2004

Address optimizations for embedded processors

Sai Pinnepalli
Louisiana State University and Agricultural and Mechanical College, sai@lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_dissertations/3504

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
ADDRESS OPTIMIZATIONS FOR EMBEDDED PROCESSORS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Sai Pinnepalli
B.E. Bangalore University, India, 1987
M.S. Louisiana State University, 1995
May 2004
Acknowledgments

This dissertation could not have been completed without significant help and input from two people. First, I would like to thank Dr. Ram (J. Ramanujam), who guided me with patience and accommodated my schedule to help me with this work. Second, I would like to thank Dr. Doris Carver, who steadfastly directed me towards this goal. I would also like to thank Dr. S. S. Iyengar, Dr. Donald E. Kraft, Dr. Jinpyo Hong, and Dr. Thomas Shaw for serving on my committee.

I would be remiss if I did not mention the amount of time Dr. Ram and Dr. Hong spent on weekends discussing my work. Dr. Ram has the ability to discuss your ideas as if every one of them merits discussion. I am grateful for these discussions, some of which are chapters in this dissertation.

Some of the work was expedited due to the use of “neato” from ATT Research Labs and sample code for heuristics provided by Dr. Hong.

Working full time while trying to pursue this degree required cooperation from my employers. Dr. Charles E. Graham and Mrs. Carol Wesson actively supported my pursuit. My appreciation for their support is heartfelt.

I would like to express my sincere gratitude to my parents, who allowed me to do my bidding, when taking care of them was my duty. I would like to express my thanks to my wife, Savitha, who handled the tasks that were due to me.

Finally, I need to thank Pratima, who allowed me to work when I should have played.
# Table of Contents

Acknowledgments ........................................................................................................ ii

List of Tables ................................................................................................................ v

List of Figures ............................................................................................................... vi

Abstract ......................................................................................................................... ix

1 Embedded Systems ................................................................................................. 1
   1.1 Classes of Embedded Systems ......................................................................... 2
   1.2 Digital Signal Processors ................................................................................. 3
   1.3 Code Optimization ............................................................................................ 7
   1.4 Dissertation Summary ....................................................................................... 8

2 Background and Related Research ......................................................................... 10
   2.1 Addressing Modes ......................................................................................... 13
   2.2 Offset Assignment ......................................................................................... 15
   2.3 Modify Register ............................................................................................. 20
   2.4 General Offset Assignment ............................................................................ 21
   2.5 Approach of this Dissertation and Experimental Work ............................... 21

3 Commutative Transformations ................................................................................ 23
   3.1 Acceptable Transformations ............................................................................ 23
   3.2 Impact of Transformations ............................................................................. 24
   3.3 Related Research in Commutative Transformation ....................................... 25
   3.4 Motivating Example ....................................................................................... 27
   3.5 Classification of Edges ................................................................................... 29
   3.6 xformSOA ...................................................................................................... 31
   3.7 Detailed Example ........................................................................................... 34
   3.8 Experimental Results ..................................................................................... 39
   3.9 Possible Variations ......................................................................................... 40
   3.10 Chapter Summary ......................................................................................... 41

4 SOA with Modify Register ..................................................................................... 43
   4.1 Modify Register .............................................................................................. 43
   4.2 Related Research in SOA with MR ................................................................. 44
   4.3 Motivating Example ....................................................................................... 47
   4.4 Edge Folding, Node Swapping, and Path Interleave ....................................... 50
   4.5 SOA2MR ......................................................................................................... 54
   4.6 Detailed Example ........................................................................................... 58
   4.7 Experimental Results ..................................................................................... 63
   4.8 Chapter Summary ........................................................................................... 63

5 General Offset Assignment (GOA) ....................................................................... 65
   5.1 Related Research ........................................................................................... 65
5.2 Motivation......................................................................................................... 67
5.3 GOA DEG........................................................................................................ 68
5.4 Detailed Example.............................................................................................. 71
5.5 Variations in Heuristic...................................................................................... 72
5.6 Experimental Results........................................................................................ 73
5.7 Chapter Summary ............................................................................................. 75

6 Offset Assignment with SSA ..................................................................................... 77
6.1 SOA with SSA-Form ........................................................................................ 78
6.2 Commutative Transformation of SSA-Form .................................................... 80
6.3 Variable Space Reuse ....................................................................................... 84
6.4 SOA with SSA .................................................................................................... 87
6.5 Chapter Summary ............................................................................................. 87

7 Incrementally Improving SOA................................................................................... 89
7.1 Deficiencies in SOA.......................................................................................... 89
7.2 Related Work .................................................................................................... 91
7.3 Limitations in Existing Incremental Algorithms .............................................. 93
7.4 Improved-Incremental-SOA ............................................................................. 94
7.5 Improved Incremental Assignment with MR and GOA ................................... 96
7.6 Node-Based SOA .............................................................................................. 97
7.7 Chapter Summary ............................................................................................. 99

8 Conclusions .............................................................................................................. 101
8.1 Future Research .............................................................................................. 104

References................................................................................................................. 106

Vita ............................................................................................................................. 113
List of Tables

Table 3.1 Results of an Implementation of xformSOA ................................................................. 40
Table 4.1 Transition Distances and their Weights ................................................................. 45
Table 4.2 Cost of Assignments Shown in Figure 4.16 ......................................................... 62
Table 5.1 Variations of GOA_DEG and their Results ......................................................... 74
Table 5.2 Comparison of GOA_DEG with other GOA Heuristics ........................................ 74
Table 8.1 Heuristics Summary .......................................................................................... 103
List of Figures

Figure 1.1 First Generation of TI’s TMS320 DSP ............................................................. 4
Figure 1.2 Second Generation of TI’s TMS 320 DSP ........................................................ 5
Figure 1.3 Harvard and von Neuman architectures ............................................................ 6
Figure 2.1 Data Path of the Motorola 56K DSP ............................................................... 11
Figure 2.2 AGU in Motorola 56K DSP ............................................................................ 12
Figure 2.3 Offset Assignment in the Order of Usage ....................................................... 16
Figure 2.4 Liao’s SOA Example....................................................................................... 17
Figure 2.5 Liao’s SOA Heuristic ...................................................................................... 18
Figure 3.1 Commutative Transformation Concepts........................................................... 25
Figure 3.2 Basic Block for Example................................................................................ 27
Figure 3.3 Access Graph for Basic Block in Figure 3.2 ................................................... 28
Figure 3.4 Basic Block for Access Graph in Figure 3.5 .................................................... 28
Figure 3.5 Access Graph for Basic Block in Figure 3.4 ................................................... 29
Figure 3.6 xformSOA Heuristic..................................................................................... 31
Figure 3.7 Procedure Measure for xformSOA Heuristic .................................................. 32
Figure 3.8 Procedure xform for xformSOA Heuristic ..................................................... 33
Figure 3.9 Basic Block from Atri et al............................................................................. 34
Figure 3.10 Access Graph for the Basic Block in Figure 3.9 .......................................... 34
Figure 3.11 Edge Classification for the Basic Block in Figure 3.9 ................................. 35
Figure 3.12 Measure of Access Graph in Figure 3.10 ...................................................... 36
Figure 3.13 Basic block after transformations to basic block in Figure 3.9 ................. 37
Figure 3.14 Access Graph for Basic Block in Figure 3.13 .............................................. 37
Figure 5.4 Access Graph and Offset Assignment for “b c e f c f b”................................. 72
Figure 5.5 Trend Graph Comparing Versions of GOA_DEG ........................................... 75
Figure 6.1 Basic Block and its SSA-Form........................................................................ 77
Figure 6.2 Branch-Join and its SSA-Form........................................................................ 77
Figure 6.3 Basic Block for SOA and its SSA-Form........................................................ 78
Figure 6.4 Access Graph and Path for cdfc1hc1abe1gb2c2ac2a1.................................. 79
Figure 6.5 Basic Block and its SSA-Form for use with xformSOA......................... 80
Figure 6.6 Access Graph of a Basic Block |S| ← 40, |V| ← 7.......................................... 81
Figure 6.7 Basic Block Commutatively Transformed...................................................... 81
Figure 6.8 Access Graph of Basic Block in SSA-Form.................................................... 82
Figure 6.9 Path Cover of the Transformed Basic Block/SSA-Form............................ 83
Figure 6.10 Lifetimes of a Variable using SSA-Form.................................................... 84
Figure 6.11 Variables Grouped for Space Reuse............................................................ 85
Figure 6.12 Access Graph with Variable Reuse............................................................. 86
Figure 6.13 Access Graph with Variable Reuse and Commutative Transformation....... 86
Figure 6.14 SOAwithSSA Heuristic................................................................................. 88
Figure 7.1 Basic Block, Access Sequence and its Access Graph................................. 90
Figure 7.2 Sub-Optimal and Optimal Cover of Access Graph..................................... 91
Figure 7.3 Incremental-Solve-SOA................................................................................ 91
Figure 7.4 Improved-Incremental-SOA......................................................................... 95
Figure 7.5 SOAbyNode Heuristic.................................................................................. 98
Figure 7.6 Transformations by SOAbyNode................................................................. 99
Abstract

Embedded processors that are common in electronic devices perform a limited set of tasks compared to general-purpose processor systems. They have limited resources which have to be efficiently used. Optimal utilization of program memory needs a reduction in code size which can be achieved by eliminating unnecessary address computations *i.e.*, generate optimal offset assignment that utilizes built-in addressing modes.

Single offset assignment (SOA) solutions, used for processors with one address register; start with the access sequence of variables to determine the optimal assignment. This research uses the basic block to commutatively transform statements to alter the access sequence. Edges in the access graphs are classified into breakable and unbreakable edges. Unbreakable edges are preferred when selecting edges for the assignment. Breakable edges are used to commutatively transform statements such that the assignment cost is reduced.

The use of a modify register in some processors allows the address to be modified by a value in MR in addition to post-increment/decrement modes. Though finding the most beneficial value of MR is a common practice, this research shows that modifying the access sequence using edge fold, node swap, and path interleave techniques for an MR value of two has significant benefit.

General offset assignment requires variables in the access sequence to be partitioned to various address registers. Use of the node degree in the access graph demonstrates greater benefit than using edge weights and frequency of variables.
The Static Single Assignment (SSA) form of the basic block introduces new variables to an access graph, making it sparser. Sparser access graphs usually have lower assignment costs. The SSA form allows reuse of variable space based on variable lifetimes.

Offset assignment solutions may be improved by incrementally assignment based on uncovered edges, providing the best cost improvement. This heuristic considers improvements due to all uncovered edges.

Optimization techniques have primarily been edge-based. Node-based SOA technique has been tested for use with commutative transformations and shown to be better than edge-based heuristics.

Heuristics developed in this research perform address optimizations for embedded processors, employing new techniques that lower address computation costs.
1 Embedded Systems

Computer systems used for everyday tasks that run a myriad of general purpose applications use general-purpose processors such as the Intel Pentium IV. These processors have the ability to perform multiple tasks. Many devices that have to perform a specific set of tasks, such as a mobile phone or microwave oven, do not need a general-purpose processor as a small subset of the abilities of a general-purpose processor will suffice.

Systems that require limited processing power may be designed as non-programmable hardware [22, 23, 30, 84]. These systems may be faster, smaller, and more efficient than general-purpose processors. However, it is not practical to design all or a majority of features in the hardware. As the complexity of the systems grows, so does the complexity of designing complete systems in hardware, which increases the design time and cost of such systems. In order to improve design-to-market time of these devices some of the functions are implemented in software, allowing the designers to make changes to a system even after hardware design is complete [40, 42]. It should be noted that hardware redesign is more difficult than software redesign. In addition to functionality, additional factors such as real-time performance, size, and power consumption, influence the decision to implement systems that are partly in software and partly in hardware.

Systems that contain such programmable components used for specific class of applications are called embedded systems [31, 41, 73, 74]. The programmable components of these systems are called embedded processors [14, 25-27, 40].
1.1 Classes of Embedded Systems

Major classification of embedded systems includes Microcontrollers, RISC Processors, DSP Processors, multimedia processors, and Application Specific Instruction-Set Processors (ASIP) [26, 27, 53, 54, 59]. These classifications are not ironclad, and the functionality of one set of processors may be found in another set.

**Microcontrollers:** These processors are typically used in control systems, and in practice they employ Complex Instruction Set Computing (CISC) architecture. CISC architectures permit high code density, but others resources available to a microcontroller are limited. The 8051 microcontroller is a popular example, manufactured by companies such as Texas Instruments, Analog Devices, Atmel, and Cygnal [80].

**RISC Processors:** Reduced Instruction Set Computers (RISC) are processors with simpler architectures [43]. RISC processors execute higher number of instructions in a given time while requiring a higher number of cycles per instruction (CPI). Simple load-store architectures such as MIPS are RISCs and have a high number of general-purpose registers. Examples of RISC are MIPS, Alpha, and Sparc, while ARM family is an example of RISC core.

**DSP Processors:** Digital Signal Processors perform computationally demanding, iteratively intensive tasks. They are sensitive to numeric errors and have real-time constraints. DSPs are designed to exploit simple memory access patterns and predictable program flow found in signal processing applications [14, 16, 26, 27, 48, 52-54]. This is achieved by dedicated hardware support and specific data path architecture. Companies manufacturing DSPs include Texas Instruments, Motorola, Analog Devices, and NEC.
Multimedia Processors: These processors are similar to DSPs with wider data paths, wider registers, and more registers than DSPs. They have more interfaces to memory such as RDRAM, SDRAM as opposed to SRAM alone found in traditional DSPs. Processors such as Philips’ Trimedia, Mediaprocessor from Microunity, and MPACT from Chromatic Research fall under this category. Multimedia processors use VLIW paradigm as in 5-way VLIW machine with 128x32 bit register file, 32KB instruction cache, 16KB data cache, and 27 functional units [59].

Application-Specific Processors: ASIPs are ASICs met half-way by general-purpose processors. These processors may be based on other embedded systems classifications such as DSP or RISC. ASIPs use application specific data paths and may require retargetable program compilation [51, 66, 67, 81, 86].

As indicated in the description, each of these categories is not distinct. Functionality and design of some of these embedded system classifications overlap. In future some general-purpose processors may be adapted for use as embedded processors as the complexity of embedded processors grows.

1.2 Digital Signal Processors

In this dissertation we concentrate on architectures similar to DSPs. We describe this architecture in greater detail. A wide variety of products in the market have some form of DSP based processor, and this number has dramatically increased in last few years [26, 27, 53, 54]. DSPs have become components of products in such areas as consumer electronics, communication devices, industrial products, and medical appliances.
Programmable DSPs are a range of microprocessors that are optimized for signal processing. A typical DSP is shown in Figure 1.1. They can be programmed and upgrades may be performed to a program in its program ROM in the field and even after the product release. The ability to handle some of the functions in software makes DSPs cheaper than custom hardware. DSPs also exhibit advantages in terms of speed and energy efficiency compared to other embedded system architectures.

![Figure 1.1 First Generation of TI's TMS320 DSP [83]](image)

DSP architectures are fairly unique as they have been modified to accommodate a DSP function or algorithm. In order to perform special tasks required by the algorithms systems might have new features such as computational units and addressing modes (Figure 1.2). Typically, general-purpose microprocessors implement multiplication by a combination of shift and add operations. These operations however require multiple clock
cycles. DSP processors such as the TMS32010 have incorporated special hardware allowing the processor to implement multiplication in a single clock cycle. For this reason modern DSPs include single cycle multipliers or combined multiply-accumulate (MAC) units. DSP processors often include several execution units that can function in parallel.

