**University of Alberta**

**Library Release Form**

**Name of Author**: Johnny Huynh

**Title of Thesis**: Minimizing Address-Computation Overhead

**Degree**: Master of Science

**Year this Degree Granted**: 2006



Johnny Huynh
Department of Computing Science,
University of Alberta,
Edmonton, Alberta, Canada

**Date**: _____

*Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*

– Brian W. Kernighan

**University of Alberta**

# MINIMIZING ADDRESS-COMPUTATION OVERHEAD

by

**Johnny Huynh**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

in

Department of Computing Science

Edmonton, Alberta
Fall 2006

**University of Alberta**

**Faculty of Graduate Studies and Research**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Minimizing Address-Computation Overhead** submitted by Johnny Huynh in partial fulfillment of the requirements for the degree of **Master of Science**.

José Nelson Amaral

Vincent Gaudet

M.H. (Mike) MacGregor

**Date**: _____

*To my parents, for your never-ending support*

# Abstract

In many digital signal processors (DSPs), variables stored in memory are accessed using address registers and indirect addressing modes. The addressing code used to access these variables can have a significant impact on code size and performance. Thus, one optimization problem DSP compilers face is the problem of minimizing address-computation overhead. This thesis identifies three problems that must be addressed in order to minimize overhead: access-sequence generation, offset assignment, and address-code generation. Although these three problems have been extensively studied individually by other researchers, this work examines all three problems simultaneously to understand how each problem affects the overhead of the final code. Specifically, we propose a Minimum-Cost Flow (MCF) model to generate optimal addressing code for a *fixed* access sequence and memory layout. In order to minimize address-computation overhead, we must find an access sequence and memory layout that generates an MCF model that has minimum overhead.

By exhaustively evaluating the solution space of five small DSP benchmarks, the results of this thesis suggest that the access-sequence generated has very little impact on address-computation overhead, while the offset assignment of variables has a significant impact. We show that current offset-assignment heuristics proposed in the literature [15, 18, 23, 29] do not adequately address the offset assignment problem. In order for these algorithms to produce an offset assignment with minimal overhead, a new combinatorial problem, the *Memory Layout Permutation* problem, must be addressed. Alternatively, offset assignment algorithms can be designed to produce an MCF model that produces low overhead. This thesis presents and evaluates three such algorithms. We observe that each algorithm has an impractical running-time or does not consistently generate low-overhead memory layouts.

# Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. José Nelson Amaral, for making this thesis possible. His advice and guidance during my undergraduate and graduate studies have been invaluable in my academic and personal development.

Next, I would like to thank two colleagues, Dr. Sid-Ahmed-Ali Touati and Paul Berube, for their assistance and feedback throughout this research project. Their knowledge and insight of the problem models and DSP architecture were instrumental in the identification and formalization of the problems presented in this thesis.

Last, but certainly not least, I would like to give my deepest thanks and appreciation to my family. I am grateful that both, my brother, Andrew, and my fiancée, Jennifer, are computer scientists who understand the time and effort I needed to dedicate to this research. To my parents, Tho and Suong, I cannot express enough gratitude for their love and support through the years.

# Table of Contents

# List of Figures

# List of Tables

# List of Acronyms

**AIG** Augmented Interference Graph

**AR** Address Register

**ARA** Address-Register Assignment

**BFS** Best-First Search

**C-GOA** Conflict General Offset Assignment

**DDG** Data Dependence Graph

**DSP** Digital Signal Processor

**GOA** General Offset Assignment

**MCC** Minimum-Cost Circulation

**MCF** Minimum-Cost Flow

**MLP** Memory Layout Permutation

**MWPC** Minimum-Weight Path Cover

**MWPM** Minimum-Weight Perfect Matching

**SOA** Simple Offset Assignment

# Chapter 1

# Introduction

Digital signal processors (DSPs) are small, low-powered processors found in embedded systems, such as cell phones, portable music players, and cameras. DSPs are designed with a minimal number of functional units and instructions to keep processor size and power consumption minimal, while maintaining enough functionality to meet a specific application, such as processing audio and video data. A common design for these processors is an accumulator-based, register-memory architecture, where instructions can use data in memory without explicitly loading the data into a general purpose register. Instead of storing data in general-purpose registers, the addresses to data are stored in special registers called *address registers* (ARs) and accessed through several different addressing modes. The architectural advantages of address registers is that they only need to be large enough to hold addresses, and can be manipulated using simplified arithmetic units called address-generation units.

