no more than two edges incident at a node are chosen.

However, in the SOA heuristic presented in [19], the method of selection of edges having equal weight from the sorted list of edges was not specified, and so the possibility of a difference in the quality of results obtained by the selection of different edges was not investigated. To probe this issue, a tie-breaking function was introduced for the SOA heuristic in [15] to take into account the ordering of edges with equal weights.

There are instances where registers utilized specifically for storing frequently occurring 'modify' values are made available. In other words, there are dedicated registers which are used to store the value of the most frequently occurring non-unary

increment operation. These registers are referred to as modify registers [14,15], and their use has been proven to result in a significant reduction in the size of the machine code by reducing the number of instructions required for address generation [17]. Just like auto-increment and auto-decrement, the value of an address register can be changed by adding or subtracting the value of the modify register for "free," i.e., in the same cycle in parallel with the operation of the data path.

It has been shown that the incorporation of more than one address register in the system design can yield certain performance benefits in certain instances, and these types of offset assignment calculation problems involving more than one address register is referred to as the "general offset assignment" (GOA) problem. The performance benefits of the same can be attributed to the fact that since a GOA heuristic results in a distribution of graph variables among the multiple address registers available, each address register is responsible for the memory accesses of only the variables assigned to it. The heuristics for the GOA problem as described in [19] and [15] incorporate the SOA heuristic as a subroutine in their execution.

1.3 Research Objective

The heuristics presented in [19] do not incorporate any tie-breaking. Leupers and Marwedel [15] present a static tie-breaking function for SOA used once while initially sorting the edges in decreasing order of weight, for determining the order of selection for testing for eligibility to be added to the current path cover update. Once a particular edge has been added to the current path cover, it changes the availability of other edges (incident at the end points of the chosen edge) to be chosen for the path cover. These changes are apparent only when the graph is updated dynamically after each edge

selection. But the heuristic in [15] does not consider the potential effect of updating the tie-break function dynamically whenever an edge is selected. Leupers and Marwedel [15] propose using the tie break values (T-values) as a basis for selecting among edges of equal weight, where the T-value of an edge is defined as the sum of weights of all edges incident at its end points. Presently, the selection of the candidate for addition to the current path cover from amongst all the edges having equal weights and equal T-values is assumed to be done arbitrarily.

In the heuristic for the GOA problem known hitherto, the establishment of the SOA cost of the original access sequence as a useful bound for the subsequent cost computation process has not been addressed. This, in addition to the issue of recognition of the access graph identity after the selection of the edge with the maximum weight or the minimum T-value [15] are potential windows for improvement over the results of the hitherto known GOA heuristics, and these become apparent through a dynamic approach wherein the SOA cost is computed on an updated access graph on an iterative basis.

The aim of this research work is to explore the possibility of obtaining a better solution to the MWPC problem by dynamically updating the list of candidate edges each time an edge is added to the current path cover after meeting the requirements – that a node on a path cover cannot have more than two edges of the path cover incident on it, and that the path cover cannot contain cycles. If instead of picking from the set of candidate edges arbitrarily, a tie-break function is implemented at every step as mentioned above, then there could be an improvement in the time performance characteristics of the code as opposed to the case in which a naïve approach is adopted in selecting edges to the current path cover. Furthermore, if there is an updating of the

access sequence (this is the case for GOA) each time a candidate edge is selected, then that might provide with some useful insight into a more efficient way of coming up with an optimal solution.

The allocation of variables into different memory banks in the processor architectures that have the facility of incorporating multiple variable banks is another related topic on which research is being carried out. It has been proven that partitioning the variables used in the code of the particular processor application into different memory banks results in increased efficiency of bandwidth allocation and utilization, as also higher levels of code quality. However, the heuristic proposed in Kernighan and Lin [11] incorporates a rigid assumption that there is an even distribution of variables between the two memory banks and considers solving the problem by the use of variable swapping, and therefore the possibility of obtaining an improvement as a result of the relaxation of this restriction has not been investigated.

1.4 Organization of This Thesis

This thesis deals with the modification of the heuristics presented in [15] and [19] to yield a dynamic heuristic for solving the GOA and SOA offset assignment problems with potentially more economical results. Chapter two is the overview of the SOA problem, a summary of its drawbacks and flaws in assumptions, motivation for a new heuristic, and the heuristic and implementation results. Chapter three explains the GOA problem, the drawbacks in the heuristics proposed hitherto, the motivation for a new approach, and the heuristic with implementation results and analysis. Chapter four focuses on a related topic in the area of variable assignment to memory banks, and chapter five of this thesis is the conclusion and offers scope of future research work in these research areas.

2. THE SINGLE OFFSET ASSIGNMENT PROBLEM

The SOA heuristic proposed by Liao *et al*. [19] begins with the process of constructing an access sequence out of the application code of the processor in the order in which the variables are accessed, and then constructing the access graph for the sequence. Out of this access graph, a subset from among the edges is picked in such a way that: (i) not more than two selected edges are incident on any node; and (ii) no cycles are induced by any subset of the selected edges. The selection process is done with the edges being sorted in descending order of edge weights, and once all the edges have been explored, the total cost of the system is the sum of the edge weights of the unselected edges, plus an additional cost unit required for the initialization of the address register.

2.1 Background

As an example, consider the following access sequence:

`aadeaeadccdbcdeedbda`

For the above access sequence, the access graph is shown in Figure 2.1.

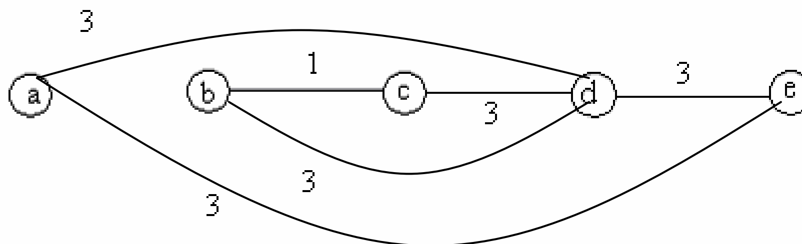

Figure 2.1: Access graph for the access sequence `aadeaeadccdbcdeedbda`

The edges of the access graph are sorted in descending order as follows.

(a, d), (d, e), (e, a), (d, c), (d, b), (b, c)
3 3 3 3 3 1

Now, the edges are picked from these in the order shown above to form a subset in such a way that the constraints mentioned above are not violated. In other words, our aim is to find the maximum weight path cover (MWPC) of this graph. Since Liao *et al.* [19] does not account for choosing between edges with equal weights, a random choice is made between the highest weighted edges of 3. We begin with the empty set, and add the first edge, say (a, d) to it. Then, we see that the next edge (d, e) when added does not form a cycle and does not cause more than two edges to concur at a node. So, it is added to the set. Now, the next edge (e, a) is discarded because it results in the formation of a cycle between the selected set edges (a, d) and (d, e).The edges (d, b) and (d, c) are discarded because they cause more than two edges to concur at node d, and the last edge (b, c) which satisfies both the conditions is finally selected. Thus, a valid selection according to [19] is as shown in Figure 2.2.

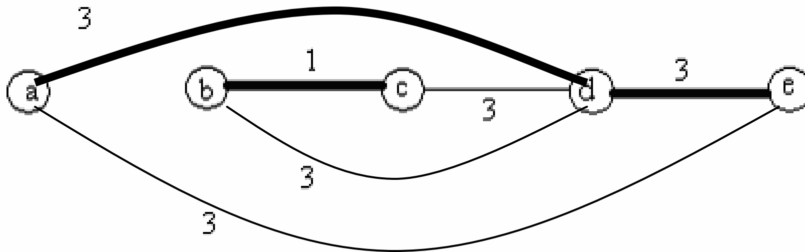

Figure 2.2: SOA cost computation for the graph of Figure 2.1

Figure 2.2 shows the SOA solution to the access graph of Figure 2.1, with the selected edges being shown darkened, and the edge weights shown by the number digits. It is seen that the variables in the sets ‘eda’ and ‘cb’ need to be adjacent in the offset assignment. Also, it should be noted that there are many possible ways of deriving the offset assignment satisfying the above conditions by means of considering the various

combinations of possible locations of these variable subsets, but further analysis on the optimal ordering of these is beyond the scope of this thesis, and therefore it would suffice henceforth to mention only one randomly out of all the possible feasible offset assignments as the solution. A valid offset assignment for this example would be `edabc`. The total cost of this example access graph using one address register is the sum of the edge weights of the unselected edges (undarkened), which would be 3+3+3 = 9 in this case; in addition, the initialization of the address register with the address of variable e needs one instruction (LDAR), and so the total cost incurred is 10 units.

However, as is seen above, this heuristic proposed by Liao *et al.* [19] does not provide for a way of determining which edge is to be selected first in the event of there being two or more edges having the same weight. Therefore, Leupers and Marwedel's [15] introduced the concept of the tie-break function. However, to prevent the addition of an edge's weight twice during the computation of its T-value, we henceforth follow the uniform convention of subtracting from the sum the weight of the edge once.

In the Leupers and Marwedel's [15] heuristic, the following procedure would be followed: everything from the derivation of the access sequence up to the sorting of the edges in decreasing order would be the same, along with the additional operation of considering the edges sorted based on their T-values. In other words, from among the set of edges having equal weights, the edge with the lowest T-value would be selected first, and so on.

For the example shown, the process of running the Leupers and Marwedel's [15] heuristic consists of the following steps: everything from the derivation of the access sequence out of the code up to the sorting of edges in decreasing order of weights is the

same as in [19]; in addition, the T-values of all the edges with equal weights are calculated. Thus, the T-values would have to be computed for the edges (a, d), (d, e), (e, a), (d, b) and (d, c) as follows.

$$T(a, d) = \sum w(a, *) + \sum w(*, d) - w(a, d) = (3 + 3) + (3 + 3 + 3 + 3) - 3 = 15$$

$$T(d, e) = \sum w(d, *) + \sum w(*, e) - w(d, e) = (3 + 3 + 3 + 3) + (3 + 3) - 3 = 15$$

$$T(e, a) = \sum w(e, *) + \sum w(*, a) - w(e, a) = (3 + 3) + (3 + 3) - 3 = 9$$

$$T(d, c) = \sum w(d, *) + \sum w(*, c) - w(d, c) = (3 + 3 + 3 + 3) + (3 + 1) - 3 = 13$$

$$T(d, b) = \sum w(d, *) + \sum w(*, b) - w(d, b) = (3 + 3 + 3 + 3) + (3 + 1) - 3 = 13$$

Now, the edge (e, a) is selected first since it has the lowest T-value, and is added into the set. Then, since (d, c) and (d, b) which have equal weights also have equal T-values, we make a random choice and select edge (d, c). It is seen that the addition of edge (d, c) does not cause any violations and so it is added to the set. Of the remaining edges, edge (d, b) has the lowest T-value of 13, and since it does not violate any of the constraints, it is added into the set. Now, edges (a, d) and (d, e) are discarded due to the node-concurrence limit, and the remaining edge (b, c) is discarded since its addition into the set would result in the formation of a cycle with selected edges (d, c) and (d, b). In this case, the total cost of the sequence would be the sum of weights of non-selected edges = w(a, d) + w(d, e) + w(b, c) = 3 + 3 + 1 = 7, plus one due to AR initialization, totaling 8 units. For this example, it should be noted that during the process of selecting an edge at random from multiple edges having equal weights and T-values, if edge (d, b) was selected instead of edge (d, c), then the maximum weight path cover would still be the same due to the fact that edges (d, c) and (d, b) when selected in any order would yield the same remaining possibilities at that stage in both the cases.