**Memory Access:** Performance of DSPs requires a high bandwidth for memory operations. These requirements are much higher than those supported in general-purpose processors. DSPs have used different architectures than their general purpose counterparts to exploit any advantages that architecture may offer. While General-purpose processors used the von Neuman architecture (Figure 1.3), DSPs tried Harvard and Super Harvard ARCHitectures.(SHARC)

![Image of Second Generation of TI’s TMS 320 DSP](image-url)
Harvard architectures (Figure 1.3) used separate buses for data and addresses. This feature refined as Super Harvard ARCHitecture (SHARC) added an instruction cache that stored frequently used instructions while simultaneously fetching two operands. High bandwidth requirements are facilitated by additional units that generate addresses in parallel with main processor functions. These units called Address Generation Units (AGUs) take advantage of predictability by supporting special addressing modes that enable the processor to access data/code more efficiently.

![Figure 1.3 Harvard and von Neuman architectures](image)

DSPs in the low cost/performance range issue and execute one instruction per clock cycle, while using complex instruction sets. Mid range DSPs use a combination of clock speed and architectural changes to obtain higher performance and speed. Features used by mid-range, DSPs achieve higher performance while keeping energy and power consumption low.
High end DSPs are enhanced conventional DSPs with wider data buses that may retrieve more data per clock cycle or use wider instructions for additional operations. DSPs are now being designed using VLIW and superscalar architectures.

1.3 Code Optimization

Despite improvements in technology of these systems, factors affecting the cost of an IC still affect performance. One main factor is the size of the IC. Semiconductor costs increase exponentially with the size of the IC, since smaller die sizes are more conducive to higher yields [47, 55]. Program code is usually the single largest factor of area in embedded systems [11, 50, 55, 57, 82]. Though reducing the size of other elements of the system is desirable, it is essential that program code be optimized to reduce the amount of ROM used by program code. When the size of an IC is a constraint, optimizing code size will allow a designer to add more features to the embedded system. This inability to reduce code size might prompt elimination of desirable features [31, 55].

Data memory, energy consumption, and power dissipation are other constraints for embedded systems [72]. Optimizing data memory is easier than optimizing program code as some memory location may be used for different data elements if their life-times are different. Reducing the size of code allow a program to execute quickly. Faster execution of code uses less energy, leading to lower power dissipation. Power dissipation may also be affected by reducing the clock cycle. Some work focuses on code generation with energy efficiency as the prime consideration. We concentrate on code optimization that will improve performance and reduce energy.
Programs written for general-purpose processors are usually written in high-level languages like C/C++. Optimizing for such architectures is easier than optimizing for embedded systems as embedded systems have irregular architectures and instruction sets depending on the application for which the system is designed. Also, the optimization techniques used in traditional compilers emphasize speed of execution and not code size [1-3, 10, 15, 21, 24, 28, 29, 46, 58, 70]. Some of the techniques may be useful and may result in code size reduction while improving speed.

Due to idiosyncrasies in architecture and instruction set, a DSP program may have to be written in assembly language. Also, optimization requirements for peculiar systems may be harder to implement. Hand-optimization has been a common practice. Some DSPs have associated C compilers and there has been much work in code generation for DSPs [4-6, 17, 81, 87-89]. It has been observed for some C compilers that on average 40% (sometimes as high as 50%) of all instructions are address computations [55, 56, 85]. Such instructions are overhead to the program and contribute to code size [44, 55]. We try to eliminate such overhead with the heuristics proposed in this dissertation.

1.4 Dissertation Summary

In this dissertation we investigate issues of optimizing code size of embedded systems such as DSPs. We present solution to issues discussed and compare our work with related research.

In Chapter 2 we describe the AGU in greater detail and describe how AGU contributes to address generation and ways of optimizing output. We describe various addressing modes and their use and limitations with DSPs. We present research on
which our work is based and also describe other related research. We review the work that describes optimization techniques similar to the ones presented in this research.

In Chapter 3 we present a heuristic that uses commutative transformations to improve code size. We present optimizations that are feasible, and we classify them into breakable and unbreakable statements. We present a heuristic that uses the classification of edges and generates optimal cost assignment.

In Chapter 4 we introduce new concepts that help achieve better offset assignment with a modify register. We introduce concepts such as edge folding, node swapping, and path interleaving that permit new assignments that are more effective.

In Chapter 5 we describe a technique to assign variables to k address registers. We describe a heuristic that uses degree of a node as a basis for tie-breaks. We demonstrate its variations and show the advantages of this approach.

In Chapter 6 we present a new approach to single offset assignment. We look at the effect of static single assignment (SSA) on offset assignment. We describe how converting a basic block to its SSA form affects its access graph and how this information may be used to reuse variable space.

Results of offset assignments can further be improved if the assignment is not optimal. We present an improved algorithm that incrementally modifies assignments to produce optimal offset assignment in Chapter 7. Traditionally offset assignment is considered from the view of an edge in access graph. We propose a node-based approach to SOA, that may be used as part of commutative transformations.

Chapter 8 summarizes the research and describes possible avenues for future research.
2 Background and Related Research

A Digital Signal Processor contains functional units, such as ALU (Arithmetic Logic Unit) that manipulates data according to instructions; a Program Control Unit (PCU) that decodes instructions and performs interrupt service routines; an AGU (Address Generation Unit) that calculates address to point to the built in memory location on the DSP and other functional units that form its data path.

The data path of a DSP processor is the path data takes where input signal is manipulated to generate programmed output. Most common computations in the data path are addition, multiplication, and multiply-accumulate operations. Elements that constitute this path to convert the input to its designed output are registers, adders, multipliers, comparators, logic operators, multiplexers, and buffers as shown in Figure 2.1.

Multiplication in a DSP is a single cycle operation. This operation is performed in a multiplier that stores results in two n-bit fixed point numbers. In order to store the results of two n-bit numbers a multiplier needs registers that are twice the length of regular (n-bit) registers. The ALU implements the basic arithmetic and logic functions of the unit such as addition, subtraction, and logical operations such as and, and or. Shifter performs scaling operations when the results of fixed-point arithmetic grow very large and results have to be passed between stages. General-purpose data registers are used as input buffers for the data bus or MAC. The registers may be used individually or in combination for long word format data manipulation. The MAC performs multiply and accumulate operations in a single clock cycle. If a multiply is initiated without
accumulate, MAC clears its accumulator before performing the operation. This operation is also atomic.

The major unit that assists in conducting the operations of a data path is the address generation unit (AGU).

Figure 2.1 Data Path of the Motorola 56K DSP [69]
The AGU is a unit dedicated to calculating addresses, which provide DSP processors the ability to generate addresses efficiently. Most DSPs have one or more such units that can generate one or more addresses per instruction cycle without using the processor data path.

Figure 2.2 AGU in Motorola 56K DSP [68]

AGUs may possess a variety of registers to assist in address computations. In Figure 2.2 the DSP contains eight address registers, eight offset registers, and eight modifier registers. AGU uses these registers and the built-in ALU to compute address. At the low end, DSPs may have just one AGU with one address register (AR). At the higher end, DSPs may have multiple address registers with modify registers associated with each AR or one or more MRs associated with each AR. In the middle we have systems with one AR and one MR, or k ARs, or k ARs with one MR and other possible combinations.
Computation of the addresses in the AR by the AGU depends on the addressing modes and the instruction set available in the DSP. In this dissertation we discuss implementation of heuristics in low end DSPs. These ideas may be extended to higher end DSPs at higher end.

2.1 Addressing Modes

DSPs may use some or all of the addressing modes [68] listed here – post-increment by 1, post-decrement by 1, post-increment by N, post-decrement by N, and no update. Address calculations are performed by the AGU using the current address in the AR and the addressing mode. For all modes calculations are performed on the current address in the AR.

No update: Instructions such as MOVE perform the operation of moving a value from register to a memory location. When such operations are performed, the AR contains the address of the memory location. If the required address is incorrect, the AGU may have to perform additional operations such LOAD, LDAR, or SBAR. Once the no update operation is complete, the contents of the address register remain unchanged. The next instruction might require or operate at the same address or address modification to a desired location may not be covered by any addressing mode. To change the address to a location that may not be achieved through any of the addressing modes, a LOAD, LBAR, or SBAR operation may have to be performed.

Post-Increment by 1: This instruction in its mnemonic representation usually contains a ‘+’ (LOAD *(AR0)+) at the end of instruction. The contents of the address register are incremented by 1 after the operation is completed.
**Post-Decrement by 1:** This instruction mode is similar to the post-increment mode. This mode is represented usually, by a ‘-’ suffixed to the instruction (LOAD *(AR0)-). The contents of the address register are decremented by 1 after the operation is complete.

In the previous three modes, the addressing mode affects only the contents of the address register. Contents of other registers such as data, index, or modify are not affected by the mode. Data is affected by the instruction.

**Post-Decrement by N:** This instruction mode is implemented in various ways. In some systems the value by which the address register has to be decremented is part of the instruction, while in some systems the value by which the change is made is in an index register (LOAD *(AR0)-4) or LOAD*(AR0)-N0). With post-decrement, the value of the address register is decremented by 4 or contents of register N0: *(N0).

**Post-Increment by N:** This instruction is quite similar to post-decrement by N. Here the contents of the address register are incremented by N.

In post-increment by N and post-decrement by N, the contents of the address register are changed by a specified value. There may be a limit imposed by DSP architecture within which the increment may be performed. If the code is so optimized that all operations are performed using these modes, no overhead due to address computations will be realized.

In this dissertation we concentrate on common addressing modes like post-increment by 1, post-decrement by 1, post-decrement by N, and post-increment by N. In the literature post-increment/decrement by N refers to a register – index or modify. To avoid confusion with the literature in the field of offset assignment, we refer to this operation as increment/decrement by modify register (MR).
2.2 Offset Assignment

After code selection is done, variables are assigned addresses in storage. This assignment of variables in storage that minimizes overhead to the AGU was formulated as an “offset assignment” problem by Liao [49, 55, 57]. Optimal placement of variables uses various addressing modes permitted in the DSP to access the next variable required. Since this placement may be made after the code is generated, the delayed storage assignment problem is referred to as offset assignment problem. Storage assignment is the final stage of code generation, irrespective of the type of embedded system concerned. The problem of finding assignment for a system with one AR is termed “Simple Offset Assignment” (SOA).

Consider the code sequence in Figure 2.3 for a system with one AR and an addressing mode that contains post-increment/decrement by 1. The assembly code associated with the code sequence assumes assignment of variables in memory in the order they are accessed. We observe that in the assembly code, post-increment is used six times, while explicit address arithmetic (LDAR, SBAR) is performed nine more times. These additional operations contribute to the code size and are considered its overhead or cost.

The goal is to minimize the overhead by minimizing the use of LDAR and SBAR instructions. Liao approached this problem as a combinatorial problem of graph covering – maximum weight path cover (MWPC). The basic block is first converted to an access sequence. A statement of the form \( x \leftarrow y + z \), where the order of accessing variables is \( y \), \( z \), and \( x \), results in the access sequence “yzx”. The access sequence of the entire basic
block is a concatenation of access sequences of individual statements. The access sequence for the basic block in Figure 2.3 is “abcdefadacdfad”

c ← a + b;
f ← d + e;
a ← a + d;
c ← d + a;
d ← d + f + a;

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AR0

LDAR AR0, &a
LOAD *(AR0)+
ADD *(AR0)+
STOR *(AR0)+
LOAD *(AR0)+
ADD *(AR0)+
STOR *(AR0)
SBAR AR0,5
LOAD *(AR0)
ADAR AR0,3
ADD *(AR0)
SBAR AR0,3
STOR *(AR0)
ADAR AR0,3
LOAD *(AR0)
SBAR AR0,3
ADD *(AR0)
ADAR AR0,2
STOR *(AR0)+
LOAD *(AR0)
ADAR AR0,2
ADD *(AR0)
SBAR AR0,5
ADD *(AR0)
ADAR AR0,3
STOR *(AR0)

Figure 2.3 Offset Assignment in the Order of Usage
An access graph $G(V,E)$ is created from this access sequence, with each variable as a node and a transition as an edge in the graph [19]. The weight of each edge is the number of transitions between variables represented by end nodes. The access graph generated from the access sequence above is shown in Figure 2.4(a). Offset assignment shown in Figure 2.3 is represented in the access graph with selected edges drawn in bold. From the graph it can be observed that the edges not covered by the assignment $<a,c>, <a,d>, <a,f>, \text{ and } <d,f>$ have edge weights 1,5,2, and 1 respectively. The sum of these weights is cost described above. An alternate offset assignment was suggested by Liao as “bcdafec”. This access graph is shown in Figure 2.4(c). The uncovered edges in this graph $<a,b>, <a,c>, <d,e>, \text{ and } <d,f>$ affect the overhead. The weight of each of these edges is 1, adding up to 4. This assignment reduces overhead/cost from 9 to 4 unless $P = NP$. It
has been shown that a polynomial time algorithm does not exist for this problem. Liao proposed a heuristic described in Figure 2.5 to solve this problem.

Solve-SOA(L)
{
    G=<V,E> ← ACCESS-GRAPH(L);
    E_sort ← sorted list of edges in E in descending order of weight;
    C=<V’,E’> : V’ ← V, E’ ← {}
    while (|E’| < |V| -1 and E_sort not empty) {
        choose e ← first edge in E_sort
        E_sort ← E_sort – {e};
        if((e does not cause a cycle in C) and
            (e does not cause any vertex v’ to have degree > 2))
            add e to E’;
        else
            discard e;
    }
    return CONSTRUCT-ASSIGNMENT(E’)
}

Figure 2.5 Liao’s SOA Heuristic [55]

In this heuristic, all edges are sorted in decreasing order of their weight. Each edge is chosen to be part of the path cover if it does cause a cycle in the path. This heuristic has some drawbacks. It does not resolve a conflict when more than one edge has the same edge weight. An edge is arbitrarily chosen when more than one edge matches selection criteria (weight).

This issue has been addressed by various heuristics [7-9, 12, 13, 18, 45, 59-65, 77, 78]. Leupers offset assignment is based on Liao’s heuristic, and it addresses the issue of multiple edges having the same edge weight. A tie-break function computes the weight of
all incident edges. These edges are then sorted in descending order of their tie-break results. The edge with the highest cumulative weight that does not cause a Hamiltonian cycle is selected. This heuristic provided a solution to resolving conflicts in Liao’s heuristic.

Hong et al. [45, 76, 77], define another heuristic that defines two tie-breaking functions called adjustment functions. The first tie breaking function is based on weights of all the adjacent edges of the end nodes. The second tie-breaking function is based on the number of adjacent edges. The edges are sorted by their edge weights in descending order of their weights. When two or more edges have a conflict due to weight, the two adjustment functions are used to prioritize edges. Hong’s results show assignments of lower costs.

Atri et al. [7-9], have shown two different ways of solving the offset assignment problem. They present a heuristic that incrementally checks for the best possible location for edges that are not part of the path cover. The edges that are not part of the assignment are sorted in the descending order of their weight. Each of these edges is found an appropriate location in the assignment. This heuristic has been found to be quite effective in Leupers’ comparison of SOA heuristics [61].