DSPs that are commonly used to process audio and video data must efficiently access large arrays of data in memory. To facilitate the memory accesses, post-increment and post-decrement indirect addressing modes are available in many DSPs. These addressing modes allow the processor to access a word in memory, specified by the address in an AR, and then modify the address in the AR by one word without additional instructions or processor cycles. When two consecutive memory accesses indexed by the same address register are to non-adjacent words in memory, an extra address-computation instruction is required. This extra instruction increases code size and introduces additional latency in the program. In this

thesis, *overhead* refers to the extra processor cycles required to perform address computations. Thus, the placement of data in memory, and the order in which the data is accessed, affects how often the post-increment or post-decrement addressing modes can be used.

Another architectural feature present in some DSPs to assist in address computations is a *modify register*. DSPs with modify registers can increment or decrement the address in an address register by the amount specified in a modify register without additional overhead. That is, an address register can be post-incremented (or post-decremented) without overhead by more than one word if the appropriate value exists in a modify register.

The overhead incurred by inefficient usage of address registers, modify registers, and addressing modes is called *address-computation overhead*. Specifically, there are two specific address-computation overhead — *initialization* and *jump*. Initialization overhead is incurred when an immediate value is initially loaded into an AR, usually requiring a double-word instruction. Jump overhead is incurred when an explicit address-computation is required for an AR to access two non-adjacent words in memory. The introduction of double-word instructions and additional single-word instructions has several undesirable consequences:

- code size increases;
- address-computations have non-zero latency, possibly reducing performance;
- more processor resources are used, increasing power consumption.

In this thesis, initialization and jump overheads are parameterized by INIT and JUMP cycles of latency, respectively.

## 1.1 Processor Model

The processor modeled in this thesis is based on the Texas Instruments `TMS320C54X` family of processors [24]. These DSPs have 8 16-bit address registers. Most instructions are 1 word in length and have 1 cycle of overhead. Initializing an address register requires a double-word instruction and has 2 cycles of latency. Instructions

may use a post-increment (or post-decrement) addressing mode to change the value of an address register by one word without additional overhead. Post-incrementing (or post-decrementing) an address register by more than one word requires an additional word to encode the explicit address-computation and results in an extra cycle of latency to execute the instruction. Thus, $INIT = 2$ cycles and $JUMP = 1$ cycle of overhead.

In the `TMS320C54X` family of processors, one address register $AR_0$ can be used as a modify register. This capability introduces two problems for deciding how to best use $AR_0$. If $AR_0$ is used as an address register, then all eight address registers can only be post-incremented (or post-decremented) by one word without overhead. If $AR_0$ is used as a modify register, then only seven address registers are available for indirect addressing, and the compiler must determine a modify value to store in $AR_0$. For this work, modify register optimizations are not considered, and $AR_0$ is only used as an address register in the processor model.

Similar to many other DSP architectures, the address registers in the `TMS320C54X` family of processors can also be used to store values other than addresses; however, the values stored in the address registers are subject to two limitations:

1. Address registers can only hold 16-bit values, while data in memory and the accumulator are 32-bit and 40-bit values respectively.

2. Address registers can only be manipulated by the address-generation unit, which is limited to addition and subtraction of 16-bit values.

Thus, address registers cannot be used as general purpose registers, and the problem of minimizing address computation overheads remains a relevant problem in relatively modern processors such as the `TMS320C54X` family of DSPs.

## 1.2 Methods to Reduce Overhead

Given a basic block of code, there are several optimization opportunities to reduce address-computation overhead: access-sequence generation, offset assignment, and address-code generation. Figure 1.1 illustrates the processes involved in minimizing

overhead. First, an access sequence must be generated to define the order in which memory locations are accessed. Next, offset assignment is performed in two phases: address register assignment (ARA) produces sub-sequences of accesses, and simple offset assignment (SOA) produces a memory layout for each sub-sequence. After a memory layout has been defined, the final addressing code can be generated.

Traditional approaches to minimizing overhead only focus on generating access sequences and memory sub-layouts. In previous research, the problem of permuting sub-layouts to form a single memory layout does not exist; and address-code generation is implicitly defined by address-register assignment. In this thesis, the offset assignment problem is extended to explicitly consider the memory layout permutation problem, and the problem of minimizing overhead is examined as a combination of three problems: offset assignment, access-sequence generation, and address-code generation.

The problem of minimizing overhead, and its associated sub-problems, are *optimization* problems; however, the problems can also be easily cast as *decision* problems. Thus, the complexity of the optimization problems discussed in this thesis will be classified using complexity classes such as P and NP [5].

### 1.2.1 Offset Assignment

Minimizing address-computation overhead is most often done by addressing the offset-assignment problem. Given a set of variables stored contiguously in memory, a *memory layout* is an ordering of these variables in memory. The order of variable accesses by the instructions in a basic block defines an *access sequence*. The *Offset-Assignment Problem* is defined as:

> Given $k$ address registers and a basic block accessing $n$ variables, find a *memory layout* that minimizes address-computation overhead.