2.2 Drawbacks, Comparison and Motivation

As was mentioned earlier, the heuristic proposed by Liao *et al*. [19] does not provide a method for determining which edge to select out of a set of edges having equal weights in the sorted list of edges. And even in the heuristic proposed in [15], once an edge is selected, the weight of the edge should cease to be a contributing factor to the T-values of other edges in the graph, and this fact can be visualized clearly in an implementation involving the dynamic update of the weights of all the edges in the graph after the selection of each edge which forms the MWPC. This issue has not been addressed therein.

2.3 Motivating Example

Let us consider the example access sequence presented in Section 2.1. If instead of calculating the T-values of the edges with equal weights only once, we consider a heuristic that dynamically updates the T-values of the remaining edges after each edge-selection process, the steps of execution of a new SOA heuristic based on this concept could be demonstrated as shown in Section 2.4.3 for the access sequence `aadeaeadccdbcdeedbda` to yield a better cost value of 7 units.

2.4 A New Approach to SOA

The heuristic proposed in Leupers and Marwedel's paper [15] is implemented using the same technique as is mentioned therein, with the only difference that after an edge is added to the set of qualified edges forming the MWPC, the T-values of all the other edges are updated accordingly. In other words, the weight of the edge that is

selected during the process of finding an optimal path cover is subtracted from the T-values of all the other edges having a common end-point with either node of the selected edge.

2.4.1 The Heuristic

The pseudo-code of our proposed heuristic for single offset assignment (SOA) taking the above into account is shown in Figure 2.5.

```
SOLVESOA_NEW

1.  Given: Access sequence; set SELECTED_EDGES = { }.
2.  Construct the access graph (G = (V, E, w)) and sort the edges in descending order.
3.  For every edge (u, v) in the sorted edge set in that order
4.  {
5.        if the edge when considered with already selected edges (does not cause a cycle)
          and (does not result in choosing more than two edges at any node)
6.        {
7.              if the edge has a unique weight
8.              {
9.                    SELECTED_EDGES ← SELECTED_EDGES ∪ {(u, v)};
10.             }
11.             else for all edges with the same weight   // maximum value of edge wt.
12.             {
13.                   Select edge (u, v) from among the equal-weighted edges having
                      the lowest T-value. // if more than 1, pick from them at random.
14.                   SELECTED_EDGES ← SELECTED_EDGES ∪ {(u, v)};
15.             }
16.       }
17.       Subtract w(u, v) from the T-values of all edges incident at its end-nodes
          u and v.
18. }
```

Figure 2.3: Pseudo-code for SOA Heuristic

2.4.2 Explanation and Working

The input to the heuristic shown in Figure 2.3 consists of the original access sequence, from which the access graph is constructed (lines 1 and 2). The edges are sorted in descending order according to their weights (line 2). The edge selection criteria

are the non-formation of cycles with the other edges of the MWPC that have already been selected and the non-concurrence of more than two selected edges on any node in the graph for an edge to be added to the set SELECTED_EDGES, which is initialized to NULL at the beginning (line 1). The edge with the highest edge weight is considered first and is tested for satisfying of the concurrence and cycle constraints (line 5). If it satisfies the constraints and its weight is unique, then it is added into the set SELECTED_EDGES (lines 7-10). If there is more than one edge with the same value of edge weights, then these edges are sorted in increasing order of T-values, and the edge with the lowest tie value is considered first and added to the set if it meets the edge selection criteria (lines 11-15). If even the T-values are equal, then a random selection is made from among those and added to the set SELECTED_EDGES if the eligibility criteria are met (line 13). All of these edges are considered first before the next set of lower-weighted edges is considered. The above process is repeated until all the edges in the graph have been explored (lines 3-18), with the primary edge sorting parameter being the edges in decreasing order of weights. Each time an edge is considered, the T-values of the other edges having a common node with the considered edge are decreased by the weight of the selected edge (line 17). At the end of the execution of the heuristic, the edges in the set SELECTED_EDGES form the MWPC. Then, the final SOA cost = 1 + $\sum$ (edge-weights of original graph NOT in the set SELECTED_EDGES), where the value 1 is due to the cost of address register initialization.

2.4.3 Example

For a better understanding of the heuristic presented above, and also for a relative comparison with the method adopted by Leupers and Marwedel [15] as discussed in

Section 2.1, the heuristic shown in Figure 2.3 can be demonstrated on the example access sequence of Sections 2.1 and 2.3, wherein the heuristics of [15] and [19] were demonstrated.

2.4.3.1 Execution Steps

The access sequence is: `aadeaeadccdbcdeedbda`. The access graph for this access sequence is shown in Figure 2.1. Like in the Leupers and Marwedel's [15] heuristic, the edges of the graph in the descending order of edge weights along with their initial T values are as shown in Table 2.1.

Table 2.1: Initial edge weights and T-values for the access sequence of Section 2.1

Edges	(a, d)	(d, e)	(e, a)	(d, c)	(d, b)	(b, c)
Edge weights	3	3	3	3	3	1
T values	15	15	9	13	13	7

It is seen from Table 2.1 that the initial edge parameters are exactly the same as those calculated when Leupers' [15] SOA heuristic was used. Initially, the set SELECTED_EDGES =∅. From Table 2.1, it is clear that among the edges (a, d), (d, e), (e, a), (d, c) and (d, b) with the same maximum edge weight of 3, the edge with the minimum T-value is (e, a). Since it is the first edge to be selected, it is added to the set SELECTED_EDGES, and it can be noted that up to this point, everything has been done identically to that in Leupers' [15] heuristic. But now, before going on to the next edge, the T-values of all the other edges are modified by subtracting the edge weight of this selected edge from the T-values of all the other edges which have a common point (e or a) with the candidate edge. The T-values of the edges can then be updated as in Table 2.2.

Table 2.2: Updated T-values after addition of edge (e, a)

Edges	(a, d)	(d, e)	(d, c)	(d, b)	(b, c)
Edge weights	3	3	3	3	1
T values	12	12	13	13	7

Now, there are two edges (a, d) and (d, e) having the maximum weight (of 3) and the minimum T-value (of 12), and therefore, a selection can be made from among these at random. Let us assume that the edge (a, d) gets selected. Then, with the set SELECTED_EDGES = {(e, a), (a, d)}, the updated T-values are:

Table 2.3: Updated T-values after addition of edge (a, d)

Edges	(d, e)	(d, c)	(d, b)	(b, c)
Edge weights	3	3	3	1
T values	9	10	10	7

Of the remaining equal-weighted edges (d, e), (d, c) and (d, b), edge (d, e) has the minimum T-value of 9. But, it is seen that its addition to the set causes a cycle to be formed with edges (e, a) and (a, d). Therefore, it is discarded, and the updated T-values of the edges after selection of edge (d, e) are shown in Table 2.4.

Table 2.4: Updated T-values after selection of edge (d, e)

Edges	(d, c)	(d, b)	(b, c)
Edge weights	3	3	1
T values	7	7	7

At this point, it should be noted that if instead of choosing edge (a, d) randomly first, we had chosen edge (d, e) first, then edge (a, d) would not have been eligible for selection, and the resulting table of edges and T-values would have been the same as computed in Table 2.4. Of the two equal-weighted edges with equal T-values (d, c) and (d, b), either one can be chosen at random. Let us assume that edge (d, b) is selected. Then, the set SELECTED_EDGES becomes {(e, a), (a, d), (d, b)}, and the updated T-values become:

Table 2.5: Updated T-values after addition of edge (d, b)

Edges	(d, c)	(b, c)
Edge weights	3	1
T values	4	4

At this point, edge (d, c) would be the next candidate and it cannot be selected because its selection would result in the concurrence of more than two edges at node 'd', edges (a, d) and (d, b) having already been added to the set. If edge (d, c) had been selected first randomly instead of edge (d, b) first as was done previously, the same logic would have applied to the ineligibility of edge (d, b), and the heuristic would have reached the present instant identically. The remaining edge (b, c) satisfies both the

constraints, and is added to the set SELECTED_EDGES, which now contains the edges {(e, a), (a, d), (d, b), (b, c)}.

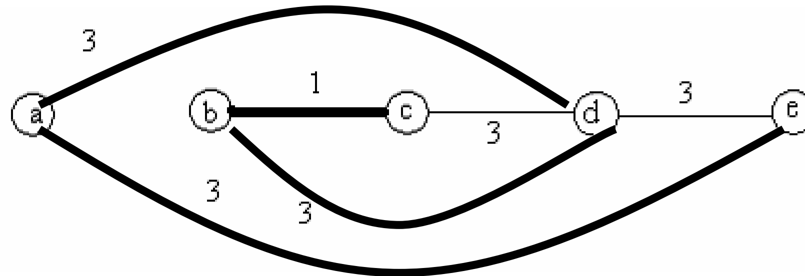

Figure 2.4: Final selection of edges for the MWPC

The final edge selection is shown in Figure 2.4 above. It is seen that a suitable offset assignment would be eadbc, and the total cost would be the sum of edges not selected into the set SELECTED_EDGES which would be w(c, d) + w(d, e) = (3 + 3) = 6 units (plus one due to AR initialization – LDAR for variable e) = 7 units.

2.4.3.2 Comparison and Analysis

The results obtained by the Leupers and Marwedel's [15] heuristic and the heuristic shown in Figure 2.3 indicate that the cost incurred by the access graph of the example access sequence was lower when the MWPC was obtained using our heuristic. The effect of our dynamic approach was studied further by implementing both the heuristics and running them on the same example access sequences, and the results are shown in Section 2.4.4.