Atri et al. [7-9], Rao and Pande [78] approach the SOA problem by first performing commutative transformation. Atri et. al look for edges of weight one that may be commutatively transformed to reduce the number of edges in access graph. They propose metrics that quantify each transformation. Transformations that have benefits are considered, while transformations that increase the cost of an assignment are ignored. Rao and Pande find all possible legal combinations of a basic block and its
transformations. SOA is performed on each of these transformations. This approach is exhaustive.

2.3 Modify Register

One instruction mode allows the next address to be computed within the instruction cycle using the value in the modify register. This value is usually a value that cannot be realized by auto increment/ auto decrement modes. If we consider a system with modes of increment/decrement by 1, MR value may be any value greater than 1. The new address is computed using the current AR value and incrementing/decrementing it with the value in MR. Most current heuristics compute the MR value that has greatest cost reduction.

Hong et al. [45], add a caveat to the common technique of finding MR with most benefit. If the path cover is partitioned into two or more paths [79], all feasible offset assignments with these partitions are computed. Then the MR value with most benefit is used. Without this technique, cost of transitions between partitions is not optimized. With various combinations of the partitions, all feasible MR values are computed. In this dissertation we incorporate this technique.

All MR-based assignments try to find a value for the MR register after the offset assignment is decided. We look at the possibility of manipulating this assignment before deciding on an optimal value for MR.
2.4 General Offset Assignment

The techniques applied for solving SOA problem are not all conducive to higher end DSPs that have more than one AR (k ARs). Liao’s GOA heuristic partitions the variables into $n/2$ partitions with each partition having two variables. When an address register is assigned two variables, each variable is offset by one memory location and transitions between these variables can be covered by auto-increment and auto-decrement operations. The overhead of this heuristic is zero other than initialization cost.

This heuristic assumes an unlimited number of ARs which is not practical. Also, it may be possible to assign more than 2 variables to an address register, reducing the number of registers used to execute the code. Reducing the number of registers, reduces the initialization cost of ARs. Hong et. al, Leupers have suggested other functions that allow more variables to be assigned to an AR. Hong et. al. consider the frequency of each variable to decide on the assignment of a variable to its AR. Leupers extends the SOA heuristic to GOA.

2.5 Approach of this Dissertation and Experimental Work

In this dissertation we look at the issues relating to embedded systems, offset assignment with one AR, with one MR, with k ARs, and their variations. With each of these issues, we discuss the problem, present our research into current approaches to the problem, discuss any deficiency of current approach and discuss alternatives with examples. We then propose heuristics to solve the problem and discuss experimental results.
We have used ATT Research’s *neato* graph visualization tool [71] to dynamically generate graphs that assisted in easily visualizing the problem. The heuristics are implemented in perl [75] as we have found its ability to manipulate text helps in experimenting with basic block and access sequence. To interface with heuristics implemented by others, we used their implementations. Hong’s heuristic 2 has been used extensively to provide a benchmark SOA cost.
3 Commutative Transformations

Single Offset Assignment (SOA) is usually derived from the access sequence, which is determined by the basic block of code under concern. The access sequence has a great impact on the offset assignment derived. From the given basic block we can find more than one offset assignment. Significantly different offset assignments for a basic block can be obtained only if the basic block can be modified. We explore the possibility of finding better assignments using commutative transformations.

3.1 Acceptable Transformations

Two operands \( x \) and \( y \) are said to be commutative under an operator \( \alpha \) if they satisfy \( x \alpha y = y \alpha x \)

Some instructions have commutable operations, while others do not. An instruction \( \text{ADD}(a, b) \) is equivalent to \( “= a + b” \). In this operation, \( a \) and \( b \) are commutable since \( a + b = b + a \). Similarly \( \text{MUL}(a, b) \), which is the same as \( “= a * b” \), is commutable since \( a * b = b * a \).

Some instructions such as \( \text{SUB}(a, b) \) and \( \text{DIV}(a, b) \) are not commutable. \( \text{SUB}(a, b) \) is equal to \( “= a – b” \) and \( a - b \neq b - a \), unless \( a = b \); \( \text{DIV}(a,b) \) is equal to \( “= a / b” \) and \( a / b \neq b / a \) unless \( a = b \).

From the definition of a commutative operation, some instructions might appear to be commutative, but commutative operations in such instructions is not allowed due to the implementation of the operation. For example \( \text{MPY}(a, b, c) \) is equal to \( t = \text{MPY}(a, b) \), followed by \( \text{ADD}(t, c) \) where \( t \) is an internal variable and the result of \( \text{MUL}(a, b) \) is stored temporarily in this variable. This operation may be represented as \( “(a * b) + c” \).
This operation implies that \((a \times b) + c = c + (a \times b)\). Algebraically, this assertion is true. But, while computing the values, the ADD operation cannot be completed before the MUL operation is complete. However MUL(a,b) within MPYA is still commutable. Such operations are implemented in a MAC described in Chapter 2.

Operation such as \(\text{SUB}(a, b)\) may be considered equal to \(\text{ADD}(-b,a) \equiv \text{ADD} (\text{SIG}(b),a)\). Such an operation makes ADD non atomic, and the nature of the operation also makes ADD non-commutative as SIG has to be completed before addition is performed. Similarly \(\text{DIV}(a,b) \equiv \text{MUL}(\text{INV}(b),a)\) makes MUL non-commutative. This operation is not atomic, and MUL is dependent on the result of the INV operation.

We note that commutativity is limited to atomic operations such as ADD and MUL that do not depend on internal results. In all examples and experiments we considered one or two operands only, but the results may be extended to any commutative transformation.

### 3.2 Impact of Transformations

Consider the access sequence shown in Figure 3.1 (a) and its basic block in Figure 3.1(b). Consider a valid commutative transformation of the second statement in Figure 3.1(b). This transformation results in a new set of statements in Figure 3.1(d). Statement \(l_2 \leftarrow f_2 + s_2\) is transformed into \(l_2 \leftarrow s_2 + f_2\). This changes the access sequence from ‘\(f_1 s_1 l_1 f_2 s_2 l_2 f_3 s_3 l_3\)’ to ‘\(f_1 s_1 l_1 s_2 f_2 l_2 f_3 s_3 l_3\)’. The commutative transformation in statement 2 may be represented as change in the weights of edges \(<s_2,l_2>, <l_1,f_2>, <l_1,s_2>\), and \(<f_2,l_2>\). This change is represented in Figure 3.1(e) as increment and decrement of weights for the corresponding edges.
In some instances, the weight of an edge may go from 1 to 0, which implies that the edge being considered will not exist in the new access graph. Similarly, the weight of an edge may change from 0 to 1, which implies creation of a new edge in the access graph. We exploit this feature in our heuristic to improve the cost of the assignment.

\[
\begin{align*}
&f_1, s_1, l_1, f_2, s_2, l_2, f_3, s_3, l_3 \\
&l_1 \leftarrow f_1 + s_1 \\
&l_2 \leftarrow f_2 + s_2 \\
&l_3 \leftarrow f_3 + s_3 \\
&(a) \\
&l_1 \leftarrow f_1 + s_1 \\
&l_2 \leftarrow s_2 + f_2 \\
&l_3 \leftarrow f_3 + s_3 \\
&(c)
\end{align*}
\]

(1) \(w(s_2, l_2)\) —
(2) \(w(l_1, f_2)\) —
(3) \(w(l_1, s_2)\) ++
(4) \(w(f_2, l_2)\) ++

Figure 3.1 Commutative Transformation Concepts

3.3 Related Research in Commutative Transformation

Rao and Pande [78], and Atri et al. [7-9] have tried to improve on the results of offset assignment provided by Liao-like SOA heuristics using commutative transformations. Rao’s heuristic computes optimal solution for a given basic block while Atri’s heuristic computes a hitherto better solution using some metrics.

Rao and Pande compute all possible access sequences; then the offset assignment of each of these sequences is computed. Since all possible legal sequences are tested, the best possible assignment for a given SOA heuristic is obtained. The fidelity of the results depends on the heuristic used to obtain assignment and the type of tie-break functions
used to determine priority of one edge over another. Despite obvious advantages of the heuristic that result in an optimal solution, it has a drawback. It may be acceptable to try all possible sequences for access sequences of short length, but long sequences with large number of variables and statements will make the heuristic exhaustive. One may argue that for embedded processors the developer is not as concerned about compilation speed as a developer with general-purpose processor. But, a prohibitively exhaustive compilation of every possibly trivial access sequence makes this heuristic impractical.

Atri et al define metrics to help decide on the transformations that can be made and ordered so that transformations with benefits may be applied to the access sequence.

Primary Benefit = \( \Sigma (\text{non-zero edges} \rightarrow \text{zero}) - \Sigma (\text{zero edges} \rightarrow \text{non-zero}) \)

Secondary Benefit = \( (\text{edges whose weight} \uparrow) + (\text{self edges whose weight} \uparrow) \)

The primary and secondary benefits are computed for only the edges of weight 1 in the access graph. These edges are sorted in descending order of their primary benefit, with secondary benefit being a tie-break function if two or more edges have the same primary benefit. Each of these edges is then transformed and the validity of the transformation is checked. Every transformation that generates a compatible set of edges is accepted. Transformations that might affect already included edges are ignored.

This heuristics has some shortcomings. As shown in the motivating example (Figures 3.3 and 3.5), the transformations of edges other than those with weight 1 can provide benefit. These edges may be of weight exceeding one. In some instances an edge of weight 3 might be transformed in each of the three instances and the weight of such an edge is reduced to zero. We look at such a possibility. There are heuristics today that have better tie-breaking mechanisms to improve the offset assignment than Liao’s
heuristics. Atri’s heuristic might benefit from one such heuristic. In our heuristic, we use Hong et al.’s SOA heuristic to compute the SOA cost of our transformations.

3.4 Motivating Example

Consider the basic block in Figure 3.2 and access graph in Figure 3.3.

\[
\begin{align*}
e &\leftarrow d \\
a &\leftarrow f + e \\
f &\leftarrow d \\
a &\leftarrow d + e \\
d &\leftarrow e + b \\
f &\leftarrow b \\
f &\leftarrow c + a \\
e &\leftarrow d
\end{align*}
\]

Figure 3.2 Basic Block for Example

This basic block results in a cost of 8 using Hong et al.’s heuristic. For this example we consider a different assignment with a cost of 10. We have primarily chosen edges <d, f> and <d, e> while ignoring the edge <a, e> all of which have a weight 3. All three of the transitions between a and e may be transformed using commutative transformations. \(a \leftarrow f + e\) may be transformed to \(a \leftarrow e + f\). This reduces the weight of edge <a, e>. A similar transformation may be made to the statement \(a \leftarrow d + e\). This transformation will further reduce the weight of edge <a, e>. In addition a transformation of the statement \(d \leftarrow e + b\) to \(d \leftarrow b + e\) can reduce the weight of the edge <a, e> to 0.

Instead we perform two of these transformations that affect the edge <a, e>. The resulting basic block and its access sequence are shown in the Figure 3.4 and Figure 3.5, respectively.
Figure 3.3 Access Graph for Basic Block in Figure 3.2

\[
\begin{align*}
    e & \leftarrow d \\
    a & \leftarrow e + f \\
    f & \leftarrow d \\
    a & \leftarrow d + e \\
    d & \leftarrow b + e \\
    f & \leftarrow b \\
    f & \leftarrow c + a \\
    e & \leftarrow d
\end{align*}
\]

Figure 3.4 Basic Block for Access Graph in Figure 3.5
This example suggests that a mechanism for classifying edges of the access graph may be beneficial in commutatively transforming the graph and therefore the offset assignment associated with it.

### 3.5 Classification of Edges

We identify the edges that can be transformed and edges that cannot be transformed. Edges that can be commutatively transformed are defined as “breakable” edges, while edges that cannot be commutatively transformed are defined as “unbreakable”.

Statement “C ← A + B” generates an access sequence ABC. The statement may also be commutatively transformed as “C ← B + A”, which generates an access sequence BAC. Edge AB still exists in the transformed access sequence as BA. This edge is
“unbreakable”. While edge BC can be eliminated in the second access sequence, such edges are “breakable”.

The following cases define other “breakable” (BR) and “unbreakable” (UB) edges. If both operands in the following statement are same, the edge between the lhs (left hand side) of the current statement and node in the rhs (right hand side) of the next statement is “unbreakable”. If the two operands are different, the edge is breakable.

S1: \[ C \leftarrow \]
S2: \[ Z \leftarrow X + X \]
The edge CX is “unbreakable” and edge ZX is “unbreakable” in the above code segment.

S1: \[ C \leftarrow \]
S2: \[ Z \leftarrow X + Y \]
The edge CX is classified as “breakable”, as is “ZY”. The newly added edges after commutative transformation will be CY, and ZX.

S1: \[ C \leftarrow \]
S2: \[ Z \leftarrow X \]
Edges CX and ZX are “unbreakable”.

S1: \[ C \leftarrow \]
S2: \[ C \leftarrow A + B \]
Edges CA and CB are “unbreakable”. It may be argued that this situation is a case for dead-code elimination, where statement S1 may be deleted. Statements in this form are not a complete representation of a task of the embedded system. Since the program code is executed in real time, it is possible to have a synchronous effect due a change in the value set at an address, \textit{i.e.} values set in C have an effect on the operation of the device.
We group each edge into groups of breakable and unbreakable edges. When we choose the edges, our heuristic prefers unbreakable edges over breakable edges.

3.6 xformSOA

We propose a heuristic that uses classification of the edges in an access graph to derive an empirically optimal offset assignment. The principal idea of this heuristic is presented in the Figure 3.6. We find an initial assignment that gives a cost $C$ to compare the effects of the transformation on the initial layout of the access graph ($G$). In practice, we used a standard SOA heuristic to compare the final result of all of the transformations.

```c
0:  flag ← 1
1:  C ← measure( layout(G) )
2:  while ( flag == 1 ) {
3:      G' ← xform(G)
4:      C' ← measure( layout(G') )
5:      if (C' ≤ C) {
6:         G ← G'
7:         flag ← 1 }
8:    else
9:        flag ← 0
10: }
11: optimalSOA(G)
```

Figure 3.6 xformSOA Heuristic

This heuristic iterates (line 2) until there are no transformations that reduce the cost of the previous iterations (line 5). After each set of transformations that affect the access graph, we find the cost of the new assignment $C'$ (line 4). If $C'$ is less than or equal to $C$, the cost estimated in the earlier iteration, the transformed graph $G'$ is assigned as access graph $G$ (line 6).
This heuristic has 2 procedures: \textit{xform} and \textit{measure}. Procedure \textit{measure}, shown in Figure 3.7, finds an offset assignment for a graph using the classification of edges such as breakable and unbreakable. An edge whose weight is 3 might have any combination of breakable and unbreakable edges (3BR+0UB, 2BR+1UB, 1BR+2UB, and 0BR+3UB, where BR is a breakable edge and UB is unbreakable edge). Procedure \textit{xform}, Figure 3.8, transforms a given access graph (G) and its basic block to obtain a different access sequence.

1. measure(G, B) \{ // G is access graph, B is the basic block defining G
2. \hspace{1em} BR\text{sort} \leftarrow \text{sorted list of breakable edges (B)}
3. \hspace{1em} UB\text{sort} \leftarrow \text{sorted list of unbreakable edges(B)}
4. \hspace{1em} P \leftarrow \text{MWPC with UB\text{sort} // this is essentially SOA with UB}
5. \hspace{1em} \text{add edges from BR\text{sort} not yet covered // add additional edges to path cover}
6. \hspace{1em} C \leftarrow \text{wt of uncovered edges // cost of uncovered edges}
7. \hspace{1em} \text{return(C)}
8. \}

Figure 3.7 Procedure Measure for xformSOA Heuristic

Procedure \textit{measure} classifies edges in G into two categories – breakable and unbreakable. In steps 2 and 3 of the procedure, a sorted list of breakable edges (BR\text{sort}) and another of unbreakable edges (UB\text{sort}) are created. If two edges have the same weight, in UB\text{sort}, then the edge with higher total edge weight is given higher priority. If two edges in BR\text{sort} have the same edge weight, then the current implementation considers the edge with higher weight to have a higher priority. Other variation of this constraint is also considered.

From the sorted list of unbreakable edges (UB\text{sort}), a Maximum Weight Path Cover (MWPC) is generated. The process of generating this path is akin to the generation
of offset assignment in other heuristics. Edges from BR_{sort} are then considered for addition to the path. Any edges that are not part of the path are now considered to contribute to the cost of the offset assignment \(C\). \(C\) is the return value of this procedure.

After the path cover is obtained, edges may be left in BR_{sort}, that can be transformed. These edges are then transformed and a measure of comparison for each of these transformations is computed as shown in step 5 in Figure 3.8. A transformation is accepted only if \(\Delta_{\text{eff}}\) is non-negative.

```plaintext
0: xform(G) {
1:    G' ← G
2:    for each edge e that is uncovered {
3:        for each breakable instance j of edge e {
4:            G'' ← xform of edge instance (G')
5:            \(\Delta_{\text{eff}}\) ← ( # of covered / self edges whose weight ↑ + # of uncovered edges whose weight ↓ - # of uncovered edges whose weight ↑ )
6:            if \(\Delta_{\text{eff}}\) ≥ 0
7:                G' ← G''
8:        }
9:    }
10: }
```

Figure 3.8 Procedure xform for xformSOA Heuristic

A measure of this transformed access graph \((G'')\) is obtain using procedure \textit{measure}(G''). If this cost is lower than the cost computed in the earlier iteration, the cycle of procedure \textit{xform} and procedure \textit{measure} are repeated. Once \textit{xformSOA} stops improving the cost of the access graph, the cost of the access graph is computed using
any benchmark heuristics, labeled optimalSOA, as these heuristics obtain assignments using only the edge weights and not their classifications.

### 3.7 Detailed Example

Consider the following basic block used in Atri et al. This basic block shown in Figure 3.9 yields the access sequence shown in the Figure 3.10. The edges of this access sequence are classified into breakable and unbreakable edges as shown in the Figure 3.11. The basic block is converted into access sequence ‘a b c d e f b a a e f d c b a f’ by xformSOA heuristic.

\[
\begin{align*}
    c &\leftarrow a + b \\
    f &\leftarrow d + e \\
    a &\leftarrow b + a \\
    d &\leftarrow e + f \\
    b &\leftarrow c \\
    f &\leftarrow a
\end{align*}
\]

Figure 3.9 Basic Block from Atri et al. [7].

Figure 3.10 Access Graph for the Basic Block in Figure 3.9
In procedure *measure* for the SOA, the edges are first computed as shown in Figure 3.11 (a). These edges are classified into $\text{UB}_{\text{sort}}$ (unbreakable) and $\text{BR}_{\text{sort}}$ (breakable) edges. For example, there are two instances of edge $<c,d>$, one between statements “$c \leftarrow a + b$” and “$f \leftarrow d + e$” and second between statements “$d \leftarrow e + f$” and “$b \leftarrow c$”. The first edge can be eliminated by commuting the statement “$f \leftarrow d + e$” into “$f \leftarrow e + d$”. However, the second edge cannot be commuted. Hence edge $<c, d>$, whose weight is two, is classified both in breakable edges and unbreakable edges. *i.e.*, edge $<c, d>$ cannot be completely eliminated. It can at most be reduced to an edge of weight 1.

Using the classification, *measure* derives an assignment as highlighted in the access graph shown in Figure 3.12. The *xform* procedure then commutes edges in $\text{BR}_{\text{sort}}$ that do not negatively affect the cost of the assignment. From the graph, edges $<a, e>$, $<d, f>$, $<a, f>$, and $<b, f>$ are the edges not included in the cover. It is desirable that these edges be commuted so that the cost of edges not covered by the MWPC is reduced, if not fully eliminated. Of the four edges, $<a, e>$, $<d, f>$, and $<b, f>$ are classified as breakable.

<table>
<thead>
<tr>
<th>Edge weights</th>
<th>Unbreakable Edges</th>
<th>Breakable edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>b-f: 1</td>
<td>a-b: 3</td>
<td>c-d: 1</td>
</tr>
<tr>
<td>a-b: 3</td>
<td>c-d: 1</td>
<td>e-f: 1</td>
</tr>
<tr>
<td>d-e: 1</td>
<td>b-c: 1</td>
<td>b-c: 1</td>
</tr>
<tr>
<td>a-e: 1</td>
<td>e-f: 1</td>
<td>b-f: 1</td>
</tr>
<tr>
<td>a-f: 1</td>
<td>d-e: 1</td>
<td>d-f: 1</td>
</tr>
<tr>
<td>c-d: 2</td>
<td>a-f: 1</td>
<td>a-e: 1</td>
</tr>
<tr>
<td>b-c: 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e-f: 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d-f: 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.11 Edge Classification for the Basic Block in Figure 3.9
edges. Breaking the edge \( <a, e> \) requires commuting \( "d \leftarrow e + f" \) to \( "d \leftarrow f + e" \) resulting in the elimination of edge \( <d, f> \). The result of this transaction is \( w<a, e>--, w<d, f>--, \)
\( w<a, f>++, \) and \( w<d, e>++ \). The net result is the elimination of two breakable edges not part of the path cover (\( <a, e> \) and \( <d, f> \)), and increase in the weight of an edge that is not part of the path cover (\( <a, f> \)) and one that is part of the path cover (\( <d, e> \)). This commutative transformation affects the net weight by \( "-1" \). The edge \( <b, f> \) may also be broken by transforming \( "a \leftarrow b + a" \) into \( "a \leftarrow a + b" \). This transformation does not affect the cost but changes the access graph. After the first iteration of transformations, the basic block with transformations that amount to \( "-1" \) is shown in the Figure 3.13.

Figure 3.12 Measure of Access Graph in Figure 3.10.
c = a + b
f = d + e
a = a + b
d = f + e
b = c
f = a

Figure 3.13 Basic block after transformations to basic block in Figure 3.9

The access graph and *measure is* computed for the new basic block. The access graph after the transformations is shown in Figure 3.14.

Figure 3.14 Access Graph for Basic Block in Figure 3.13.
It is evident from the access graph that the cost of the new assignment is 2. It is possible to further reduce the cost if \( <d, e> \) is breakable and its transformation only decreases the cost. This assignment is feasible only by reversing earlier transformation of “\(d ← e + f\)” to “\(d ← f + e\)”. We stop the transformations here with an optimal cost of two. This access sequence is then presented to a benchmark SOA. The SOA algorithm used in our heuristic is Hong et al.’s SOA algorithm. The cost returned for this assignment is also 2. We consider these transformations empirically optimal.

The other basic block (Figure 3.14) in the motivating example also commutes to an optimal solution in four stages as shown in the Figure 3.16. The final transformation resulting from the \(xformSOA\) heuristic, shown in Figure 3.15, yields an optimal cost of 2.

\[
\begin{align*}
c &= a + b; \\
f &= d + e; \\
c &= d + a; \\
a &= a + d; \\
d &= a; \\
b &= f;
\end{align*}
\]

Figure 3.15 Basic block from Atri’s Motivating Example

\[
\begin{align*}
c &= b + a \\
f &= e + d \\
a &= d + a \\
c &= d + a \\
d &= a \\
b &= f
\end{align*}
\]

Figure 3.16 Transformed Basic Block of Figure 3.14
3.8 Experimental Results

The $xformSOA$ heuristic was tested with random sequences of varying lengths $|S|$ and number of variables $|V|$. It is assumed that 80% of the statements are of the form $x \leftarrow y + z$ (two operands in the rhs), and 20% of the statements are of the form $x \leftarrow y$ (one
operand in rhs). Each test was repeated 1000 times before generalizing the result. The results of these tests are tabulated in Table 3.1.

The benefit is compared in an SOA heuristic not part of $xformSOA$. We use Hong’s SOA heuristic to check initial and final costs. We observe that at least 60% of the time there could be benefits in commutatively transforming statements. The benefits are seen in up to 90% of access sequences. The change in cost is as high as 12 in some instances.

This implementation was tested on motivating examples used in other research. The $xformSOA$ heuristic produced the optimal value.

| $|S|$ | $|V|$ | % affected | Max Wt. Change |
|---|---|---|---|---|
| 25 | 6 | 63.8 | 5 |
| 50 | 9 | 81.2 | 10 |
| 50 | 20 | 88.9 | 10 |
| 100 | 20 | 88.2 | 12 |
| 100 | 60 | 78.4 | 7 |
| 100 | 80 | 76.6 | 6 |
| 1000 | 300 | 90.4 | 10 |

### 3.9 Possible Variations

In step 9 in Figure 3.8, $\Delta_{\text{eff}}$ is verified to be non-negative. It is possible to consider $\Delta_{\text{eff}} > 0$. If we assume $\Delta_{\text{eff}} > 0$, we will not consider effects of subsequent transformations that show zero cost change locally, though this change might have an effect on total assignment. Additionally, a local change might adversely affect the cost with respect to complete assignment.
A variation of \textit{xformSOA} is to compute all local changes and their impact globally, while computing SOA with each transformation. We can perform incremental changes of the best possible transformation and iterate until no further beneficial commutation is feasible. This feature is similar to incremental SOA discussed later in Chapter 7.

\subsection*{3.10 Chapter Summary}

Commutative transformations affect the access sequence used for generating an offset assignment. An access sequence has a great impact on the offset assignment for a basic block; we explore effects of commutative transformations and propose a heuristic to effectively perform commutative transformations.

We introduced the concept of breakable and unbreakable edges in a basic block. Breakable edges are those edges, the operands of which can be commuted affecting the access sequence, while unbreakable edges are those that do not affect the access sequence or no commutative transformation is feasible. We find an offset assignment based first on the unbreakable edges and then if possible on breakable edges. Once a maximal weight path cover is derived, edges in the breakable list that are not part of the cover are transformed. Only transformations that have a positive effect locally are considered. If this change has a positive effect globally (improvement in cost), the \textit{xformSOA} is repeated with the new basic block.

A cost is computed with a benchmark SOA algorithm at the start and the end of this heuristic, as they do not classify each edge but use the weight of each edge to optimize. We found that 60-90\% of the time, basic blocks may be transformed. This
affects the cost of the assignment significantly. Variations of this technique are still being researched to obtain a better offset assignment.
4 SOA with Modify Register

In the Chapter 3 discussion of SOA, we assumed an embedded processor architecture that has one address register (AR). The architecture also had limited addressing capabilities such as auto increment and auto decrement. Some of these systems had additional addressing modes that allowed an increment or decrement of the address in AR by N. These architectures are quite common in older systems and systems that have been designed with limited chip size of limited functionality requirements. One additional mode of addressing can be achieved by using a Modify Register (MR). This register allows an address to be incremented or decremented by a value in the MR.

4.1 Modify Register

An enhancement to the above architecture is a system with another register in its AGU, the Modify Register. The addressing modes involving the modify register allows for the completion of the instruction and address change by the AGU in one clock cycle. Without this mode two clock cycles might be required, one for completion of the instruction and another for the change of address with ADAR or SBAR instructions.

Figure 4.1 shows the block diagram of an AGU with both AR and MR. In this example, the effective address is a combination of values to which the AR points to and the value of MR. This is the case where the system has m ARs and n MRs. If we consider a system with one AR and one MR, the effective address is AR±MR. In addition to generating an effective address, the value in AR is changed to that of the effective address. The effective address is not AR value indexed by MR value as in indexed by N
addressing mode that does not change contents of AR. Also, Indexed by N requires more than 1 clock cycle.

Figure 4.1 AGU with AR and MR [59]

4.2 Related Research in SOA with MR

Current research involves two approaches. One approach involves setting a predefined value in the address register and the other involves changing the value of MR as the need arises. The first approach is simple to address and implement. Once an offset assignment is computed by an SOA heuristic, a value for the MR is computed in the post processing phase.

Consider the edges that are not covered by the offset assignment, i.e., not covered by auto-increment or auto-decrement in AR. These transitions will now need either an LDAR or an SBAR instruction. In Figure 2.3 there are 9 instances of LDAR and SBAR.
Consider these transitions as a set of transition distances for the given assignment. $T_{\text{dist}}$ for Figure 2.3 is $\{5, 3, 3, 3, 2, 2, 5, 3\}$. From its access graph in Figure 2.4, the edges not covered by the assignment - $<a, e>$, $<a, d>$, $<a, f>$ and $<d, f>$ need these transitions. The transition between $a$ and $e$ needs an AR change of two, between $a$ and $f$ an AR change of five, between $a$ and $d$ an AR change of three, and between $d$ and $f$ an AR change of two. The weight of these edges specifies the number of transitions needed between the nodes. This information is listed in Table 4.1.

Table 4.1 Transition Distances and their Weights

<table>
<thead>
<tr>
<th>Edge</th>
<th>AR change</th>
<th># of transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;a, d&gt;$</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>$&lt;a, e&gt;$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$&lt;a, f&gt;$</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>$&lt;d, f&gt;$</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

From $T_{\text{dist}}$, it is clear that there are five instances of AR change by three, two instances of AR change by five and two instances of AR change by two. An MR value of three reduces the SOA cost by five, whereas an MR value of two or five changes SOA cost by only 2. The maximum benefit to SOA is accrued by selecting the value that has the most transitions among uncovered edges in the access graph.

This fundamental property is expanded upon by Hong [45]. Nodes in an offset assignment need not be part of one partition. In some instances there may be more than one partition of nodes that form an offset assignment. Various partitions may be combined in $2^{n-1}n!$ ways. Each partition may have uncovered edges; these edges are termed as *intra edges*. The edges between different partitions are called *inter edges*. Various combinations do not change intra edges, but the $T_{\text{dist}}$ of these transitions may
differ in each combination. Hong finds a combination of *intra* and *inter edges* that minimize the cost of the assignment.

![Figure 4.2 Access Graph with Intra Edges of $T_{\text{dist}} = 3$](image)

The access graph shown in the Figure 4.3 has one path cover, *i.e.* no disjoint path covers. The assignment has 3 *intra edges*. This offset assignment will benefit from an MR value of 3. With an MR value of 3 the cost of the assignment will be reduced from 3 to 0.

![Figure 4.3 Representation of Access Graph as Assignment](image)

In Hong’s heuristic, the value of MR remains constant during the execution of the code, whereas Leupers considers changing the value of MR as the assignment needs it. If MR is initialized to 5 with the given $T_{\text{dist}}$, is it beneficial to change the value to 3 which is
the next MR value required, as 5 is used again later. If 5 were not used again changing it
to 3 will not have any effect. With the given \( T_{\text{dist}} \), changing MR to 3 has benefits. This
evaluation is done each time there is a need for a different value from the transition
distances list.