2.4.4 Results of the SOA Heuristic Implementation

The heuristic presented above in Figure 2.3 (Section 2.4.1) was implemented on several sample randomly generated access sequences, each set having a different set of access sequence parameters. In addition, we implemented a variation of that heuristic, in

which the updating of T-values of remaining edges is carried out only if an edge is added to the set SELECTED_EDGES; this corresponds to moving line 17 of the heuristic in Figure 2.3 to the position between lines 15 and 16. Ten iterations were performed for each set of access sequences having the same parameter specifications; 500 randomly generated sequences were used as inputs for each iteration. The results so obtained were compared every time with the results of the implementations of the original Leupers and Marwedel's [15] heuristic on the same set of access sequences, and tabulated as is shown in Table 2.6.

Table 2.6: Results of the SOA implementation

No.	Length of access sequence	No. of variables in the access sequence	Avg. Cost (Leupers')	Avg. Cost (Our method)	Avg. Cost (Our variation)	Avg. saving in cost by our method (%)	Avg. saving in cost by our variation (%)
1	10	5	1.634	1.638	1.637	-0.2447	-0.1836
2	20	5	5.057	5.058	5.058	-0.0002	-0.0002
3	50	10	23.756	23.794	24.068	-0.1600	-0.0131
4	50	20	24.472	24.469	24.474	+0.0123	-0.0082
5	100	20	60.445	60.493	60.461	-0.0794	-0.0026

2.4.5 Observations and Analysis of the SOA Results

As is seen from the data in Table 2.6, it turns out that the running of both the heuristics on the sample access sequences yielded results with variable average cost values in all the cases. Though the average result displays fluctuations in terms of the costs, the difference in the heuristics resulted in changes in the exact edges selected during the process of MWPC computation in some cases, as was also seen in the example illustrated in this chapter. Form a practical standpoint, it appears to be beneficial to run

all the three heuristics (two from this thesis and one from Leupers and Marwedel [15]) and choose the best of these.

2.5 Chapter Summary

The heuristic proposed by us in Section 2.4.1 differs from the Leupers and Marwedel's [15] heuristic in the addition of the extra step of updating the T-values of all the edges in the access graph that share an end-point with those of the latest edge that has been selected during the selection process for computing the MWPC. It was seen in the example graph that the cost resulting out of implementing our heuristic was better than that of Leupers and Marwedel's [15], and this motivated the research on multiple random access sequences. The results of the implementation of the heuristics including a variation of ours on several random test sequences showed that there were variable differences in the average costs of the access sequences, depending on the access sequence parameters such as length and number of variables. And, even though the procedure to be followed in the event of two edges having the same edge weights and the same T-values has not been addressed in either of the heuristics, there are a lot of possible ways of approaching that problem, including but not limited to the use of secondary tie-break functions based on the degrees of the end-nodes of the edges, the difference between the dynamic and static T-values of the particular edges, etc. From the point of view of using these in code generation, it appears that choosing the best result from our heuristic variations and Leupers and Marwedel [15] is practically useful.

3. THE GENERAL OFFSET ASSIGNMENT PROBLEM

It was mentioned in Section 1.2 that the availability of more than one address register for the solution of the offset assignment problem yields better results in terms of the cost of the access graph. In this chapter, we propose and evaluate a new heuristic for the general offset assignment problem.

3.1 Leupers and Marwedel's GOA Heuristic

The access sequence is constructed from the code of the specific application, and an access graph is constructed out of the access sequence as is described in Section 1.2. This access graph serves as the input to the Leupers and Marwedel's [15] GOA heuristic, which consists of two phases, phase one and phase two. We assume that the number of address registers available is k. In Phase one, the set of variables is partitioned into k partitions each consisting of at most two variables. Phase one begins with the creation of empty partitions $V_1, \ldots, V_k$ where k is the number of address registers available, and each partition would eventually contain either two variables or no variables. From the list L of edges sorted in descending order of weight, the first element (the edge with the maximum weight) is selected, and its end-points are assigned to the partition V_1. If there is more than one such edge, then the T-values of the edges are also computed, and the edge weight with the minimum T-value is selected first. The selected edge is deleted from L; in addition, all edges incident at the two end-points of the selected edge are also deleted from L. After this, the next element of the list L (in decreasing order of weights) is selected using tie-breaking (if needed) as described and the endpoints of this edge are assigned to the partition V_2, and the selected edge along with any edges incident at the

two end-points of the selected edge are deleted from L. This process is repeated over again, and continued until (i) either all of the partitions $V_1,\ldots,V_k$ have nodes (i.e., variables) assigned to them and there are zero or more nodes of the graph remaining to be assigned to one of the partitions $V_1,\ldots,V_k$, or (ii) until the list L becomes empty as a result of reaching the stage when all the edges of the graph have been deleted at some point of time, whichever is earlier. Any remaining nodes in case (i) will be handled in Phase two, which attempts to assign them to one of the partitions $V_1,\ldots,V_k$. Phase one of the heuristic ends at this point of time; by then if the number of edges whose end-points have been assigned to the partitions is m, then m is the number of address registers that will be used in the process of coming up with the best solution. Note that $m = k$ corresponds to case (i) and $m < k$ corresponds to case (ii).

The access sub-sequence induced by a set of variables Z is derived from the original access sequence by picking every occurrence of all and only the variables in Z from the original access sequence in the original order. The second phase of the heuristic is executed only in the event that there are unassigned nodes remaining by the time the first phase is completed, i.e., nodes in the access graph that have not yet been assigned to any of the partitions $V_1,\ldots,V_k$ by the end of Phase 1. In such a case, a node x is picked from the remaining nodes that have not been selected. For each of the cases $V_1 \cup \{x\}$, $\ldots,V_k \cup \{x\}$, the access sub-sequence induced by the variables $V_1 \cup \{x\}, \ldots,V_k \cup \{x\}$ are constructed and the SOA heuristic is used on each of these access subsequences to compute the SOA costs; let these costs be $c'_1,\ldots,c'_k$. In addition, for each of the cases V_1, $\ldots,V_k$, the access sub-sequence induced by the variables $V_1, \ldots,V_k$ are constructed and the SOA heuristic is used on each of these access subsequences to compute the SOA

costs; let these costs be $c_1,\ldots,c_k$. The selected variable x is assigned to that partition V_* for which the difference $c'_i - c_i$ (where i is between 1 and k) is the lowest. This process is repeated for all the remaining nodes (or variables) that have not been picked, and each time during the difference calculation, the updated latest available partitions are considered. Finally, since each partition corresponds to an address register, the variables in each partition are assigned to the corresponding address register, and so the cost of the system would be the number of registers (each requiring one unit for initialization), in addition to sum of the SOA costs of the access sub-sequences induced by the variables of each partition. According to Leupers and Marwedel [15], this gives us a solution to the GOA problem by breaking down the original graph into a partitioning of variables, followed by solving the SOA problem on each partition.

Let us consider the following example access sequence for demonstrating the working of Leupers and Marwedel's [15] heuristic exactly as presented therein, with the assumption that there are four address registers available: `afagaeadacababcdefgh`. The weighted access graph G = (V, E) is shown in Figure 3.1.

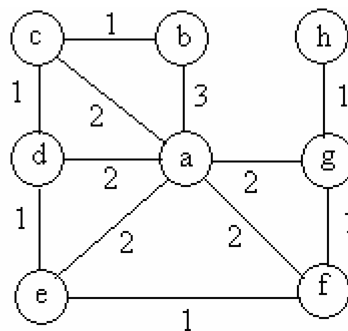

Figure 3.1: Weighted access graph for the access sequence `afagaeadacababcdefgh`

The first phase of the Leupers and Marwedel's [15] heuristic begins with the formation of the list L by sorting the edges in decreasing order as follows.

(a, b)	(a, c)	(a, d)	(a, e)	(a, f)	(a, g)	(b, c)	(c, d)	(d, e)	(e, f)	(f, g)	(g, h)
3	2	2	2	2	2	1	1	1	1	1	1

There is only one edge with the maximum weight 3 which is (a, b). Its end-points a and b are assigned to the partition V_1, i.e., V_1 = {a, b}. Now, according to Leupers and Marwedel [15] all the edges of the graph incident on the nodes a and b are deleted, and the graph after deleting those edges is shown in Figure 3.2.

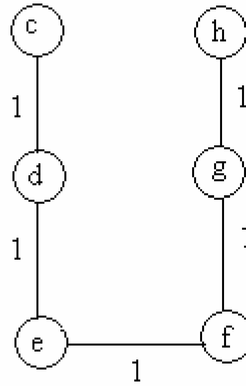

Figure 3.2: Updated access graph after the selection of the edge (a, b).

As is seen in Figure 3.2, the edges that are remaining now after the access graph update are (c, d), (d, e), (e, f), (f, g) and (g, h), all of whose weights = 1. Since all of the edges are of equal weight, we invoke the tie-break function as defined in Leupers and Marwedel [15] here. According to the tie-breaking function: T(c, d) = ∑w(c, *) + ∑w(*, d) - w(c, d) = 1 + 2 - 1 = 2, T(d, e) = ∑w(d, *) + ∑w(*, e) - w(d, e) = 2 + 2 -1 = 3, T(e, f) = ∑w(e, *) + ∑w(*, f) - w(e, f) = 2 + 2 -1 = 3, T(f, g) = ∑w(f, *) + ∑w(*, g) - w(f, g) = 2 + 2 -1 = 3, and T(g, h) = ∑w(g, *) + ∑w(*, h) - w(g, h) = 2 + 1 - 1 = 2. Since the T-values of the edges (c, d) and (g, h) are the least (each = 2) as seen above, a random choice can be made between these edges; let us assume that edge (c, d) is selected. Then, the partition V_2 would be {c, d}. At this point, the number of partitions we have is equal to 2. Now, out of the remaining edges in the graph, all edges of the graph incident on nodes c and d are deleted, and the resultant graph is as shown in Figure 3.3.

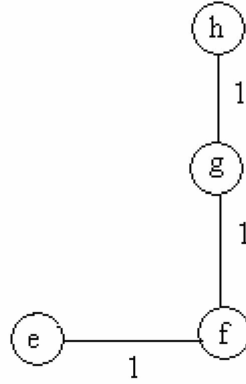

Figure 3.3: Updated access graph after the selection of the edge (c, d).

As is seen in Figure 3.3, the edges that are remaining now after the access graph update are (e, f), (f, g) and (g, h), again all of whose weights = 1. Since all of the edges are of equal weight, we invoke the tie-break function as defined in Leupers and Marwedel [15] here. According to the tie-breaking function: T(e, f) = ∑w(e, *) + ∑w(*, f) - w(e, f) = 1 + 2 -1 = 2, T(f, g) = ∑w(f, *) + ∑w(*, g) - w(f, g) = 2 + 2 -1 = 3, and T(g, h) = ∑w(g, *) + ∑w(*, h) - w(g, h) = 2 + 1 - 1 = 2. Since the T- values of the edges (e, f) and (g, h) are the least (each = 2) as seen above, a random choice can again be made between these two edges; let us assume that edge (g, h) is selected. Then, the partition V_3 would be {g, h}. At this point, the number of partitions we have is equal to 3. Out of the remaining edges in the graph, all edges of the graph incident on nodes g and h are deleted, and the resultant graph is as shown in Figure 3.4.

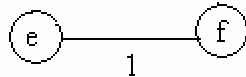

Figure 3.4: Updated access graph after the selection of the edge (g, h).

As is seen in Figure 3.4, the edge that remains now after the access graph update is (e, f), and its end-points are assigned to the partition V_4 which would be {g, h}. At this

point, the number of partitions we have is equal to 4, which is equal to the number of available address registers as assumed by us to begin with.

In the heuristic of Leupers and Marwedel [15], it is mentioned that phase two of the heuristic is executed if there are nodes remaining in the graph after the allocation of nodes to the partitions corresponding to the available address registers. In the example above, it is seen that after the partitions have been assigned nodes, there are no nodes remaining in the graph that have not been assigned to any partition. Therefore, phase two of the heuristic is not executed, and the total cost incurred by the present allocation of variables to the partitions is equal to 4, due to the initialization of the four address registers corresponding to the partitions $V_1 - V_4$. It should be noted that for this particular example, due to the symmetry of the access graph, if at every stage involving a random selection between candidate edges an alternate selection from that shown in the above example was made, then the final cost computed would still be the same, with only the order of assigning certain variables to partitions corresponding to address registers being different.

3.2 Drawbacks of the GOA Heuristic of Leupers and Marwedel

Although the approach adopted by the heuristic of Leupers and Marwedel [15] seems plausible enough to reach a solution, it has some undesirable properties: (i) for some access sequences, the graph at different stages of Phase one are not guaranteed to be access graphs, i.e., graphs that correspond to any access sequence; and (ii) the heuristic may end up using more address registers than necessary for a given cost.

3.2.1 Static Nature of the Access Sequence

The process of selection of edges from the list L after sorting them in decreasing order is carried out statically. In Phase 1, once an edge has been selected from L, its end-points are assigned to some partition V_i, the edge is deleted from L along with all the other edges in L that are incident on either of its end-points. But, the resultant graph does not correspond to any meaningful access sequence or access subsequence. For example, an access graph cannot have isolated nodes.

Let us consider the example from Section 3.1 again, under the assumption that there are four address registers available. If instead of deleting the edges from the original access graph, we consider the access sub-sequences of the original access-sequence, the steps of execution of the phase one of a new GOA heuristic taking this into account could be as shown below for the access sequence `afagaeadacababcdefgh`. First, the access graph shown in Figure 3.1 is constructed. Then, the list of edges sorted in decreasing order of weights is as follows.

(a, b)	(a, c)	(a, d)	(a, e)	(a, f)	(a, g)	(b, c)	(c, d)	(d, e)	(e, f)	(f, g)	(g, h)
3	2	2	2	2	2	1	1	1	1	1	1

Now, it is clear that the edge having the maximum weight has to be selected first, and so its end-points ‘a’ and ‘b’ are assigned to the partition V_1. Then, if instead of deleting the edges incident at ‘a’ and ‘b’ from the graph, all the occurrences of the variables a and b were removed from the original access sequence (implying that the first address register would be responsible for the variables a and b and all the other address registers combined together would be responsible for accesses to the remaining

variables), then an access sub-sequence could be constructed as follows: `fgedccdefgh`.

The well-defined access graph constructed out of this sequence as will be shown in Figure 3.5 clearly conveys the notion of assigning the responsibility of the remaining variables to the remaining address registers for the present. Since a and b have been assigned to partition V_1, the remaining address registers $V_2,\ldots,V_k$ are responsible for accessing the remaining variables in the access sequence in the order in which they are accessed in the original access sequence. For example, in Figure 3.5, the remaining address registers are responsible for providing access to variables c, d, e, f, g, and h. This is in contrast to the access graph of Figure 3.2, which depicts the result of simply deleting edges from the access graph. The access graph referred to above is constructed from the updated access sequence, as shown in Figure 3.5.

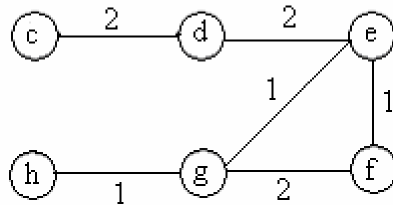

Figure 3.5: Access graph from the updated sub-access sequence "`fgedccdefgh`".

During the execution of phase 1 of the heuristic, the addition of variables to a particular address register implies that the address register is responsible for all the accesses to those variables out of the access sequence, and that all the other registers combined together are responsible for the accesses to other variables not covered by any partition up to that point. This could be visualized more effectively if the access sub-sequence constructed out of the original access sequence by selecting in their order of occurrence the hitherto unpicked variables is considered.

3.2.2 Register Utilization

The heuristic proposed by Leupers and Marwedel [15] is based on the fact that there are 'k' registers available. From the heuristic, it is seen that for any access graph containing |V| vertices and |E| edges, the maximum number of disjoint edges that can exist and whose end-points can be added to the partitions $V_{1},\ldots,V_k$ for each of which an address register is assigned is (|V|/2). There are three possible scenarios:

1. The number of disjoint edges > k
2. The number of disjoint edges = k
3. The number of disjoint edges < k

If the number of disjoint non-zero weight edges is more than the total number of available registers, then the end-points of the first 'k' disjoint edges as gleaned from the list L are selected and stored in the partitions $V_{1},\ldots,V_k$ and there is no need for the execution of phase 2. If the number of disjoint edges is equal to the number of registers available, then the situation is still reasonable with all the available registers being exactly utilized for all the disjoint edges in the list L, and there is again no need for the execution of the phase 2. But however, if the number of disjoint non-zero weight edges is smaller than k while the total number of variables in the access graph is greater than 2k simultaneously, then the heuristic calls for the usage of only as many address registers as there are disjoint non-zero weight edges, rather than all the k available address registers. The utilization of the balance amount of available registers in addition to the ones incorporated by the heuristic could possibly point towards a hitherto unexplored avenue of arriving at a better solution, or at least at a pattern across different problem instances

that could provide us with some useful leads and insights into framing or modifying existing heuristics.

Also, it might be true in some cases that the usage of fewer than k registers can result in a better solution than would be achieved in the case when k registers are used. The heuristic proposed in Leupers and Marwedel [15] does not consider the general case of using any number of registers less than k, and thus investigating the possibility of arrival at a better solution during an intermediate step.

3.2.3 Element of Randomness

If all the available partitions $V_1,\ldots,V_k$ are all non-empty and there are nodes in the graph that have not been assigned to any partition in $V_1,\ldots,V_k$, then there would be (|V|-2*k) variables remaining to be assigned at the end of Phase one. In that case, at the beginning of the execution of phase 2 of the heuristic, there would be a total of (|V|-2*k)! possible combinations to choose from and this issue has not been addressed. Selection of edges in an order determined by generating a random combination from the above may not be the best option in the process of computing the optimum cost.

3.2.4 Edges with Equal Weights

As was discussed earlier, the issue of selection of an edge out of those having equal weights has not been addressed.

3.3 An Analytical Insight

An access graph that is constructed from an access sequence corresponding to a particular code is characterized by the following: the number of vertices V - which is equal to the number of variables in the code, the number of edges E – which corresponds to each pair of distinct consecutive variables in the original access sequence [19], and the

edge weights – the number of such adjacent occurrences of two particular variables in the access sequence. The execution of a round of the GOA heuristic revolves around the availability of address registers. In other words, the complexity and time consumption of the execution of the GOA heuristic (which, in turn, incorporates the SOA heuristic) depends on the number of address registers available. If, without loss of generality, we assume that there are a certain 'k' number of address registers available, then the total number of variables out of V that would be allocated to these address registers during the variable allocation phase of the GOA heuristic would be equal to 2k. If the total number of variables V is less than or equal to 2k, then the computation of the GOA cost is straightforward: the GOA cost would simply be 1 (initialization cost associated with each address register) times the number of address registers used, since the SOA cost of any (sub) graph with only two variables is always equal to zero due to the proximity of the two assigned variables in the offset assignment.

If however, the total number of variables V is greater than the number 2k by some number - say m, then at the end of the first phase of the GOA heuristic, there would be 2k variables assigned to the k registers, and m unassigned variables. These m remaining variables would then be added by turn to one of the k partitions of variables, depending on the difference in SOA cost calculations. This means that, for each unassigned variable, two SOA cost calculations have to be calculated for each of the k variable partitions: one with the unassigned candidate variable in the partition and the other without. With 2k SOA cost calculations necessary for each unassigned variable, and with the procedure having to be carried out for all the m remaining unassigned variables, the total number of SOA cost calculations that have to be performed is 2km. This amount

is $O(m^2)$ or $O(k^2)$ when k and m are of comparable magnitudes, and can be quite large for codes involving a large number of variables.

The possibility of achieving a reduction in the execution time from that necessitated by the performing of 2km SOA cost calculations, in addition to the other salient points discussed in the Section 3.2 are the main motivations behind the postulation of a more general and inclusive heuristic to solving the GOA problem.