We use the following terminology from Hong.

**Uncovered edge:** An edge \( e = <x, y> \) is called an uncovered edge when variables that
correspond to the vertices are not assigned to adjacent positions in the memory.

**Intra-edge:** An edge \( e = <x, y> \) is called an intra-edge if \( <x, y> \) is uncovered and the two
variables/vertices belong to a path cover.

**Inter-edge:** An edge \( e = <x, y> \) is called an inter-edge if \( <x, y> \) is uncovered and not an
intra-edge

### 4.3 Motivating Example

We use the motivating example used by Hong shown in Figure 4.4. Figure 4.4 is
an access graph and its offset assignment. From the access graph we find that the offset
assignment consists of two partitions “f b a c” and “d e”. The assignment of “f b a c” has
an *intra edge* \( <b, c> \) and there are two *inter edges* between the two partitions. Hong’s
heuristic computes all possible combinations of partitions and finds an assignment with
lowest cost. The assignment found is shown in Figure 4.5.

This assignment has three transitions \( <b, c> \) with distance 2, \( <a, d> \) with distance
2, and \( <a, e> \) with distance 3. With two transitions having distance 2 and 1 transition with
distance 3, an MR value of 2 is chosen resulting in an assignment with a cost of 1. The
cost of this assignment can further be reduced with additional changes.
With an MR value of 2, the assignment “f b a c” can be modified to “f b c a”. This change affects the transitions in edges <a, b> and <b, c>. The transition distance between <a, b> changes from 1 to 2. The transition that was earlier accomplished with auto-increment or decrement now has to be accomplished with MR = 2. The transition <b, c> that was earlier covered with an MR can now be implemented with an auto-increment or decrement.
The main benefit of this change in offset assignment is achieved in inter edges \(<a, d>\) and \(<a, e>\). The new assignment is shown in Figure 4.6.

![Figure 4.6 Modified Offset Assignment for Assignment in Figure 4.5](image)

The edge \(<a, d>\) which earlier could have been implemented using MR = 2 can now be achieved with an auto increment/decrement. The primary benefit is now realized with the transition in edge \(<a, e>\) which was not covered in earlier SOA with MR. The new assignment “f b c a d e” can be covered with an MR = 2 without any additional costs.

We further look at another example shown in the Figure 4.7.

![Figure 4.7 Offset Assignment with Inter and Intra edges](image)

This assignment has 3 \(\text{intra edges} \ <a, c>, \ <b, d> \text{ with } T_{\text{dist}} \text{ of } 2, \ \text{and } <d, i> \text{ with } T_{\text{dist}} \text{ of } 3\). In addition, there are 2 \(\text{inter edges} \) with the given assignment; \(<a, h> \text{ with } T_{\text{dist}} \text{ of } 7 \text{ and } <e, f> \text{ with } T_{\text{dist}} \text{ of } 2\). From this set it is apparent that if there is 1 transition
on each edge MR = 2 is appropriate choice, with two transitions uncovered by MR or auto-increment or decrement.

We consider some changes to the layout that may affect the offset assignment, consequently changing the cost. The new assignment shown in the Figure 4.8 leaves 1 transition uncovered. Changes made in this assignment will further be discussed in section 4.4

![Figure 4.8 New Offset Assignment for the OA Shown in the Figure 4.7.](image)

4.4 Edge Folding, Node Swapping, and Path Interleave

We introduce three changes that can be made to an assignment using MR = 2: edge folding, node swapping, and path interleaving

**Edge Folding** is defined as folding an edge around a node, such that the node now forms an end of the path and all nodes within the path are no more than two nodes away.

All nodes part of the fold and node adjoining the fold cannot be part of any other transformation. As detailed in Figure 4.9, the offset assignment may be changed in two different ways. The assignment in Figure 4.9(a) may be changed into the assignment shown in 4.9(d) using the transformation in 4.9(b) or it may be changed into assignment shown in 4.9(e) using transformation in 4.9(c).

In Figure 4.9(d) all nodes cannot be part of any transformation, while node a may be part of any other transformation as it is not adjoining the fold in Figure 4.9(e).
Figure 4.9 Example of Edge Folding
**Node Swapping** is defined as switching of two nodes in an assignment such that the nodes in concern and the two adjoining nodes are no more than two nodes away, *i.e.* these nodes may be accessible with MR = 2.

Once a set of nodes are swapped, the four nodes involved cannot be part of any further transformations without additional cost. Figure 4.10 shows an example of node swapping. Nodes d and e are swapped in Figure 4.10(a). Once the swap is made, the set of nodes (c, d, e, f) cannot further be transformed. In the assignment shown in 4.10 (b), a transition between c and d may be completed with an MR = 2. Similarly, a transition between e and f is achieved without any cost with an MR = 2.

Nodes of the assignment that are not part of the swap-zone can be part of any other transformation such as node swapping, edge folding, or path interleaving provided the swap-zone is not part of any further transformation.

![Figure 4.10 Example of Node Swapping](image-url)
**Path interleaving** is defined as the merging of 2 disjoint paths in path cover, such that no two edges in the original assignment are more than 2 nodes away.

This transformation can be applied to disjoint partitions, *ie inter edges* are not covered. Also, if the number of *inter edges* and its weights is limited, path interleaving may not be beneficial. If there is one *inter edge* that connects to another partition with a weight of 1, then MR will not result in any additional benefit.

![Diagram of path interleaving](image)

**Figure 4.11 Path Interleaving of Two Partitions**
Figure 4.11 shows path interleaving. “a b c i d h” and “e g h” are two partitions in the assignment with two inter edges <c, f> and <d, e>. These two partitions may be combined as “a b c i d h e g h” or “e g h a b c i d h” with MR = 2 or 3 respectively. There are two other possible combinations with higher MR values are also possible. Path interleaving shown in 4.11(b) results in offset assignment shown in Figure 4.11(c). All the variables that were accessible with an auto increment/decrement can now be accessed with MR = 2 if the assignment is modified.

4.5 SOA2MR

SOA2MR, described in Figure 4.12 is an offset assignment heuristic that uses edge folding, node swapping, and path interleaving to find a better assignment with MR = 2. In this heuristic we use a benchmark SOA with MR heuristic, the cost of which has to be bettered. A benchmark algorithm is used to calculate an MR value and the resultant cost of the offset assignment with MR.

Each path cover (PC) and its set of inter edges are first transformed using the EdgeFold procedure, shown in Figure 4.13. For each pair of nodes belonging to an inter edge, EdgeFold converts the path cover into two new paths, with the nodes separated by more than two are placed adjacent to each other. In the resulting PC, the two paths will be of the form “…v1 v2…” or “…v2 v1...”. The nodes outside the candidate nodes may be part of other transformations, whereas the nodes between the candidate nodes cannot be used in any further transformation.
**SOA2MR**(PC, UC) { // PC ← Path Cover from another SOA algorithm
    // UC ← uncovered edges
    OA ← Offset assignment from a benchmark algorithm
    C ← sum of uncovered edge weights //cost
    MR ← value set by MR algorithm
    PC\textsubscript{fold} ← \emptyset
    \textbf{foreach} PC\textsubscript{ind} in (PC){
        PC\textsubscript{fold} ← PC\textsubscript{fold} \cup \text{EdgeFold}(PC\textsubscript{ind})
    }
    PC\textsubscript{fold} ← PC\textsubscript{fold} \cup PC
    \textbf{foreach} PC\textsubscript{ind} in (PC\textsubscript{fold}){
        PC\textsubscript{fold} ← PC\textsubscript{fold} \cup \text{NodeSwap}(PC\textsubscript{ind})
    }
    (newOA, newCost) ← \text{PathInterleave}(PC\textsubscript{fold}, \text{inter edges})
    \textbf{if} (newCost < C){
        C ← new Cost
        MR ← 2
        OA ← newOA
    }
}

Figure 4.12 SOA with MR heuristic
**Procedure** `EdgeFold (PC)` { // PC ← path cover to be folded

    pathCovers ← \( \phi \)

    **foreach** \((v_1, v_2)\) in \((\text{intra edges})\) { 

        \(i_1 \leftarrow \text{index of } v_1\)
        \(i_2 \leftarrow \text{index of } v_2\)
        \(PC_{\text{pre}} \leftarrow \text{PC till } i_1\)
        \(PC_{\text{post}} \leftarrow \text{PC till } i_2\)
        \(PC_{\text{main}} \leftarrow \text{PC from } i_1 \text{ to } i_2\)
        path1 ← \(\phi\)
        
        **while** \(PC_{\text{main}} \neq \phi\) { // shift removes from left, pop removes from right
            path 1 = path1 \* pop(\(PC_{\text{main}}\)) \* shift(\(PC_{\text{main}}\)) // concatenation
        }

        \(PC_{\text{main}} \leftarrow \text{PC from } i_1 \text{ to } i_2\)
        path2 ← \(\phi\)
        
        **while** \(PC_{\text{main}} \neq \phi\) {
            path 2 ← path2 \* shift(\(PC_{\text{main}}\)) \* pop(\(PC_{\text{main}}\))
        }

        pathCovers ← pathCovers \(\cup\) (path1, path2)
    }

    **return** (pathCovers)
}

Figure 4.13 Procedure EdgeFold
**Procedure** NodeSwap (PC){
    PCunfold ← part of PC not folded // nodes that can be transformed
    PCfold ← part of PC folded
    Vswap ← nodes in *inter edges*
    PCswaps ← φ
    PCall ← set of all legal node swaps of Vswap // such that Tdist is reduced
    foreach PCtemp in (PCall){
        PCswaps ← PCswaps ∪ (PCtemp • PCfold) // add concatenated PC to set
    }
    return (PCswaps)
}

Figure 4.14 Procedure NodeSwap

**Procedure** PathInterleave(PCset, Einter){
    PCgroups ← paths with similar nodes in PCset // partition the complete set into groups
    Egroups ← group Einter by PC partitions
    PCset ← φ
    foreach Eset in (Egroups){
        (PCgroup1, PCgroup2) ← PCs joining Eset in PCgroups
        PCpi ← 4 possible interleaves per edge in Eset
        PCset ← best of PCpi
    }
    PCcombine ← best match between PCgroups using Hong’s combination scheme
    Cost ← cost using PCcombine
    return(PCcombine, Cost)
}

Figure 4.15 Procedure PathInterleave
Each folded path and the original paths are then converted into a new form using NodeSwap procedure, shown in Figure 4.14. Each path might have a part that may not be transformed. If the nodes from the candidate nodes are not part of the path fold, they may be swapped. Every swap also restricts further transformation. Every node may be swapped two ways. The NodeSwap procedure returns all the paths from the EdgeFold procedure with candidate nodes swapped.

If two paths have inter edges, they may be interleaved to reduce cost due to inter edges. Procedure PathInterleave, described in Figure 4.15, interleaves paths around candidate nodes. Once all possible interleavings are computed, paths that cannot be transformed are combined end to end, such that the inter edge cost is minimized. The best assignment (newOA) and its cost (newCost) are then returned to SOA2MR heuristic.

The returned cost is compared with the benchmark cost (C). If C > newCost, then the newOA is considered the offset assignment.

### 4.6 Detailed Example

Consider the access graph in the Figure 4.4. The OA provided by Hong’s SOA_MR is “f b a c d e” with a cost of 1 using MR = 2.

\[
\begin{align*}
\text{OA} & \leftarrow \text{“f b a c d e”} \\
\text{C} & \leftarrow 1 \\
\text{MR} & \leftarrow 2 \\
\text{PC} & \leftarrow (\{f b a c\}, \{d e\})
\end{align*}
\]

Procedure EdgeFold is executed twice, first for \text{PC}_{\text{ind}} “f b a c” and then with \text{PC}_{\text{ind}} “d e”. Edgefold transforms path “f b a c” only path “d e” does not have an intra edge.

\[(v_1, v_2) \leftarrow (b, c) \text{ from the candidate edge } <b, c>.
\]
With $PC \leftarrow \text{"f b a c"}$ we get

$PC_{\text{pre}} \leftarrow \text{"f"}$

$PC_{\text{post}} \leftarrow \varphi$

$PC_{\text{main}} \leftarrow \text{"b a c"}$

$PC_{\text{main}}$ can be folded in 2 ways

path1 $\leftarrow \text{"c b a"}$

path2 $\leftarrow \text{"b c a"}$

Now the $PC_{\text{fold}}$ contains 4 paths “f b a c”, “f b c a”, “f c b a”, and “d e”

Paths “f b c a” and “f c b a” yield $PC_{\text{unfold}} \leftarrow \varphi$.

Path “f b a c” may be swapped to get “f a b c” and “f b c a”. Now

$PC_{\text{swap}} \leftarrow (\{f a b c\}, \{f b c a\})$

$PC_{\text{fold}}$ now contains the following path covers (\{f a b c\}, \{f b c a\}, \{f b a c\}, \{f c b a\}, \{d e\})

This set is transformed into OA using procedure PathInterleave.

$P_{\text{groups}} \leftarrow (\{f a b c\}, \{f b c a\}, \{f b a c\}, \{f c b a\})$ and (\{d e\})

$E_{\text{groups}} \leftarrow (<a, >, <a, e>)$

Node “a” is part of the transformation in paths (\{f a b c\}, \{f b c a\}, and \{f c b a\}). These paths cannot be part of a node interleave. Paths \{f b a c\} and \{d e\} may be interleaved, resulting in assignments “f b d a e c” and “f b e a d c”. The transformed paths can also be combined into

- “d e f a b c”, “e d f a b c”, “f a b c d e”, and “f a b c e d”
- “d e f b c a”, “e d f b c a”, “f b c a d e”, and “f b c a e d”
- “d e f c b a”, “e d f c b a”, “f c b a d e”, and “f c b a e d”
The 14 assignments resulting from these transformations are shown in Figure 4.16. These assignments are summarized in the Table 4.2

Figure 4.16 Assignments Produced by Detailed Example. (cont.)
Assignments Produced by Detailed Example. (cont.)
Table 4.2 Cost of Assignments Shown in Figure 4.16

<table>
<thead>
<tr>
<th>Assignments</th>
<th>&lt;b, c&gt;</th>
<th>&lt;a, d&gt;</th>
<th>&lt;a, e&gt;</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>defabc</td>
<td>+/-</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>edfabc</td>
<td>+/-</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>fabcde</td>
<td>+/-</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>fabced</td>
<td>+/-</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>defbcba</td>
<td>+/-</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>edfbca</td>
<td>+/-</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>bcade</td>
<td>+/-</td>
<td>+/-</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>fbcaed</td>
<td>+/-</td>
<td></td>
<td>+/-</td>
<td>0</td>
</tr>
<tr>
<td>defcba</td>
<td>+/-</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>edfcbba</td>
<td>+/-</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>fcbade</td>
<td>+/-</td>
<td>+/-</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>fcbaed</td>
<td>+/-</td>
<td></td>
<td>+/-</td>
<td>0</td>
</tr>
<tr>
<td>fbdaec</td>
<td>MR = 4</td>
<td>+/-</td>
<td>+/-</td>
<td>1</td>
</tr>
<tr>
<td>fbeadc</td>
<td>MR = 4</td>
<td>+/-</td>
<td>+/-</td>
<td>1</td>
</tr>
</tbody>
</table>

There are four possible assignments that have zero cost. Procedure PathInterchange returns the first assignment with zero cost.
4.7 Experimental Results

Random sequences of $|S| \leftarrow 25$ and $|V| \leftarrow 5$ were generated to compare the costs of Hong’s SOA, xformSOA, SOA with MR, and SOA2MR heuristics. Since SOA2MR accounts for the MR value and cost generated in the benchmark SOA with MR, the heuristic generates the best cost. In the experiment, SOA2MR produced an assignment with lower cost than the benchmark as shown in Figure 4.17.