3.4 A New GOA Heuristic

Consider the GOA problem for an access sequence A where k address registers are available. If we use only one of these address registers, the cost we incur is 1 + SOA cost of A. Thus, there is no need for the GOA cost to exceed this value. In addition, if the access sequence has V variables, there is no need for more than ceiling (V/2) address registers. Thus, there is no need to use any more registers than the minimum of the two values, (1 + SOA cost of A) and ceiling (V/2). This is one observation used by the new heuristic in Figure 3.6. In addition, we can always use the value of (1 + SOA Cost for the access graph induced by the currently un-assigned variables) at any point in Phase 1 as an upper bound for the GOA cost. Exploiting this leads to both a reduction in the number of registers; also, it eliminates the need to initialize a register, thereby reducing cost.
The general GOA heuristic using some of these observations is shown in Figure 3.6.

```
1. algorithm SOLVEGOA_NEW(AS,k) // AS: Original access sequence k: No. of ARs
2. begin
3. // Phase 1 //
4. AS ← initial access sequence
5. AG ← (V,E,w) // initial access graph for AS
6.  k = min (k, ceiling( |V|/2 ))
7.  p ← SOA cost of AG;
8.  if (p+1 < k) then k = p + 1; // k = min(k, p+1)
9.  if (k= =1) then total cost = SOA cost of the access sequence; EXIT.
10. i = 0;
11. // At the end, V1,….,Vk will contain the variables assigned to registers R1,….,Rk.
12. Vj ← ∅; // for j = 1 through k
13. repeat
14.         i = i+1;
15.        (u, v) : edge of the access graph AG with maximum weight.
16.        AS' ← Access sequence AS with all occurrences of (u) and (v) removed.
17.        AS ← AS'
18.        AG ← access graph formed out of AS.
19.         if (i != k) {
20.                   q ← SOA cost of AG; Vi = (u,v)
21.                    bound ← min (bound, i+1+q)// Initially bound is set to q value for i=0
22.                    save the solution corresponding to bound.
23.                    if (q <= 1) {
24.                            i ← i+1
25.                            Vi ← all nodes in AG //Ri will be used to access all variables in AG
26.                                     EXIT;  // from the entire algorithm
27.                    }
28.        }
29. } until (i = = k)
30. // Phase 2 //
31. do for all unassigned variables v {
32.         V* ← V* ∪ {v} // V* is that partition from V1,...,Vk for which the SOA cost
            difference with and without v is minimum.
33.  }
34.  cost = the smallest sum of SOA costs of all partitions V1,…,Vk.
35.  if (cost + k < solution corresponding to bound) {
36.           retain this variable assignment as the solution.
37.  }
38.  else {
39.  go back to the saved solution.
40.  }
41.  end.
```

Figure 3.6: Pseudo-code for the new GOA heuristic

3.4.1 Explanation and Working

The inputs to the pseudo-code shown in Figure 3.6 consist of the original access sequence (AS: access sequence derived from the original program on the system) and the number of address registers (k) available. Phase one of the heuristic begins with the construction of the access graph AG from the access sequence AS. Since the number of address registers available is a parameter that is independent of the number of memory locations corresponding to the variables in the access sequence, the value of k could be such that its double could either be greater than, equal to or less than the total number of variables in the access sequence and therefore also the access graph. An important observation noteworthy at this point is the following: if the number of available address registers is less than or equal to the ceiling of |V|/2, then a maximum of k address registers would be used and therefore the value of k remains fixed. Further, there would be (|V|/2)-k (which is 0 in the case when they are equal) unassigned nodes at the end of phase 1. However, if there are more address registers available than |V|/2, then in the worst case we would use a maximum of |V|/2 address registers only, because at least two variables would be assigned to each address register partition. In this case, k would assume the ceiling value of |V|/2, and that is depicted in the statement in line 6 of the heuristic shown in Section 3.4.

The SOA cost (p) of the original input access sequence is calculated, as seen in line 7. This plays a pivotal initial role in calculating the upper bound on the number of registers to be used, the motivation behind computing this value being the following: let us assume that p+1 is less than k by a certain value q. Then this means that with a single address register, the total cost of the original access sequence is p units + 1 for

initialization = p+1. Now the total number of address registers that can be used varies from 1 through k. But for any number of address registers more than one and less than k, nothing can be said about the system cost in advance, and therefore that aspect is taken care of in the loop of the pseudo-code, as shall be discussed later. Then the worst case scenario for variable assignment would occur when only two variables each can be assigned to each of the k variable subsets corresponding to their respective address registers. Under this situation, the total cost of the access sequence would be equal to k*0 (because the SOA cost of a graph having two nodes is always equal to zero) + k*1 (initialization cost for each address register) = k. But for any k > p+1, the use of all the 'k' available registers is uneconomical due to the already known fact that this would incur a cost of k units as seen above, while it is already pre-calculated that the cost of the sequence with one address register is equal to p+1, which is less than k by the amount q that we defined above. Using more than p+1 address registers for the access sequence when p+1 < k is uneconomical due to the fact that the usage of every additional address register would incur an additional cost of at least 1 unit due to initialization thus making the total cost exceed the known SOA cost, the upper bound of the number of address registers used is kept at p+1, the SOA cost of the original access sequence. This update of the value of k, the number of resources to be used is shown in line 8 of the heuristic. Also, it is obvious that if p+1 > k, then again nothing can be said about the total cost with different number of address registers, and so the value of k, the number of available registers is kept intact.

The value of i, the iteration index, is initialized to zero, and k number of partitions, which would eventually at the end of the heuristic's execution contain the

variables of the access sequence that would be assigned to address registers R_1 through R_k, are initialized to NULL as shown in lines 11 and 12 respectively. The loop of the heuristic in phase 1 runs through k iterations, each instance signifying the occurrence of the event of 'i' address registers being used. At the beginning of every iteration, the edge with the maximum edge weight is identified (line 15), and all the occurrences of its end-points are removed from the access sequence, and its end-points are assigned to the ith address register. If there is more than one such edge, then the edge having the maximum T-value from among these is chosen. With the new access sub-sequence, a new access sub-graph is constructed and that serves as the input graph for the successive iteration, as is shown in the lines 14-18 of the heuristic. The SOA cost of the new graph is constructed, and as long as the variable i has not yet reached the value of k, the SOA cost is stored as q separately. Here, there are two circumstances that can arise: the total cost of the system with the additional address register 'i' can be either less than or equal to the old value of (q+i+1) or greater than the old value of (q+1+i) which is stored as the value of bound prior to updating in the current iteration. The smaller of these two values is taken to be the better cost of the access sequence with a total of 'i' address registers being used, and this value is saved as we go along for each iteration, as shown in lines 20 and 21. If the value q is equal to 0 or 1, then this means that the solution obtained in the current iteration is the best solution, and so all the remaining unassigned variables get assigned to the ith address register and the execution stops, without the necessity of phase 2. This is what is depicted in the lines 22 through 26. But if the value of q is greater than 1, then the loop continues executing on the account of the fact that nothing can be

predicted in advance regarding the cost of using additional address registers. The loop executes up to a maximum of k times before termination.

The second phase undergoes execution only in the event of and after the completion of a full round of k iterations of phase 1, and only if there are unpicked nodes remaining. Just like in Leupers and Marwedel's [15] heuristic, these unpicked nodes are assigned sequentially to that partition from among $V_1,\ldots,V_k$ where the difference in SOA costs with and without the addition of the unpicked node is a minimum. After this process is completed for all the remaining unselected nodes, the cost of this assignment of variables is calculated, and is compared with the value of 'bound' corresponding to the availability of 'k' registers. If this solution is less than 'bound', then the variable assignment corresponding to the value of this solution is finalized, else the saved solution corresponding to the value of 'bound' is finalized (lines 31 through 40 of the heuristic).

3.4.2 Example

The above heuristic can be understood better using the example presented in Section 3.1. Consider the access sequence: `afagaeadacababcdefgh`. Initially, the value of AS = "`afagaeadacababcdefgh`" and AG is as shown in Figure 3.1. Initially k = 4, i = 0, p = SOA cost of AS = 9, and bound = 10. Now, i = i + 1 =1. The edge with the maximum weight (u, v) = (a, b) with a weight of 3. So, AS' = AS with all the occurrences of `a` and `b` removed, which is "`fgedccdefgh`". Now, AS $\leftarrow$ AS', and so AS = `fgedccdefgh`. AG is now the access graph formed out of this access sequence AS and this is shown in Figure 3.5.

Now, $i \neq k$ ($1 \neq 4$), q = SOA cost of the AS = 1, and V_1 = (a, b).

bound = min(10,1+1+1) = 3. Since q = 1, i = i + 1 = 2, and V_2 = (c, d, e, f, g, h).

Here, it is seen that the total cost of the system is 3 and the number of address registers used is 2. It was seen in Section 3.1 that if the above problem were to be solved by the heuristic of Leupers and Marwedel [15] under the assumption that there are four address registers available, then the total cost incurred is 4 (one for the initialization of each address register).

3.4.3 Results

The GOA heuristic presented above was implemented and run on several sample access sequences. The results so obtained are compared with the results of the implementations of the original Leupers and Marwedel's [15] heuristic in Table 3.1. For all the sample sequences in Table 3.1, we assumed 8 as the number of registers available.

Table 3.1: Results of GOA on sample access sequences

		Leupers GOA			**Our Heuristic**				
A	B	C	D	E	F	G	H	I	J
afagaeadacababcdefgh	8	4	{ab}{cd} {ef}{gh}	4	2	{ab} {edfgch}	3	1	2
bfbcdefefafa	8	2	{afe}{bcd}	2	2	{ef}{bcda}	2	0	0
abacdcefeghgijiehdfefba	8	5	{ef}{ab} {gh}{ij} {cd}	5	4	{ef}{ab} {gh}{cdij}	5	0	1
abacacadadadaefgfgheh	8	3	{adb}{gf} {ehc}	3	2	{ad} {bcehgf}	3	0	1
ababcdcdefefefdfdcecfb	8	3	{ef}{cd} {ab}	2	2	{ef} {abcd}	2	1	1
dcbfeadacaba	8	2	{acb} {efd}	4	2	{ab} {dcef}	3	1	0
abcdabcdefcfecg	8	3	{abg} {ef}{cd}	3	2	{cf} {abdeg}	3	0	1
habgfcdegfh	8	3	{gf}{cbed} {ha}	3	2	{gf} {habcde}	3	0	1
abcdefabe	8	2	{ab} {efcd}	2	2	{ab} {cdef}	2	0	0
abcbcdadefa	8	3	{bc}{da} {ef}	3	2	{bc} {adef}	3	0	1

Legend for Table 3.1:

A: Access sequence
B: No. of ARs available (assumed)
C: No. of ARs used by Leupers' [15] heuristic
D: Allocation of Variables to ARs using Leupers' [15] heuristic
E: Cost using Leupers' [15] heuristic
F: No. of ARs used by the proposed heuristic
G: Allocation of variables to ARs using the proposed heuristic
H: Cost using the proposed heuristic
I: Saving in cost (E-H)
J: Saving in the number of registers used. (C-F)

Table 3.2: Results of the GOA heuristic on randomly generated access sequences assuming the availability of k = 16 address registers.

Length of access sequence	Number of variables in the access sequence	Number of available AR's assumed	Average Cost (Leupers')	Average Cost (our method)	Average saving in cost (%)
10	5	16	2.21	2.02	+8.59
20	5	16	2.71	2.66	+1.84
50	10	16	5.18	4.96	+4.25
50	20	16	7.72	8.18	-5.96
100	20	16	9.61	9.84	-2.39

Table 3.3: Results of the GOA heuristic on randomly generated access sequences assuming the availability of k = 8 address registers.

Length of access sequence	Number of variables in the access sequence	Number of available AR's assumed	Average Cost (Leupers')	Average Cost (our method)	Average saving in cost (%)
10	5	8	2.21	2.02	+8.59
20	5	8	2.71	2.66	+1.84
50	10	8	5.18	4.96	+4.25
50	20	8	7.63	8.05	-5.50
100	20	8	9.84	9.41	+4.37

Table 3.4: Results of the GOA heuristic on randomly generated access sequences assuming the availability of k = 4 address registers.

Length of access sequence	Number of variables in the access sequence	Number of available AR's assumed	Average Cost (Leupers')	Average Cost (our method)	Average saving in cost (%)
10	5	4	2.21	2.02	+8.59
20	5	4	2.71	2.66	+1.84
50	10	4	5.46	5.27	+3.48
50	20	4	9.41	8.94	+4.99
100	20	4	24.08	23.24	+3.49

Table 3.5: Results of the GOA heuristic on randomly generated access sequences assuming the availability of k = 2 address registers.

Length of access sequence	Number of variables in the access sequence	Number of available AR's assumed	Average Cost (Leupers')	Average Cost (our method)	Average saving in cost (%)
10	5	2	2.23	2.01	+9.87
20	5	2	2.71	2.67	+1.48
50	10	2	13.68	13.46	+1.61
50	20	2	17.54	17.21	+1.88
100	20	2	45.07	44.66	+0.91

3.4.4 Observations and Analysis

The GOA heuristic, which was proposed as in Section 3.4, yielded the results shown in Tables 3.1 through 3.5 when run on the sample access sequences. Table 3.1 shows the results for specific sample sequences. Tables 3.2-3.5 show the results for randomly generated sequences. In many cases, these results display improvements over the results obtained by the execution of the heuristic of [15], either in terms of a reduction

in the cost or in terms of a reduction in the number of address registers required, or both, as is evident in some cases. This is due to the integrated approach adopted by the heuristic presented above, which considers the remnants of the original access sequence at every stage once a selection of an edge has been made, the variables assigned to the address register, and the graph updated. However, it is to be noted that even in some cases where there is no gain either in the cost or in the number of address registers saved, the execution time of the proposed heuristic is less when compared to the heuristic of Leupers and Marwedel [15].

Tables 3.2 through 3.5 show that our heuristic when run on randomly generated access sequences yields variable results under conditions of availability of different number of address registers, for each set of parameters, i.e., combination of length of access sequence and number of variables. For smaller value of k (the number of available registers), our heuristic always outperformed Leupers and Marwedel [15]. For k=8 and k=16, the results were mixed.

3.5 Chapter Summary

In this chapter, a heuristic was proposed for the solution of the Global Offset Assignment problem which involves the assignment of variables of the application code to different address registers, and the address registers' responsibility towards all the accesses of all their assigned variables. This heuristic was an improvement over that proposed in [15] in terms of clarifying the meaning of the updated access graph constructed immediately after the selection of the end-points of an edge into the partition corresponding to a particular address register, and also in terms of the number of required address registers at every stage of the heuristic execution. In other words, the

incorporation of address registers to the existing collection was done as needed, as opposed to the heuristic in [15] which starts off with a certain number of address registers and does not explore the possibility of there existing an upper bound (namely, the SOA cost of the entire access sequence) to the number of address registers needed. The new heuristic was implemented and run on several random access sequences, and the results so obtained were compared with the results obtained by the heuristic of Leupers and Marwedel [15]. For smaller value of k (the number of available registers), our heuristic always outperformed Leupers and Marwedel [15] with as much as 10% savings. For larger k, the results were mixed.

4. VARIABLE ASSIGNMENT INTO MEMORY BANKS

The availability of multiple memory banks in processors requires the assignment of variables to locations all within a contiguous unit, as is illustrated in [16]. However, in modern day DSP and embedded processors, the architectural design of the memory component of the processor is carried out under various design considerations, such as power and energy consumption, size limitations, and other performance characteristics such as speed and execution time reduction. In addition to these, due to software constraints [31] and schedule deadlines, a common practice followed during the design of memory components in these processors is the practice of equipping these DSP processors with two separate memory banks [22] from which parallel readability is realized, as is shown in the black-box representation of the Gepard DSP Architecture [16] shown in Figure 4.1.

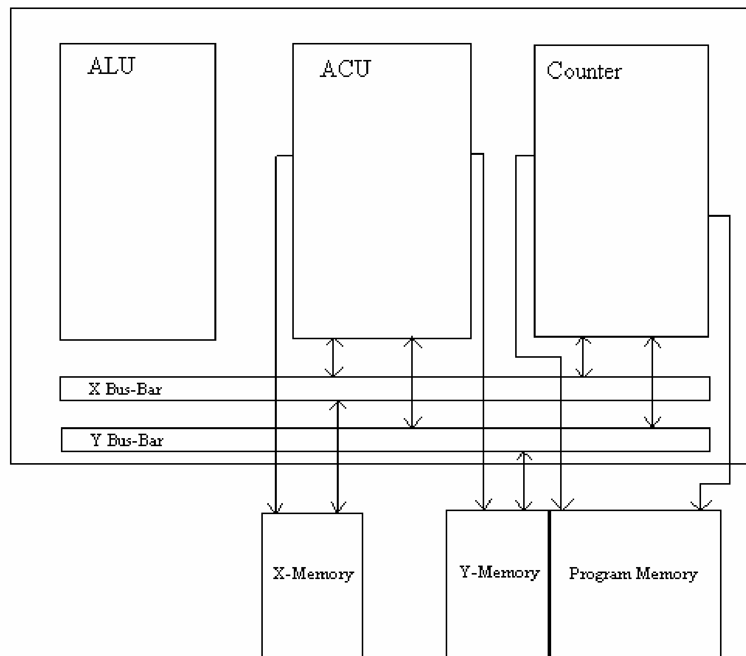

Figure 4.1: Black box model of the Gepard DSP architecture [16].

It is seen in Figure 4.1 that both the memory banks X and Y are connected to the bus bars X and Y that serve as an interface between the address calculating unit (ACU) and the two memory banks. For this design, the variable partitioning problem can be defined as the problem of choosing an optimal allocation of the program variables to these memory banks. In attempting to achieve higher levels of code quality, and also to solve the problem of efficient bandwidth allocation and distribution, it should be noted that the efficient solution of the variable partitioning problem is as significant and complex as the issue of finding an optimal offset assignment, the assignment of variables to memory locations within an array of contiguous registers. This chapter deals with research into the problem of partitioning variables automatically between two memory banks in processors, further to the research done by Leupers and Kotte [16].

4.1 Overview

The heuristic proposed in [16] for solving the problem of finding an optimal distribution of code variables between two partition blocks, say X and Y, is done using two steps: in the first step, a data-dependency graph is constructed from the processor application code by denoting each memory access of the code by a node which is labeled by the name of the variable accessed. In this graph, the edges between two nodes represent scheduling precedences between them. Then, from this, in the second step, an interference graph [24] is constructed in which edges between two nodes exist only if the nodes are not accessible to each other in the data-dependency graph. From this, a folded version is created wherein each node represents one unique variable of the code and the weight between two nodes represents the total number of accesses to the variables represented by the two nodes in the code. On this interference graph, the partitioning

heuristic is executed and variables assigned to partition blocks X and Y after computing the sets of variables in X and Y for which the sum of edge weights between the variable sets X and Y is maximized [16]. The reason for this is that a high edge weight between two nodes of the interference graph indicates the number of possible parallel memory accesses of these two variables. The nodes of the interference graph correspond to the variables in the original code, and the weight of an edge between any pair of nodes in the interference graph represents the number of times there exists the possibility of these two nodes being assigned to different variable banks. Since a high edge weight between two nodes indicates that there are a high number of instances where a gain associated with allocating the end-point variables to opposite memory banks and accessing them in parallel is achieved, the problem of finding an optimal allocation of the set of variables of the code into the memory banks translates to finding an optimal max-cut of the interference graph. Since the edges of the interference graph indicate the possible benefits of allocating the end-point variables to different banks, the term “interference graph” will henceforth be referred to as the “benefit graph” for the remainder of this thesis. However, the heuristic presented in Leupers and Kotte [16] does not take into account the initial partitioning of variables for memory bank assignment, and therefore an attempt is made to formulate a general heuristic that addresses this issue and adapts hitherto known heuristics for NP-complete problems.

4.1.1 Example

In order to illustrate the construction of the benefit graph, let us consider the following example dependency graph of a particular code.

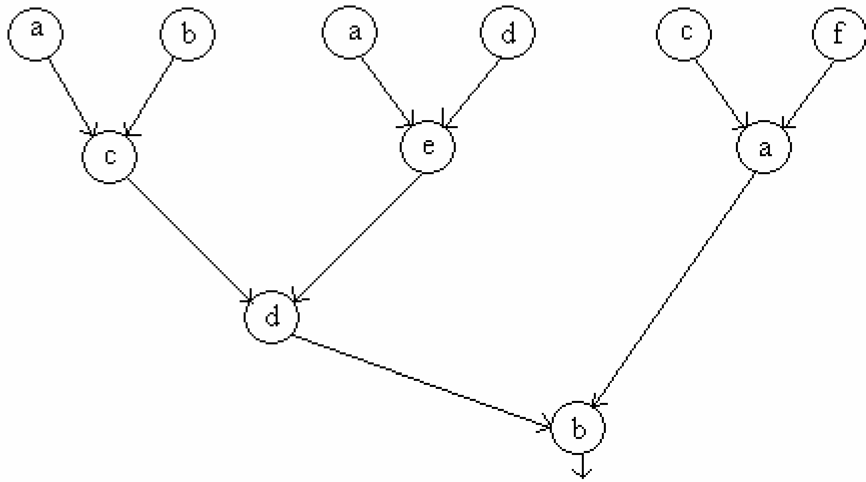

Figure 4.2: Example Data Dependency Graph

Following the procedure explained in Section 4.1, the benefit graph for the data dependency graph shown in Figure 4.2 can be constructed as shown below:

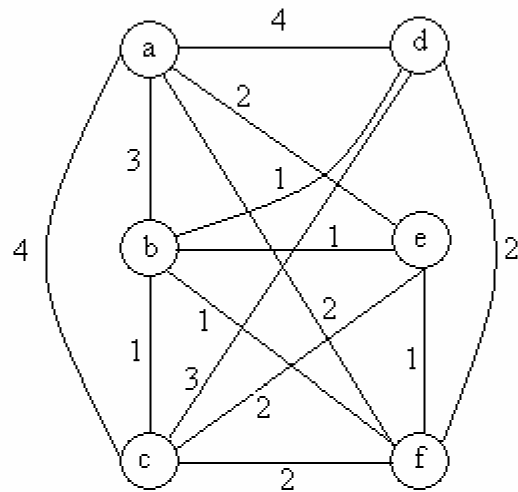

Figure 4.3: Benefit graph for Figure 4.2

As is seen in Figure 4.3, the benefit graph consists of 6 nodes, each node corresponding to a distinct variable in the original data dependency graph.