![Figure 4.17 Results from SOA2MR Test with $|S| \leftarrow 25$ and $|V| \leftarrow 5$](image)

4.8 Chapter Summary

We explored new ways of generating assignments given a single offset assignment and a modify register. We introduce three new transformations to SOA that allows MR to be fixed at 2. Edge folding is a technique that brings nodes in a candidate
intra edge within auto increment/decrement range. Node swapping allows nodes to swap positions in the assignment so that no two nodes are more than two nodes away. Path interleaving allows two disjoint paths to be unified into one, allowing nodes of a candidate edge to be accessed by auto-increment or decrement. SOA2MR uses all of these techniques to obtain best possible assignment.
5 General Offset Assignment (GOA)

In Chapters 3 and 4, we concentrated on systems that use one AR and one MR, if any. In some architectures, as shown in Figure 2.2, there may be more than one AR and perhaps more than one MR. Such cases are referred to as \(k\) AR and \(n\) MR architectures, where \(k \geq 1\) and \(n \geq 0\). In this chapter we primarily discuss a system with \(k\) ARs. Each AR may be associate with its own MR. In such instances the system has \(k\) ARs and \(k\) MRs. Some systems share a pool of MRs, where \(k\) ARs have access to \(n\) MRs and usually \(k > n\).

It is common to partition the variable set into disjoint partitions and associate each partition with an AR. Rarely is more than one AR is used to access a single variable. Heuristics are explored that use more than one ARs to access the same variable.

5.1 Related Research

We primarily look at three fundamental approaches – by Liao [55], Leupers [59], and Hong [45]. Liao extends his work on SOA and utilizes the results to generate an offset assignment for \(k\) address registers. Cost for the SOA is expected to be lower using more ARs. A subset \((P)\) of list of variables \((L)\) is derived. Liao’s SolveGOA partitions the variables into 2 sets \((P, L-P)\). The cost of the assignments \(P\) and \(L-P\) is calculated as \(H_1\) and \(H_2\) respectively. If the cost of the assignments \(H_1\) and \(H_2\), along with setup cost for the new address register is higher than the cost of initial assignment \(H\), the initial assignment is use; otherwise, GOA is again attempted on the set of variables in \(L-P\). This process is repeated with at most \(k\) address registers.
In its trivial form, a selection of two variables (P) from the original set of variables (L) can be assigned to an AR without additional cost. If the cost of an assignment with L-P variables is less than cost (H), GOA heuristic is applied to variables in (L-P). If k greater than (number of variables in L)/2, 2 variables can be assigned to each AR for the cost of initialization. If cost (H) is greater than (number of variables in L)/2, the GOA problem has a trivial solution.

Leupers improves on Liao’s solution. Leupers determines a procedure for selecting nodes for each of the k ARs. In Leupers’ SolveGOA heuristic, the set of edges L is sorted in descending order of weight. An edge is associated with each of the k ARs, such that nodes belonging to each AR form a disjoint partition of the entire set of nodes (V). The partitioning is complete when no more edges can be assigned to an AR or if all the ARs have an edge (i.e., a pair of nodes) assigned to them.

If some nodes are still not part of the ARs, then they have to be assigned to any of the k partitions. The heuristic computes the cost of associating the node with each partition. Each node is added to the AR that adds the minimum cost. This heuristic generates offset assignments for k ARs more efficiently than Liao’s GOA heuristic.

Leupers’ heuristic does not consider the order in which edges are inserted into the partition associated with each AR. Hong et al., consider a GOA heuristic based on the frequency of a variable in the access sequence. Since the frequency of a variable in the access sequence is a static property of the variable, the property is constant throughout the heuristic.

Hong’s GOA heuristic GOA_FRQ sorts the variables in descending order of their frequencies. The two nodes are associated with each AR till the cost of the sequence of
the remaining variables is greater than one or no more ARs are available to which to partition the variables. The variables not yet associated with the k ARs are associated with an address register one at time, such that the added cost is the smallest. The resulting partition of the variables into at most k partitions is the GOA using frequency as the primary criteria.

5.2 Motivation

Leupers does not specify the order in which the nodes not included in the k ARs are picked. This order of selection has an impact on the cost of the assignment. This issue has been addressed by Hong, by using frequency of the variables as the attribute that determines the order in which variables from an access sequence are selected.

We hypothesize that the degree of a variable in the access graph is a better attribute to consider. Consider the access sequence “a a a a b c b c d d d d d”. This sequence yields the access graph shown in Figure 5.1.

![Figure 5.1 Access Graph for GOA Motivation](image)

The frequency of a and d is five, both have a degree of one, discounting self-loops that do not add to the cost of an assignment. From the graph, variables b and c, which have frequencies two and three respectively, have higher degree of two. In this dissertation we consider degree to be more important and consider the effect of choosing variables that have high and low degree.
5.3 GOA_DEG

GOA_DEG(V, s, k) { // V is # of variables, s is access sequences, k is number of registers
    AG_v ← access sequence (s)
    V_deg ← sorted list of variables in descending order of degree in AG_v
    bestCost ← SOA_Cost(V, s)
    if(bestCost ≤ 2){ // initialization cost + 1
        return (V)
    }
    K_min ← minimum (k, bestCost, number of variables / 2)
    (v_a, v_b) ← top 2 nodes from V_deg
    V_deg ← V_deg - (v_a, v_b)
    S_deg ← s - (v_a, v_b)
    V_0 ← (v_a, v_b)
    i ← 1
    while ( i < K_min and |V_deg| > 1){
        if (SOA_Cost(V_deg, S_deg) ≤ 2) {
            V_i ← V_deg; return(V_0 .. V_i)
        }
        j ← 0
        foreach V_temp (V_0 .. V_i) {
            S_temp ← s - (V_0 .. V_i) + V_temp
            AG_temp ← (S_temp)
            V_test ← top node not in V_temp
            C_j ← SOA_Cost(V_temp ∪ V_test, S_temp)
            if(C_j == 1){
                index = j;break;
            }
        }
        j ← j + 1
    }
}

Figure 5.2 GOA_DEG Heuristic (cont)
if \((C_j = 1)\) {
    \(V_{\text{index}} \leftarrow V_{\text{index}} \cup V_{\text{test}}\)
} else {
    \((v_a, v_b) \leftarrow \text{top 2 nodes from } V_{\text{deg}}\)
    \(V_i \leftarrow (v_a, v_b)\)
    \(V_{\text{deg}} \leftarrow V_{\text{deg}} - (v_a, v_b)\)
    \(S_{\text{deg}} \leftarrow s - (v_a, v_b)\)
    \(i \leftarrow i + 1\)
}

if \((V_{\text{deg}} = \emptyset)\) {
    return \((V_0 .. V_{i-1})\)
}

while \((V_{\text{deg}} \neq \emptyset)\)
    \(V_{\text{temp}} \leftarrow \text{top node from } V_{\text{deg}}\)
    \(j \leftarrow 1\)
    lowCost \leftarrow \text{SOA Cost}(V_0 \cup V_{\text{temp}}, S_o \cup V_{\text{temp}})\)
    foreach \(V_{\text{temp}} (V_1 .. V_i) \) {
        \(S_{\text{temp}} \leftarrow s - (V_0 .. V_i) + V_{\text{temp}}\)
        \(AG_{\text{temp}} \leftarrow (S_{\text{temp}})\)
        \(V_{\text{test}} \leftarrow \text{top node not in } V_{\text{temp}}\)
        \(C_j \leftarrow \text{SOA Cost}(V_{\text{temp}} \cup V_{\text{test}}, S_{\text{temp}})\)
        if \((C_j < \text{lowCost})\) {
            \(\text{index} = j; \text{lowCost} = C_j\)
        }
        \(j \leftarrow j + 1\)
    }
    \(V_{\text{index}} \leftarrow V_{\text{index}} \cup V_{\text{temp}}\)
}

return \((V_0 .. V_{i-1})\)
In the GOA_DEG heuristic, the access sequence \( s \) is used to generate the access graph \( AG_v \). A list of variables sorted by degree, \( V_{\text{deg}} \), is computed from \( AG_v \). Before the variables are partitioned to various ARs, the SOA_Cost of the access sequence is computed. If the cost is less than 2, additional ARs may not have any benefit as additional AR’s initialization cost will be the lower bound on the cost and may increase the cost of an assignment.

GOA cost should be less than the cost of SOA, \( \text{bestCost} \), and no more than half the number of variables, \( |V|/2 \). Therefore maximum number of address registers used is the minimum of SOA Cost, half the number of variables, and given AR limit, \( k \). If GOA cost exceeds SOA limit, the primary purpose of using additional registers is negated. If an access sequence has 20 variables, no more than ten ARs are required as an assignment of two variables per AR will result in ten assignments of cost zero.

The two variables with the highest degree in \( V_{\text{deg}} \) are selected for the first AR. If the cost of the access sequence containing the rest of the variables does not exceed 2, these variables are assigned to the next AR. If the cost exceeds two, a new access graph \( AG_{\text{temp}} \) is generated and two nodes that have highest degree are chosen for the next AR.

In each subsequent iterations, an access graph \( AG_{\text{temp}} \) is generated for each access graph, and the node with highest degree not already included in the AR is selected for the test access sequence. The resulting access sequence is used to find SOA cost. If the cost is one, the variable is included in the AR and the process is repeated; otherwise, this process is repeated with all ARs that have variables associated with it.

If \( V_{\text{deg}} \) is \( \emptyset \), then all variables in \( V_{\text{deg}} \) are allocated to ARs \( \leq k \); if \( V_{\text{deg}} \neq \emptyset \), the remaining variables are allocated to the \( k \) ARs, choosing 1 variable at a time and
assigning to an AR that produces the lowest cost change with the addition of the variable. The heuristic is complete when $V_{deg}$ is $\emptyset$ and the input access sequence is converted into multiple assignments.

### 5.4 Detailed Example

Consider the access sequence $s = \text{“a b c d e f a d a d a c a d f b”}$. The set of nodes $V$ is $(a, b, c, d, e, f)$. The access graph and the SOA for this sequence are shown in Figure 5.3.

![Access Graph and its Offset Assignment](image)

**Figure 5.3** Access Graph and its Offset Assignment for “a b c d e f a d a d a c a d f b”

- $\text{bestCost} \leftarrow 6$
- $K_{min} \leftarrow \text{minimum (6, 6/2, k)}$
- Ignoring $k$, $K_{min} \leftarrow 3$
- $V_{deg} \leftarrow (a (4), d (4), f (4), c (3), b (3), e (2))$
- $V_1 \leftarrow (a, d)$
- $s_{deg} \leftarrow (b c e f c f b)$
- $V_{deg} \leftarrow (f (3), c (3), b (2), e (2))$
The access graph and assignment for $s_{\text{deg}}$ is shown in Figure 5.4.

![Figure 5.4 Access Graph and Offset Assignment for “b c e f c f b”](image)

V$_2$ ← (c, f)

$s_{\text{deg}}$ ← (b e b)

V$_{\text{deg}}$ ← (b (1), e (1))

The SOA Cost of $s_{\text{deg}}$ is 1. The node in V$_{\text{deg}}$ is assigned to V$_3$. The original access sequence is now partitioned into two access sequences containing variables ((a, d), (c, f), (b, e)) for a cost of 3.

### 5.5 Variations in Heuristic

We discuss some variations of the heuristic in this chapter. It may be argued that combining one variable with high degree in the access graph with variables that have low degree will permit a larger number of variables in each partition than that shown in the example. Liao considers number of variables in each register in the range of two to six. For this variation, the following statement in the heuristic

$$(v_a, v_b) \leftarrow \text{top two nodes from } V_{\text{deg}}$$

may be modified to
(v_a) ← top node from V_{deg}
(v_b) ← last node from V_{deg}

Conversely, nodes that are low in degree could be selected for inclusion in a given partition. For this variation, node selection statement may be changed to

(v_a, v_b) ← last 2 nodes from V_{deg}

We suggest a further variation that eliminates the distribution of remaining variables. In this variation, we suggest partitioning the variables into k-1 partitions. Any variables remaining in V_{deg} after k-1 partitions are assigned to AR_k.

5.6 Experimental Results

GOA_{DEG} was tested with randomly generated sequences. Four variations of the heuristic were implemented:

- DHH – both variables have highest degree
- DHL – 1 variable has highest degree while another lowest degree
- DLL – both variables have lowest degree
- Rand – both variables were chosen at random.

Each heuristic was iterated through 100 access sequences of length |S| and number of variables |V|. The results of these runs are summarized in Table 5.1 and Figure 5.5. Results of these experiments indicate similar partitioning with various variations of GOA_{DEG}. When the ratio of |S| to |V| is low DHH exhibits a remarkable advantage over other variations. In other instances, the variations do not show marked difference in the number of partitions. DHL

DEG_GOA and its variations are also compared with other GOA implementations. For this run, we used Hong’s implementation of Liao, Leupers, and Hong’s GOA algorithms. The results of these iterations are listed in Table 5.2. GOA_{DEG} uses a lower number of ARs on average. From the entire data, we observe that DHL shows the best results, while DHH has an advantage when the ratio of |S| to |V| is low.
Table 5.1 Variations of GOA_DEG and their Results

<table>
<thead>
<tr>
<th></th>
<th>DHH</th>
<th>DHL</th>
<th>DLL</th>
<th>Rand</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-5</td>
<td>2.04</td>
<td>1.87</td>
<td>1.96</td>
<td>1.99</td>
</tr>
<tr>
<td>20-5</td>
<td>2.54</td>
<td>2.5</td>
<td>2.48</td>
<td>2.49</td>
</tr>
<tr>
<td>20-15</td>
<td>3.61</td>
<td>4</td>
<td>4.17</td>
<td>4.02</td>
</tr>
<tr>
<td>25-5</td>
<td>2.82</td>
<td>2.74</td>
<td>2.82</td>
<td>-</td>
</tr>
<tr>
<td>25-6</td>
<td>2.9</td>
<td>2.98</td>
<td>2.98</td>
<td>2.96</td>
</tr>
<tr>
<td>50-7</td>
<td>3.88</td>
<td>3.81</td>
<td>3.85</td>
<td>-</td>
</tr>
<tr>
<td>50-8</td>
<td>3.95</td>
<td>4</td>
<td>3.99</td>
<td>3.98</td>
</tr>
<tr>
<td>50-10</td>
<td>4.86</td>
<td>4.73</td>
<td>4.89</td>
<td>4.85</td>
</tr>
<tr>
<td>50-40</td>
<td>5.01</td>
<td>6.06</td>
<td>5.97</td>
<td>6.07</td>
</tr>
<tr>
<td>80-10</td>
<td>5</td>
<td>4.99</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>100-10</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>100-15</td>
<td>7.69</td>
<td>7.41</td>
<td>7.56</td>
<td>7.39</td>
</tr>
<tr>
<td>100-16</td>
<td>7.95</td>
<td>7.54</td>
<td>7.82</td>
<td>7.83</td>
</tr>
<tr>
<td>100-50</td>
<td>7.55</td>
<td>7.91</td>
<td>7.87</td>
<td>7.8</td>
</tr>
<tr>
<td>100-80</td>
<td>5.18</td>
<td>6.18</td>
<td>6.14</td>
<td>6.07</td>
</tr>
</tbody>
</table>

Table 5.2 Comparison of GOA_DEG with other GOA Heuristics

<table>
<thead>
<tr>
<th></th>
<th>Liao</th>
<th>Leupers</th>
<th>Hong</th>
<th>DHH</th>
<th>DHL</th>
<th>DLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>100-15</td>
<td>7.69</td>
<td>7.69</td>
<td>7.69</td>
<td>7.69</td>
<td>7.41</td>
<td>7.56</td>
</tr>
<tr>
<td>80-10</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4.99</td>
<td>5</td>
</tr>
<tr>
<td>50-7</td>
<td>4.57</td>
<td>4.57</td>
<td>4.57</td>
<td>3.88</td>
<td>3.81</td>
<td>3.85</td>
</tr>
<tr>
<td>25-5</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
<td>2.82</td>
<td>2.74</td>
<td>2.82</td>
</tr>
</tbody>
</table>
Figure 5.5 Trend Graph Comparing Versions of GOA_DEG

5.7 Chapter Summary

General offset assignment is the partitioning of the variables into k offset assignments with lowest cost. Existing GOA heuristics use edge weight and variable frequency to partition the variables. We believe that the degree of a node in the access graph is more critical a factor in determining partitions.