4.2 The Variable Partitioning Problem

It has been discussed in [8, 29] 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 is NP-complete. The same is true for the problem of maximizing the sum of the edges between X and Y. Kernighan and Lin [11] have proposed a heuristic for this problem using node swapping But this heuristic assumes the fact that at any given point of time during the execution of the heuristic, the number of nodes in both the sets X and Y remains constant throughout, which masks the possibility of computing a more optimal solution (if any) that could possibly be arrived at if the existence of a certain degree of imbalance [8] between the sets of variables X and Y was tolerated. 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 would need some thought and consideration. In the following section, we propose a heuristic that incorporates 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.

4.3 A General Approach

In consideration of the above, general heuristics can be formulated for the bank assignment problem, taking into account partition imbalance assumptions. In the heuristic presented by Leupers and Kotte [16], a discussion on the construction of the data dependency graph and subsequently, the benefit graph is presented, and this can be modified and expressed on the lines of the version of Kernighan and Lin [11] described in Yelick [29] as well as using Fiduccia and Mattheyses' heuristic [8].

Let the weighted benefit graph be G = (V, E, w). Assume a partitioning of the form $V = X \cup Y$, where X and Y are the two partitions into which variables would be assigned eventually. Let S be the maximum allowable size of each of the banks X and Y. An initial partitioning could be derived using the heuristic in Figure 4.4.

// Initial assignment of variables into memory banks X and Y:

- From the benefit graph, sort the edges in descending order of weights
- Select all the possible disjoint edges (u, v) in descending order and assign all such edges: node u to block X and node v to block Y.
- For all the remaining nodes (if any), place them at random one by one into the variable set X or Y for which the sum of the edge weights between the node and the nodes in the opposite set is a maximum, and the number of variables in X and Y are each no greater than S.

Figure 4.4: A heuristic for an initial partition.

Assume that the sets X and Y have been initially formed as explained in Figure 4.4. We use the following definitions.

- The total cost of the two sets X and Y, denoted by T = cost(X, Y) is defined as the sum of the weights edges one of whose end-points is in X and the other end-point is in Y. Thus $T = \sum \{w(e)$ where e connects nodes in X and Y$\}$.
- We define the *internal cost* of a node u (denoted I(u)) 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. I(y) for nodes y in the set Y are defined similarly.
- We define the *external cost* of a node u (denoted E(u)) 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, $E(x) = \sum w(x, y)$ for other y not in the set

X; with just two sets X and Y, this means that y belongs to the set Y. E(y) for nodes y in the set Y are 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:

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. **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).

Using these, for the case of bank assignment, the objective is to find the partitions X and Y of the graph yielding the maximum cost [16], and therefore the problem translates to

the max-cut problem, subject to the constraint that the transfer of the node ‘p’ from one partition to the other yielding the maximum gain does not cause the size of any partition to be greater than the pre-specified value ‘S’.

4.3.1 Heuristic

If the maximum possible size S of a partition block is less than |V|/2, then it would be impossible to distribute |V| variables between the two blocks, and if S is greater than |V|, then there exists the possibility of all the |V| nodes of the graph being assigned to one partition block. With the assumption that the maximum allowable size of banks X and Y is S, such that $|V|/2 < S < |V|$, or in other words, $2S > |V|$, a heuristic taking into account the view in Section 4.3 is expressed in pseudo code form as shown below.

```
ALGORITHM BANK_ASSIGNMENT(X, Y, w)

1.  Obtain an initial partition X, Y using heuristic of Figure 4.5
2.  Compute T = cost(X, Y) for initial X, Y;
3.  Repeat {
4.        Compute costs D(u) for all u in (X ∪ Y)
5.        Unmark all nodes in X ∪ Y
6.        while (there are unmarked nodes) {
7.                Find an unmarked node p in (X ∪ Y) maximizing gain(p).
8.                Mark p but do not transfer it.
9.                if (transferring p from its current partition to other partition results in the
                      other partition having size > S)
10.                       Mark all the nodes in the current partition.
11.               else
12.                       Update D(u) for all unmarked u as though p had been
                           transferred, and save the value of gain(p) and p.
13.       } endwhile;
14.       Pick ‘k’ for which Gain = ∑gain(p_i) for i = 1 through k is maximum; 1≤k≤m.
          // m = no. of saved ‘p’s.
15.       if (Gain > 0) then {
16.               Update new set X.
17.               Update new set Y.
18.               T = T + Gain.
19.       } endif;
20. } until (Gain ≤ 0);
```

Figure 4.5: Pseudo-code for the general variable partitioning heuristic

4.3.2 Explanation

The heuristic presented in Figure 4.5 begins with the computation of the initial cost for the initial partitions X and Y arrived at using the heuristic of Figure 4.4, as is seen in lines 1-2. Iterations (lines 3-20) are carried out until the value of Gain (calculated in the iteration) is non-positive, and each iteration, which computes the maximum Gain resulting out of the transfer of certain nodes in V, consists of the following: for each u in V, the value of D(u) is computed, and all the nodes in V are unmarked (lines 4-5). The iterative process of finding an unmarked node resulting in the maximum gain, marking it, and updating the 'D' values for all the other unmarked nodes as though the node has been transferred subject to the fulfillment of the partition size constraints is repeated for as long as there are unmarked nodes remaining (lines 6-13), and the nodes and gains are saved separately. The partition size constraint is taken into account as follows: if the transfer of a node 'p' from its current partition to the opposite partition results in the size of the latter to a value greater than the size limit S, then all the nodes of the current partition are marked and locked, to ensure that the partition size stays within the constraint limits (lines 9-10). At the end of the execution of line 13, a sequence of nodes ($p_1 \ldots p_m$) would have been computed with associated gains of gain(1)... gain(m), m being at most |V| and bearing the sequence of the marking order of the nodes. A value of k where $1 \leq k \leq m$ is chosen such that the value of Gain = $\sum$gain(i) for i = 1 to k is maximized (line 14). This value of Gain is the increase in cost as a result of transferring nodes p_1 through p_k from their present partitions to their opposite partitions. If this Gain value is positive, then it means that the transfer operation of these k nodes is fruitful, and

so the transfer is performed, with the following steps as shown in lines 15-19: the sets X and Y are updated and the value of T is updated to T + Gain.

At the end of the execution of the heuristic in its entirety, we are left with an assignment of variables (nodes) to the memory banks (partitions) X and Y.

4.3.2.1 Illustration

For the benefit graph presented in Figure 4.3, the heuristic proposed in Section 4.3.1 can be illustrated as below on the assumption that the maximum allowable size of a memory bank is four: Based on the heuristic shown in figure 4.4, an initial partitioning of variables can be assumed as: partition block X: `a f b` and partition block Y: `c d e`. Now, the initial cost incurred by the partitions is equal to 18, which is the sum of the weights of all the edges with end-points in different partitions. For every node, the initial computation of D values (using the formulae presented in Section 4.3) is in Table 4.1.

Table 4.1: Initial computations of the costs associated with the nodes of Figure 4.3.

Node	I(node)	E(node)	D(node)
a	5	10	-5
b	4	3	1
c	5	7	-2
d	3	7	-4
e	2	4	-2
f	3	5	-2

Initially, all of the nodes are unmarked. It is seen that node ‘b’ has the maximum total value, and since the partition size of Y would be four after the potential transfer of

node b, which is within the bank size constraint, the node ‘b’ is marked, and then the D values of all the remaining unmarked nodes are updated as if ‘b’ had been transferred. The new D values of the nodes would be as follows.

New D(a) = D(a) – 2*w(a, b) = -5 – 2*3 = -11

New D(f) = D(f) – 2*w(f, b) = -2 -2*1 = -4

New D(d) = D(d) + 2*w(d, b) = -4 +2*1 = -2

New D(c) = D(c) + 2*w(c, b) = -2 + 2*1 = 0

New D(e) = D(e) + 2*w(e, b) = -2 + 2*1=0

And the process would be repeated until all the nodes are unmarked, and then the nodes whose transfer yields the maximum cumulative gain would be transferred and the partitions updated, and the total cost of the intermediate partitions would be updated. The heuristic continues execution until the cumulative gain assumes a non-positive value.

4.3.2 2 Results

The heuristic presented in Figure 4.5 was implemented for the graph shown in Figure 4.3, and it was found that the maximum cost incurred by a partition of the variables into two sets was equal to 19. This value of cost was achieved by multiple combinations of allocation of variables to the partition blocks, some of them being: X = {a, b, f} and Y = {c, d, e}, X = {a, f} and Y = {c, d, e, b}, and X = {c, f, b} and Y = {a, e, d}. It should be noted that the particular solution arrived at after the execution of the heuristic depends upon the initial partitions assumed, because the execution of the heuristic proceeds only until the current cost is equal to the maximum possible cost, or in other words, until any further transferring of nodes would yield a non-zero gain.

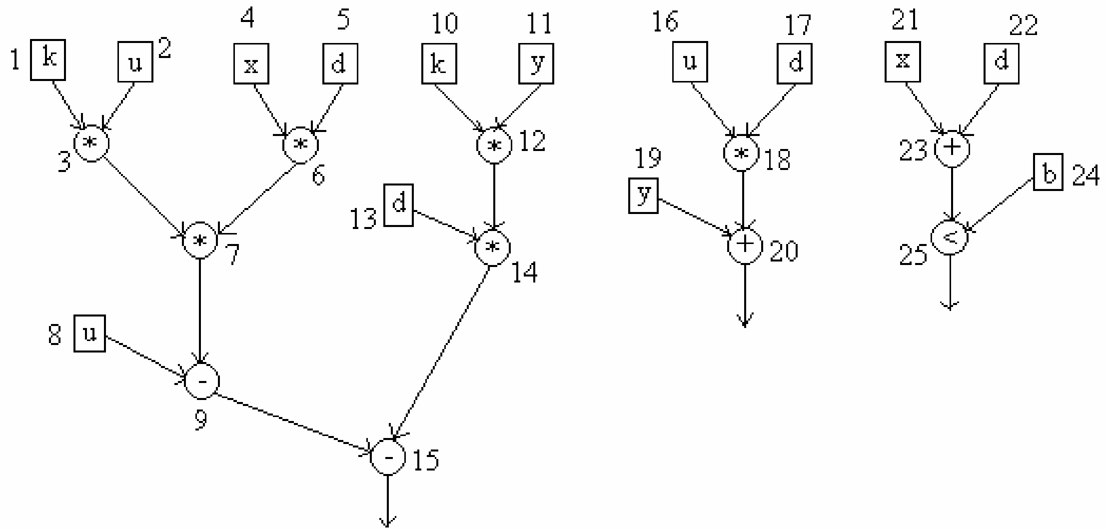

Figure 4.6: Data Dependency Graph for the differential equation problem.