We propose a heuristic GOA_DEG that uses the degree of a variable in the access graph as the primary criterion. We show various variations in the choosing of variables for each partition. We show that choosing a variable with highest degree first, followed by the lowest has the most benefit. In special cases, selecting nodes with highest degree
exhibits significant difference. GOA_DEG compares favorably with some popular GOA heuristics.
6 Offset Assignment with SSA

Static Single Assignment (SSA) [20] is a technique to optimize code. SSA is an intermediate step that allows optimizations to be performed easily and efficiently. Each variable in the code is written to (i.e., assigned a value) exactly once in its life time. SSA also contains φ-functions that combine such optimizations if different assignments were made to a variable in each incoming path.

Figure 6.1 shows the conversion of a basic block into its Single Assignment (SA) form. In this example variable x is assigned values twice. Each such assignment is changed to a different value. Optimizing compilers use the SSA form to transform basic blocks to optimized code.

```
x ← a            x1 ← a
p ← x            p ← x1
x ← b            x2 ← b
q ← x            q ← x2
```

Figure 6.1 Basic Block and its SSA-Form

```
if TRUE  
x ← a   x1 ← a
else    
x ← b   x2 ← b
end if  
```

Figure 6.2 Branch-Join and its SSA-Form
Programs also have branch and join nodes in the form of “if-else” statements. Figure 6.2 shows an example of using \( \varphi \)-function to combine two separate assignments to variable \( x \).

6.1 SOA with SSA-Form

The offset assignment will be derived for the basic block shown in Figure 6.3. This basic block is first converted into its SSA-form. For the first statement is

\[
c \leftarrow c + d + f
\]

The expression “\( c + d + f \)” is assigned to variable \( c \). The conversion to SSA-form requires that each assignment of a variable be performed exactly once. The SSA-form of this statement is

\[
c_1 \leftarrow c + d + f
\]

Similar transformations are performed on assignments of variables \( a, b, \) and \( c \). This form will have the same access length as the basic block while making the access graph sparser.

\[
\begin{align*}
c & \leftarrow c + d + f & c_1 & \leftarrow c + d + f \\
a & \leftarrow h + c & a & \leftarrow h + c_1 \\
b & \leftarrow b + e & b_1 & \leftarrow b + e \\
c & \leftarrow g + b & c_2 & \leftarrow g + b_1 \\
a & \leftarrow a + c & a_1 & \leftarrow a + c_2
\end{align*}
\]

Figure 6.3 Basic Block for SOA and its SSA-Form

The access sequence is now \( cdfc_1hc_1abe_1gb_2e_2ac_2a_1 \). The access graph for this sequence is given in Figure 6.4. The cost of the assignment shown for the original basic block is 4 with eight variables. The cost of the new path cover shown as bold edges in Figure 6.4 is 2, though the number of variables has increased to eleven. Usually data
memory is easier to optimize than program memory; thus an increase in the number of variables, with a decrease in the cost of variable access is preferred.

Figure 6.4 Access Graph and Path for cdfc1hc1abeb1gb2c2ac2a1

Conversion to SSA changes a given access graph into a sparser access graph reducing the weights of some of the edges while creating new edges. A new access graph with more edges for the same access sequence usually generates a path cover whose cost is lower than a denser graph with less number of edges.
6.2 Commutative Transformation of SSA-Form

The SSA-form relies on the reassignment of every variable that is written. Conversion to SSA is performed on the basic block similar to a commutative transformation. This allows us to use commutative transformations, discussed in detail in xformSOA (Chapter 3), on a given basic block to derive a more optimal offset assignment.

Figure 6.5 shows a randomly generated basic block with \(|S| \leftarrow 40\) and \(|V| \leftarrow 7\). The basic block generated for this sequence and its SSA form are given in Figure 6.5.

\[
\begin{align*}
f & \leftarrow d + d \\
d & \leftarrow a + e \\
f & \leftarrow f + e \\
c & \leftarrow b + a \\
b & \leftarrow c + d \\
f & \leftarrow c \\
a & \leftarrow c \\
b & \leftarrow g + e \\
c & \leftarrow c + b \\
g & \leftarrow g + b \\
e & \leftarrow c + d \\
g & \leftarrow b + f \\
b & \leftarrow d + c \\
d & \leftarrow e + f
\end{align*}
\]

The access graph generated for the basic block is given in Figure 6.6, the access sequence for which is “ddfaedfedefbacedbcfegabecegbcbgcdebfdebfed”. The SOA cost for this access sequence is 21. The access sequence for the SSA-form of this basic block is “ddfaed1fedefbaced1b1cf2ca1geb2cb2c1gb2g1c1d1e1b2f2g2d1c1b3e1f2d2” and its access graph is
shown in Figure 6.8. The cost of this assignment is 20. There is an immediate gain in the cost of this assignment.

Figure 6.6 Access Graph of a Basic Block |S| ← 40, |V| ← 7

Figure 6.7 Basic Block Commutatively Transformed
A commutative transformation may be performed on the SSA-form of the basic block. xformSOA may be applied to such a basic block. The result of this transformation is an access graph, whose cost is 17. The access graph and its path are as shown in Figure 6.9.

Figure 6.8 Access Graph of Basic Block in SSA-Form
Figure 6.9 Path Cover of the Transformed Basic Block/SSA-Form
It may however be noted that $xformSOA$ of the basic block results in a transformation, the cost of which is 13. The SSA form of the basic allows variables to be grouped, facilitating variable space reuse. $xformSOA$ may be used on

### 6.3 Variable Space Reuse

In SSA-form of a basic block the same variable is used in more than one form, allowing the use the same space for more than one variable, as a variable expires after the last read. If a variable is assigned a value, then it takes on its new SSA-form, which implies that the space used by the variable from its last read to its write, may be used for any other variable.

![Figure 6.10 Lifetimes of a Variable using SSA-Form](image)

From the grid shown in Figure 6.10, we can deduce the number of variables required in each step. In step 5, we need no more than three variable spaces, while in step 1, seven variable locations are required. The maximum number of variables among these steps limits the minimum number of variables required to compute the whole block. Five of the variables may use a variable space which is no longer used. Twelve variables need only seven spaces to compute the five statements.
Variables that can use the same space can be grouped together and a new access sequence is generated to find the offset assignment. Variables may be grouped together in more than one way.

\[

table
\begin{array}{cccccccccccc}
<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>d</th>
<th>f</th>
<th>c_1</th>
<th>h</th>
<th>a</th>
<th>e</th>
<th>b</th>
<th>b_1</th>
<th>g</th>
<th>c_2</th>
<th>a_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{array}
\]

Figure 6.11 Variables Grouped for Space Reuse

Once the variables are grouped, each group is named uniquely and a new sequence is generated. From Figure 6.11 we have the following variable grouping

\[
\begin{align*}
x & \leftarrow (a_1, c_2, g) \\
y & \leftarrow (b_1, b) \\
z & \leftarrow (a, h) \\
p & \leftarrow (c_1, f)
\end{align*}
\]

This assignment converts the basic block to

\[
\begin{align*}
p & \leftarrow c + d + p \\
z & \leftarrow z + p \\
y & \leftarrow y + e \\
x & \leftarrow x + y \\
x & \leftarrow z + x
\end{align*}
\]

and the access sequence to “c d p p z p z y e y x y x z x x”.
The access graph generated from this transformation is shown in Figure 6.12.

The cost of the assignment is one.
This basic block assigns variables into groups, which might make the access graph denser. Commutative transformations may further be performed on such sequences. In this instance, the cost of such transformations is still one, the access graph for which is shown in Figure 6.13.

6.4 SOAwithSSA

The framework discussed earlier in this chapter is part of the SOAwithSSA heuristic shown in Figure 6.14. The start and end times of each variables life time is computed. The variables are sorted in the descending order of their start times. Each variable is grouped with a variable, the end time of which is earlier than its start time. This greedy approach groups variables into groups that can utilize the same variable space. The results of this heuristic may be used with any SOA/GOA heuristic.

6.5 Chapter Summary

SSA is a technique that can be used to optimize code. It is currently employed in optimizing compilers. SSA can also help with offset assignment. Converting a basic block into its SSA form makes an access graph sparse. A sparser graph is likely to have an offset assignment with lower cost.

The SSA form of a basic block converts one variable into multiple variables that have defined start and end times. Variables with life spans that do not overlap may be grouped together and can be assigned one variable space. This process tends to reduce the total space required for the execution of a program.

SOAwithSSA is an SOA heuristic that uses the SSA form of a basic block to generate offset assignment. SOAwithSSA reuses variable space to optimize program.
\textbf{SOAwithSSA}(BB) \{ // BB – Basic block

\begin{verbatim}
  BBSSA ← SSA_FORM(BB)

  V ← list of all nodes in BBSSA
  VSSA ← V – list of all nodes that carry over
  V_start ← VSSA sorted in nodes start time, in decreasing order
  V_end ← VSSA sorted by variable end-time, in decreasing order
  i ← 0

  while (V_start ≠ φ) {
    v ← shift (V_start) // remove top node and assign it to v
    V_end ← V_end – v
    G[i] ← v

    \textbf{foreach} V_test in (V_end) {
      if (StartTime(v) > EndTime(V_test)) {
        G[i] ← V_test
        V_start ← V_start - V_test
        V_end ← V_end - V_test
        v ← V_test
      }
    }
    i ← i + 1
  }

  Every G\{i\} that has > 1 variable assign a variable new name

  BBSSOA ← BBSSA substituted with G[i]
  SOA(BBSSOA)
\}
\end{verbatim}

Figure 6.14 SOAwithSSA Heuristic
7 Incrementally Improving SOA

Extensive work has gone into the problem of variable placement in codes for embedded processors. SOA heuristics return an assignment that is a concatenation of the disjoint path covers. Hong’s technique [45] of combining disjoint paths might yield a better assignment. Tie-break functions used in SOA heuristics result in the generated offset assignment. Choosing an alternate edge, might have resulted in a better assignment. We believe that alternate locations for variables in uncovered edges might result in an assignment with lower cost. Such options can be incrementally tested.

7.1 Deficiencies in SOA

Two deficiencies are found in Liao’s Solve-SOA [55]. First, even though the edges are sorted in descending order of weight, the order of consideration of the edges of the same weight are ordered is not specified. We believe this to be important in deriving the optimal solution. Second, the maximum weight edge is always selected since this is a greedy approach. The proposed algorithm addresses both these cases. This algorithm takes as input an offset sequence, produced by any means, and tries to include edges not previously included in the cover. Consider the example of Figure 7.1.

Consider the code sequence shown in Figure 7.1(a). Its corresponding access sequence is shown in Figure 7.1(b) which translates into the access graph in Figure 7.1(c). Path cover shown in Figure 7.2 (a), as bold edges, is one possible outcome of Liao’s Solve-SOA using the access sequence in Figure 7.1(b).
The offset assignment associated with the cover is \(a,d,b,c,e\) or \(d,b,c,e,a\). This solution is clearly a non-optimal solution. The cost of this assignment is 2. The optimal solution would be \(d,b,a,c,e\). It is possible to have achieved the optimal cost of 1 by having considered either edge \((a,b)\) or edge \((a,c)\) before edge \((b,c)\). But since Solve-SOA does not consider the relative positioning of the edges of the same weight in the graph, we get a cost of 2. The solution that is produced by the proposed algorithm attempts to optimize the results obtained from other heuristics. A possible solution to the suboptimal cover shown in Figure 7.2(a) is shown as an optimal cover in Figure 7.2(b).
7.2 Related Work

Atri et al.,[7-9] proposed an incremental algorithm to address the issue of obtaining optimal solutions from suboptimal offset assignments. Figure 7.3 shows their algorithm in detail.

1. //INPUT: Access Sequence AS, Initial Offset Assignment, OA
2. //OUTPUT: Final Offset Assignment
3. Procedure Incremental-Solve-SOA(AS, OA)
4. $G = (V, E) \leftarrow \text{AccessGraph}(AS)$
5. BEST $\leftarrow$ Initial Offset Assignment OA
6. repeat
7. $E^\text{outer}_\text{sort} \leftarrow$ sorted list of unselected edges from BEST configuration
8. OUTER_FLAG $\leftarrow$ FALSE

Figure 7.3 Incremental-Solve-SOA [7, 9] (cont)
9. Unlock all edges in $E^U_{\text{sort}}$
10. INNER_BEST ← BEST
11. repeat
12. INNER_FLAG ← FALSE
13. $e$ ← topmost edge from $E^U_{\text{sort}}$
14. $(A_0, \ldots, A_3)$ ← 4 possible assignments due to $e$
15. // An assignment is illegal if it involves changing a locked edge
16. // Otherwise, an assignment is legal
17. $S$ ← the set of legal assignments from $(A_0, \ldots, A_3)$
18. if (S has at least 1 legal assignment)
19. INNER_FLAG ← TRUE
20. CURRENT ← MinCost(S)
21. Lock the edges that change
22. Delete the locked edges from $E^U_{\text{sort}}$ ensuring that $E^U_{\text{sort}}$ stays sorted
23. if (CostOf(CURRENT) < CostOf(INNER_BEST))
24. INNER_BEST ← CURRENT
25. endif
26. else if ($E^U_{\text{sort}} \neq \emptyset$)
27. INNER ← TRUE
28. endif
29. until (INNER_FLAG ≠ TRUE)
30. if (CostOf(INNER_BEST) < CostOf(BEST))
31. BEST ← INNER_BEST
32. OUTER_FLAG ← TRUE
33. endif
34. until (OUTER_FLAG ≠ TRUE)
35. return BEST

This algorithm picks the maximum weight edge not included in the cover and tries to include it. The process of inserting a node into the cover is detailed next. Let the
maximum weight edge not included in the cover be between two variables $a(n)$ and $a(n + x)$, in that order. We consider the case where we try to include that edge and see the effect on the cost of an assignment. There are four possible offset assignments when we try to bring two variables together not previously adjacent. The initial offset assignment is assumed to be $a(n - 1) a(n) a(n + 1) ... a(n + x - 1) a(n + x) a(n + x + 1) ...$. We consider the following four sequences that would result when edge $(a(n)a(n + x))$ is included in the cover:

$a(n - 1) a(n + x) a(n) a(n + 1) ... a(n + x - 1) a(n + x + 1) ....$

$a(n - 1) a(n) a(n + x) a(n + 1) ... a(n + x - 1) a(n + x + 1) ....$

$a(n - 1) a(n + 1) ... a(n + x - 1) a(n) a(n + x) a(n + x + 1) ....$

$a(n - 1) a(n + 1) ... a(n + x - 1) a(n + x) a(n) a(n + x + 1) ....$

For each legal assignment, the cost is computed. If the assignment cost is less than the current best assignment cost, then the new assignment is accepted. This process is employed on all the edges in the $E^U_{\text{sort}}$ list. If no assignment results in an assignment cost less than the original assignment, the original assignment is considered optimal, else the new assignment is accepted as an optimal assignment.

### 7.3 Limitations in Existing Incremental Algorithms

The algorithm in Figure 7.3 is based on the assumption that an edge with highest edge weight might provide the highest cost benefit and would need to be included in the assignment first. There is no proof to support this assumption. It is entirely possible to include a higher weight edge after the inclusion of a lower weight edge, while an ordered inclusion does not test a higher weight edge once discarded. This situation prompts us to make the first modification – test the effect of the inclusion of all edges in $E^U_{\text{sort}}$ instead.
of testing serially in decreasing order of the weights. With this method, it is possible to get a lower cost assignment than using a higher weight edge, if such an assignment exists. Also, such an assignment might make the inclusion of higher weight edge later feasible.

Each of the edges that are not included in the cover has a reason for its exclusion from the cover. This information, if available while performing the incremental modification, may reduce computations made to determine if inclusion of an edge is legal. An edge may be excluded from a cover for various reasons.

1. If edge <a, b> and <c, d> have the same edge weight and the tie-break function cannot resolve their precedence, then only one of the edges is chosen. This arbitrary selection of edges, as the selection of one of the edges is delayed, could be reversed in incremental algorithm, which might result in a lower cost.

2. An edge <a, b> might cause a cycle, if path axyzb exists in the path cover. Knowledge of this can reduce legality checking.

3. An edge <a, b> might violate degree condition of the path at node a, if edges <a, x> and <a, y> are included in the cover.

This set of information about unselected edges may be used in incremental algorithm to reduce computation and also prioritize a set of edges over the other. This information may be collected from any SOA heuristic and is the second modification to incremental SOA algorithm. The incremental algorithm that includes these modifications is listed in Figure 7.4.

### 7.4 Improved-Incremental-SOA

1. //INPUT: Access Sequence AS, Initial Offset Assignment OA, Priority Edge PE, Cycle Edge CE, Degree Edge DE
2. // PE, CE, DE contain info about why they belong to those sets

Figure 7.4 Improved-Incremental-SOA

3. //OUTPUT: Final Offset Assignment

4. Procedure Incremental-Solve-SOA(AS, OA, PE, CE, DE)

5. G = (V,E) ← AccessGraph(AS)

6. BEST ← Initial Offset Assignment OA

7. INNER_BEST ← BEST

8. Add BEST to a set of assignments and costs – MinOS

9. AllOS - all legal offset assignments; initialize cost array

10. repeat

11. EU ← list of unselected edges from INNER_BEST configuration

12. INNER_FLAG ← FALSE

13. //first modification

14. foreach e ← EU

15. (A₀, …, A₃) ← 4 possible assignments due to e

16. // do not compute an assignment if conditions of CE and DE are valid

17. S ← the set of legal assignments from (A₀, …, A₃)

18. if (S has at least 1 legal assignment and S is not in AllOS)

19. INNER_FLAG ← TRUE

20. compute all legal S and costs and add to AllOS

21. cost[S] ← MinCost(S)

22. endif

23. end for

24. if (INNER_FLAG = TRUE)

25. INNER_BEST = MinCost(S from cost[S]

26. add INNER_BEST to MinOS

27. endif

28. initialize cost array

29. until (INNER_FLAG ≠ TRUE)

30. BEST ← MinCost(MinOS)
31. return BEST

In the previous algorithm, unselected edges were sorted and incrementally tested for changes to the assignment. Here we compute all possible legal assignments and find the assignment with the least cost. This cost may be higher than the cost of the current assignment. Use the current assignment to find other lower cost assignments, while computing legal assignments. This process will exhaustively compute legal assignments and their costs. The function minCost(S) will compute the cost of an assignment S in any given iteration. AllOS[S] tracks all legal offset assignments and their costs. Keeping track of computations already performed reduces the cost of computing assignments.

This algorithm is aggressive in computing more assignments than the earlier algorithm. In addition this algorithm tries to use the following knowledge to limit unnecessary computation.

- If an edge is in the set of edges in <c, b> and <d, e>, and the conditions in current assignment are still valid, do not compute the assignment.
- If an edge is in <p, e>, and the edge with equal tie-breaking value is in the assignment, replace that edge – unless it was chosen in this iteration.
- If “x a y” is a sequence in the assignment and edge <a, b> is to be tested, do not compute this assignment if it had already been computed before. Use the computed value in subsequent iterations.

7.5 Improved Incremental Assignment with MR and GOA

Let the weight edge not included in the cover be between two variables \(a(n)\) and \(a(n + x)\), in that order. Consider again the case where we try to include that edge and see the effect on the cost of an assignment. There are four offset assignments when we try to
bring two variables together not previously adjacent, as mentioned earlier in section 7.3.

The initial offset assignment is \(a(n - 1) a(n) a(n + 1)\ldots a(n + x - 1) a(n + x) a(n + x + 1)\ldots\) When we consider the effect of the Modify Register (MR), we have 4 additional assignments. If the value in MR is \(d\), the said assignment can be modified into the following four assignments

\[
\begin{align*}
a(n - d) a(n + x) a(n - d + 1) & \ldots a(n - 1) a(n) a(n + 1) \ldots a(n + x - 1) a(n + x + 1)\ldots \\
a(n - 1) a(n) a(n + 1) & \ldots a(n + d - 1) a(n + x) a(n + x + d)\ldots a(n + x - 1) a(n + x + 1)\ldots \\
a(n - 1) a(n + 1) & \ldots a(n + x - d) a(n) a(n + x - d + 1)\ldots a(n + x - 1) a(n + x) a(n + x + 1)\ldots \\
a(n - 1) a(n + 1) & \ldots a(n + x - 1) a(n + x) a(n + x + 1)\ldots a(n + x + d - 1) a(n) a(n + x - d)
\end{align*}
\]

For an SOA with MR, the incremental-SOA algorithm has to compute eight assignments and their costs, instead of four as detailed in the algorithm.

This algorithm may also be extended to General Offset Assignment (GOA). With GOA, consider the edges that are not covered by any address register. The edges not covered belong to an address register and its cover. Each address register and the edges that were not covered by it in its partition, may be addressed as an independent SOA problem, allowing the solving of \(k\) SOAs with incremental SOAs to obtain a better assignment.

### 7.6 Node-Based SOA

SOA heuristics have been observed to be primarily edge-based. In some instances such as GOA\_DEG, the primary arbiter has been a nodes property. Using a node’s property for selection of nodes to partition a set of variables produced useful results. We try and extend this notion to Single Offset Assignment. We test the effectiveness of this approach by using it as procedure \textit{measure} in \textit{xformSOA} of Chapter
We use the access graph shown in Figure 3.17(a) and its associated basic block. The cover that is selected based on nodes in each stage is different as shown in Figure 7.6. The final transformed access graph has a cost of 2, which is the optimal cost for this access sequence and its transformations.

**SOAbyNode**(BB) {//BB - Basic block

  AS ← AccessSequence(BB)
  AG ← AccessGraph(AS)
  NodeDegree ← degree(AG)
  NodeWeight ← Weight(AG) // weight of all edges incident at node
  AvgWeight ← NodeWeight/NodeDegree for each node V // sorted descending
  BR ← list of breakable edge from AG
  UB ← list of unbreakable edges

  foreach node (v) in (AvgWeight){
    cover ← v
    Select an edge with v with has highest weight, unbreakable
    if count of v in cover = 2{
      delete v from AvgWeight
      delete all incident edges in AG, UB, BR
      recompute AvtWeight
    }else{
      delete edge from AG, UB, BR
    }
  }

  cost(cover) // cost of all edges not covered by node selection order
}

Figure 7.5 SOAbyNode Heuristic
An offset assignment from a heuristic that may be suboptimal can further be optimized with incremental solutions. Currently available solutions assume that edges with higher weights should be considered first. We believe that various factors affect the selection of an edge, such as tie-breaking function, cycle, and node degree. So we consider finding all possible assignments for all unselected edges and choosing the most
optimal solution. Our algorithm uses the results of computations already performed, and avoids redundancy. Though the solution is exhaustive, it can be implemented to improve any other heuristic.

This solution can be extended to a system with one address register and a modify register (SOA with MR) and a system with k address registers (GOA).

The SOA problem is approached with the goal of including most common transitions in the access graph. These solutions are generally edge-based. A node-based approach that solves the assignment one node at a time is feasible. This approach was tested for xformSOA with comparable results. Further research on this solution could reveal the benefits of this approach in other situations such as SOA with MR.
8 Conclusions

In this research we addressed issues concerning address optimizations in embedded processors. We look at the spectrum of ideas that address the same issues, primarily from Liao [55], Leupers [59], Hong [45], and Atri [7]. Embedded processor architectures considered include systems with one AR, with one AR and one MR, and systems with k ARs. In addition we propose an SOA solution using Static Single Assignment, a solution to Incrementally improve SOA/GOA, and another SOA solution using access graph nodes to determine next priority.

In chapter 3 we address the issue of SOA. We observed that most solutions start with an access sequence to find a solution. The access sequence plays a crucial role in the offset assignment solution. The basic block from which the offset assignment is derived may be transformed using commutative transformations. Atri et al., and Rao and Pande have proposed solutions employing commutative transformations, which are either limited or exhaustive. We propose a solution that classifies the edges to be used in SOA to assist in generating an offset assignment. The edges are classified as breakable and unbreakable edges. Unbreakable edges are preferred over breakable edges. Once an assignment is made, the breakable edges not chosen in the assignment are transformed to reduce the cost of the assignment. This heuristic iterates as long as the cost of the assignment demonstrates benefit in each iteration.

Generating offset assignment for embedded processors with a modify register requires a different approach. Common approaches find edges not covered by single offset assignment and select a value for MR that reduces the cost most. We introduce three new techniques in Chapter 4: edge folding, node swapping, and path interleaving.
These techniques bring variables closer together such that most transitions are covered with auto-increment or decrement, or with an AR modified by an MR value of two. Experimental results have shown reduction in cost in over 60% of cases. This technique can also be extended for other MR values such as MR = 3 or higher.

Higher end embedded processors may have more than one AR. With additional resources, the variables may be partitioned such that they are distributed across the ARs reducing the cost of assignment. A trivial case of using two variables per AR reduces any cost due to assignment to zero, but the initialization cost may be high. Some heuristics, such as the GOA_DEG heuristic use only the required number of variables. Additional ARs are used if they are available, if the cost of using additional ARs is less than the cost using 1 AR, or if the number of ARs used is less than half the number of variables. Our initial implementation chose the next variable that had the highest degree in the access graph. Variations include choosing variables with lowest degree and choosing one variable with highest degree followed by variables with lowest degree. All variations of the basic GOA_DEG heuristic result in better cost than Leupers’ or Gong’s GOA solutions.

In Chapters 6 and 7 we looked at other alternatives to address optimizations. In Chapter 6, The Static Single Assignment form of the basic block is explored. The SSA form has been used in optimizing compilers to optimize code. by extending that technique, we observe that the SSA form make a graph sparser, which usually reduces the cost of an assignment. The SSA form can also be used to commutatively transform. More importantly the SSA form allows address space reuse. a variable that has an earlier end-time than another variable’s start-time can be coalesced to use the same variable space.
We discuss a greedy technique that groups variables for space reuse. The resulting basic block can also be commutatively transformed.

In Chapter 7 we started with an offset assignment and improved the cost of the assignment. This technique can be applied to any offset assignment heuristic in the post processing phase. While generating an offset assignment an edge may not have been chosen for various reasons: tie-breaking rules, arbitrary selection when other rules do not resolve, etc. Such edges are incrementally tested in the given assignment to generate another assignment with lower cost. We permit the possibility of the assignments cost to be higher than the initial value in the intermediate steps.

We also introduced a new approach to offset assignment that is node-based. Almost all techniques are edge-based. This heuristic prioritizes nodes that are chosen using a nodes degree and the weight of edges incident at it. We tested this technique with commutative transformation SOA and found it to be effective.

Table 8.1 Heuristics Summary

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>1 AR</th>
<th>1MR</th>
<th>k AR, n MR</th>
<th>Cost Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>xformSOA</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SOA2MR</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GOA_DEG</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SOAwithSSA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>improved incremental</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SOAbyNode</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
In Table 8.1 we summarize the results from the research in tabular form. As noted in the table all the heuristics showed benefits in address optimizations in embedded processors. The xformSOA heuristic, SOA2MR and SOAwithSSA introduce concepts that use classifications and techniques. GOA heuristics based on degree show cost benefit over other GOA heuristics based on different access graph attributes.

8.1 Future Research

Commutative transformations exhibit significant improvement in cost of offset assignments. Instances where a zero-cost change can make an improvement in the next iteration have not been fully explored.

Assignment modification techniques: edge folding, node swapping, and path interleaving, have been explored for MR = 2. We believe that these techniques can be applied for MR values of 3 and higher. This assumption has not been fully explored nor its limitations tested.

Offset assignment with SSA currently has a greedy technique. After a group of variables is reassigned a common variable space, SOA techniques can be applied to the basic block. Grouping variables into a common variable space is similar to resource allocation problem. A non-greedy solution for resource allocation problems may have a greater advantage coalescing variables than the greedy approach.

Basic blocks and access sequences considered in this research can be parts of loops and these variables may be parts of arrays. An access sequence that is repeated needs additional offset assignment techniques or use of addressing modes that allow modulo operation for the optimal assignment to be repeated. If different elements of an array are used in every iteration then the access sequence is modified constantly with
very little change. Future changes to current assignment have to be considered in these instances.

We plan to use the ideas of network flow [31-38] to allocate variables to different memory banks, which might affect variable access patterns. We plan to consider applying the work in this research in these related areas.
References


[68] Motorola. DSP56k AGU, Section 4, Manual


[75] PERL. http://www.perl.org/


[83] Texas Instruments. TMS 320 Second Generation Devices. SPRS010B


Vita

Sai Pinnepalli is from Anantapur, India. He received his Bachelor of Engineering degree in electronics engineering from Bangalore University, in April 1987. He worked at Electronics Corporation of India Ltd. (ECIL) for over three years. He received his Master of Science degree in systems science from Louisiana State University in May 1995. He expects to receive his Doctor of Philosophy degree in computer science in May 2004.

Sai Pinnepalli is currently employed by Louisiana State University.