In addition, we considered the data dependency graph shown in Figure 4.6, which shows node numbers. For our analysis, it is assumed that the outputs of the operators numbered 3, 6, 7, 9, 12, 14, 15, 18, 20, 23 and 25 are stored in temporary variables A, B, C, D, E, F, G, G, H, I and J respectively. From this data dependency graph, a folded interference graph is constructed which is not shown here due to space constraints. The heuristic presented in Figure 4.5 was implemented on this folded graph, and it was found that the maximum cost of partitioning was 141, and that there were many possible partitions yielding the same value, depending on the choice of the initial partitions by using the heuristic of Figure 4.4. Some example partitions of the variables are: X = {y, b, E, J, F, I, H, d} and Y = {u, B, C, D, A, G, k, x}, X = {k, C, A, u, D, G, B, x} and Y = {y, I, d, J, b, H, F, E}.

4.3.3 Complexity Analysis

An attempt at comprehending the magnitude of the involved computation complexity with the above heuristics can be made on the following lines: The cost

between the two sets X and Y requires the consideration of all the edges, and therefore, the initial computation of the cost between X and Y requires $O(|V|^2)$ time. Then, each pass consists of the following: computation of the D(u) values for all u which again requires the consideration of all the edges from u to other vertices and therefore requires $O(|V|^2)$ time, finding an unmarked node p maximizing gain(p) which requires the consideration of all the unmarked nodes and therefore O(|V|) time, marking node p which requires O(1) time and marking the nodes of the partition in which p is located in the event that the size bound is exceeded requires O(|V|) time, updating D(u) values for the unmarked nodes u which requires the consideration of at the most all the nodes and therefore O(|V|) time, picking 'k' maximizing Gain can involve the selection of all the stored nodes in the worst case which is |V| and therefore requires at most O(|V|) time, and the updating of the new sets X and Y each of which requires the consideration of all the nodes in the worst case and therefore O(|V|) time, in addition to the O(1) time associated with the updating of the value of T. If the total number of passes that are computed is P, then the total complexity of the heuristic can be estimated from the above analysis to be $O(P*((|V|^2) + (|V|) + (1)))$ time, which is similar to what is shown in [29].

4.4 Chapter Summary

A general approach to solving the problem of variable partitioning was proposed in this chapter. This approach incorporates the relaxation of the initial assumption that the variables have to be initially distributed between the two banks X and Y equally, and also implements the technique of variable transferring, rather than that of variable swapping, which supports the generality of the initial assumption made regarding the variable partition sizes, and also translates to the max-cut graph partitioning problem. Though the

time complexity of the above heuristic is similar to what the Kernighan and Lin [11] algorithm has been proven to involve by Yelick [29], the heuristic proposed above adopts a more general approach in terms of the options available to a particular node at a particular point in the execution of the heuristic, and that the memory overhead and storage requirements involved with the implementation of this heuristic are lower than that which uses the approach in Kernighan and Lin [11].

Thus, by running the initial partition heuristic in Section 4.3 and implementing the heuristic of Section 4.3.1 for particular partition imbalance constraints, we can ostensibly obtain a solution to the variable partitioning problem by maximizing the sum of the edge weights between the two sets X and Y, or in other words, the gain [8], and thus, maximizing the cost.

5. CONCLUSION AND SCOPE FOR FUTURE WORK

The new SOA heuristic proposed in Chapter 2 of this thesis introduces to the heuristic of Leupers and Marwedel [15] the extra step of updating the T-values of the edges sharing a common node with the latest edge that has been selected into the path cover set; this allows one the opportunity to explore the possibility of using a dynamic approach to potentially reduce the total access cost for an access sequence. For the more general problem of global offset assignment (GOA), the heuristic proposed in Chapter 3 of this thesis provides a meaning to the access graph formed after the selection of the end-points of the edge with the maximum weight/T-value into the set assigned to a particular address register. Further, it was also demonstrated that the heuristic, when implemented on some access sequences, yields better cost values and a smaller number of required address registers as when compared to the heuristic of Leupers and Marwedel [15].

From the results obtained in Chapters 2 and 3 for the SOA and the GOA executions, respectively, it was seen that there were no gains in average cost for the SOA access sequences, and that there were significant improvements in some of the GOA examples in terms of either the cost associated with the access sequence, or the number of resource units required, or even both. Even though there existed certain examples where there was no apparent improvement in either of the above terms, it is observed that the execution time of the heuristic proposed in this thesis is much less compared to the GOA heuristic proposed in Leupers and Marwedel [15].

The issue of the assignment of variables of a code into memory banks was researched, and it was found that the heuristic proposed in [8, 11] adopted the technique

of variable swapping, which is too rigid to explore the possibility of relaxing the memory bank balance constraint that could lead to a better solution in terms of the variable partitions. The heuristic presented in Chapter 4 of this thesis addresses this issue, and proposes a more general heuristic which begins with a more general assumption of the availability of an imbalanced variable partition, subject to a constraint of there being a maximum limit on the number of variables assigned to a particular bank. This heuristic uses the technique of variable transferal from one set to another, and was found to be similar to the heuristics of [8, 11] in complexity.

5.1 Possibilities for Further Work on SOA and GOA

The research carried out in this thesis provides a new insight to the SOA and GOA problems. This, in addition to the results obtained in the above could be used further in the consideration of the following aspects.

5.1.1 Introduction of Other Tie-Break Parameters

In the computation of the T-values of the edges in the SOA heuristics, the edge weights were used as the key parameter of the tie-breaking function; and in the determination of the candidate edge in the event of a tie between edges having the same weight, the tie-breaking function was incorporated, and a random choice was made from among the edges if even their T-values were equal. However, the use of the degrees of the nodes and the expression of the edges as their function instead of the edge weights, and thus computing the T-values using the tie break function so formed is also an avenue that could be explored. Arithmetic combinations of the above mentioned parameters are other possible variables that could be used as the baselines for the development of the tie-break function. Also, the selection of the baseline could be tailored to the nature of the

graph as defined by its density, size, number of variables, edge weight distribution, etc. Further, the differences between the T-values of the edges by the static and dynamic methods could also be used as a second-level tie-breaking criterion, both in the SOA and in the GOA cases.

5.1.2 Sequencing of Tie-Break Criteria

During the intermediate process of the selection of edges for computation of the maximum weight path cover, different parameters could be used sequentially in the event of the occurrence of edges having the same T-values, such as using the edge weights during the first round of T-value computations, and then resolving ties in the T-values by further using the sum of the degrees of the end-points of such edges between which there exists a tie. The effects of each of these parameters and the sequences in which they are invoked could be analyzed for different test scenarios and a better insight to the problem could be achieved.

5.1.3 Formulation and Heuristics for Higher-Orders

The feasibility of implementing the heuristics proposed hitherto, in addition to the heuristics proposed in this research in the scenario of higher-order post-increment/decrement operations of address registers is one area that offers tremendous potential for further research, as also relevant and necessary compiler optimization techniques [12,31] and their implementations in the lines of and in continuation of those presented in Marwedel and Goossens [20], and the improvement and usage of optimal and innovative architectural features such as reconfigurable instruction set processors [30]. Also, further attempts can be made for incorporating the use of efficient algebraic

transformations to the access graph for computing a good offset assignment, as is done in [23].

5.2 Possibilities for Further Work on the Variable Partitioning Problem

The heuristic presented in Chapter 4 of this thesis was based on the network partition problem, after extensive research was conducted on prior investigations and the present possibility of translating the NP-complete variable partitioning problem to another NP-complete problem, for which a known efficient heuristic exists. Although the heuristic proposed here provides for a more general approach, there always exists the possibility of mapping this problem into other problems, such as those which are solved by genetic algorithms [22] and [23], and these mappings could provide useful leads to the postulation of heuristics with better designs and lower complexity. Also, further avenues of optimizing costs involving access of variables within a partition on the lines of [7] could be explored.

REFERENCES AND BIBLIOGRAPHY

[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 offset assignment on embedded processors using transformations. In *Proc. High Performance Computing (HiPC 2000)*, pp. 367-374, December 2000.

[4] 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.

[5] D. Bartley. Optimization Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software Practice and Experience*, 22(2):101-110, February 1992.

[6] Y. Choi and T. Kim. Address assignment combined with scheduling in DSP code generation. In *Proc. ACM/IEEE Design Automation Conference* (DAC), pp. 225-230, New Orleans, LA, June 2002.

[7] E. Eckstein and A. Krall. Minimizing cost of local variables access for DSP-processors. In *Proc. LCTES'99 Workshop on Languages, Compilers and Tools for Embedded Systems*, Y. A. Liu and R. Wilhelm, Eds., vol. 34, no. 7, pp. 20-27, Atlanta, GA, May 1999.

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

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

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

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

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

[13] R. Leupers, A. Basu, and P. Marwedel. Optimized Array Index Computation in DSP Programs. In *Proc. Asia South Pacific Design Automation Conference* (ASP-DAC), 1998.

[14] R. Leupers and F. David. A Uniform Optimization Technique for Offset Assignment Problems. In *Proc. 11th Int. System Synthesis Symposium* (ISSS), 1998.

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

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

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

[18] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, G. Araujo, A. Sudarsanam, S. Malik, V. Zivojnovic, and H. Meyr. Code Generation and Optimization Techniques for Embedded Digital Signal Processors. In *Hardware/Software Co-Design*, Kluwer Academic Publishers (G. De Micheli and M. Sami, Editors), 1995.

[19] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235-253, May 1996.

[20] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

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

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

[23] A. Rao and S. Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs. *In Proc. ACM SIGPLAN 99 Conference on Programming Language Design and Implementation* (PLDI), pp. 128-138, Atlanta, GA, USA, May 1999.

[24] 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.

[25] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures. In *Proc. Design Automation Conference (DAC)*, 1997.

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

[27] B. Wess and M. Gotschlich. Optimal DSP Memory Layout Generation as a Quadratic Assignment Problem. In *Proc. Intl. Symp. on Circuits and Systems* (ISCAS), 1997

[28] B. Wess, S. Froehlich, and M. Gotschlich. Optimizing Data Address Computation in DSP Programs: Analysis and Evaluation. *INTHFT*, Vienna University of Technology, Vienna, Austria.

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

[30] Q. Zhao. *Static Resource Models for Code Generation of Embedded Processors.* PhD Thesis, Dept. of EE, Eindhoven University of Technology, the Netherlands, 2003.

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

VITA

Satya Swaroop Mahapatra was born in India in 1978. He completed his high school from KVIISc, Bangalore, India, in 1996 and then obtained his Bachelor of Engineering degree from the M.S. Ramaiah Institute of Technology, Bangalore, India, in 2000. After working at Wipro Technologies as a software engineer during the period 2000-2002, he arrived at Louisiana State University to begin pursuing his master's degree with a great deal of enthusiasm, zeal, and a keen desire to delve into the realm of embedded systems. He is expected to graduate with his Master of Science degree in Electrical Engineering in the spring of 2005.