Memory layouts with minimum overhead are called optimal memory layouts. This problem is called "offset assignment" because the address of each variable can be obtained by adding an *offset* to a common base address. If $k = 1$, then the problem is know as the *Simple Offset Assignment* (SOA). If $k > 1$ the problem is referred to

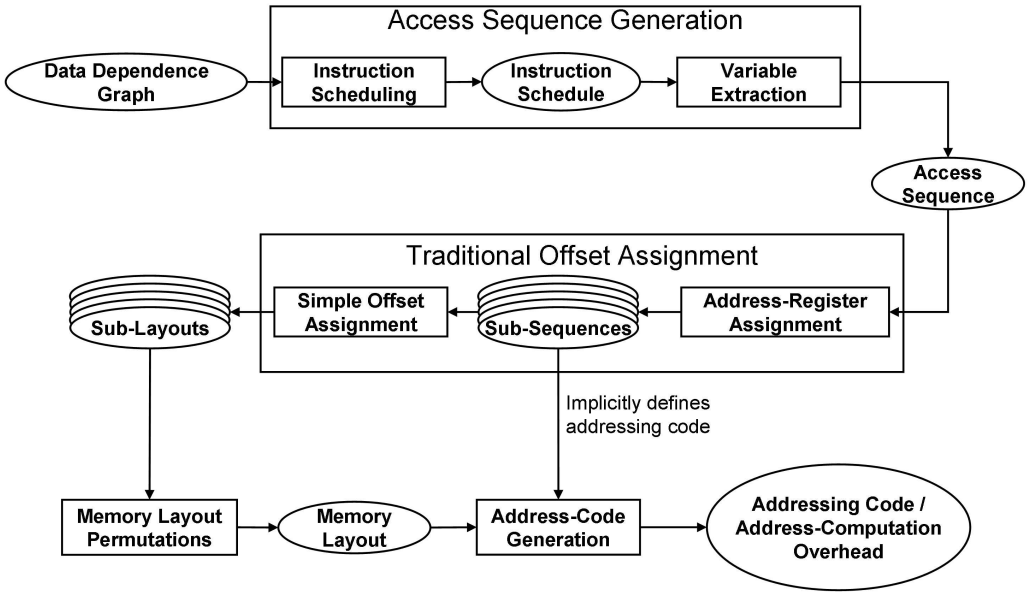



Figure 1.1: The processes involved in minimizing address-computation overhead. Traditionally, assumptions made during offset assignment implicitly define the final memory layout and addressing code.

as the *General Offset Assignment* (GOA).

In the Simple Offset-Assignment (SOA) problem, a single AR is available to access all the variables in the memory. Liao *et al.* convert the access sequence to an undirected *access graph* [18]. Variables are vertices in the graph, and edge weights indicate the number of times two variables are adjacent in the access sequence. Liao *et al.* show that the SOA problem can be reduced, in polynomial time, to the problem of finding a maximum-weight path cover in the access graph. Finding a maximum-weight path cover is an NP-complete problem; thus, the SOA problem is also NP-complete [5]. Approximation algorithms for the SOA problem are presented in Section 2.1.

In the General Offset-Assignment (GOA) problem, each *access* to one of the $n$ variables in an access sequence must be assigned to one of $k$ address registers. This assignment creates multiple access *sub-sequences* — one for each address register. A memory *sub-layout* can be found for each sub-sequence. Sub-layouts cannot be computed independently because a variable may appear in multiple address registers, but the union of all sub-layouts must still form a contiguous layout. Liao *et al.*

simplify the GOA problem by assigning *variables*, instead of *variable accesses*, to address registers [18]. This simplification produces sub-sequences that access disjoint sets of variables. A sub-layout can be obtained by solving the SOA problem for each sub-sequence, and the resulting set of sub-layouts can form a single memory layout. In this thesis, the problem of assigning *variables* to address registers is called the *Address-Register Assignment* (ARA) problem, and the problem of forming a single memory layout from a set of disjoint sub-layouts is called the *Memory Layout Permutations* (MLP) problem. Approximation algorithms for the ARA problem are discussed in Section 2.2. The MLP problem has not been addressed in the literature because each sub-layout is traditionally assumed to be independently accessed in memory. The formulation of the MLP problem is presented in Section 3.1.

### 1.2.2 Access-Sequence Generation

Since instance of the offset assignment problem is defined by the input access sequence, changing the access sequence can potentially change the final overhead of a given basic block of code. Generating an access sequence can be separated into two separate processes: instruction scheduling and variable extraction. Previously proposed algorithms that attempt to optimize the two processes are discussed in Section 2.3.

The first process involved in generating an access sequence is instruction scheduling. Let $G = (V, E)$ be a directed graph where each vertex in $V$ represents an instruction using three-address code. Each edge $(u, v) \in E$ represents a scheduling dependency where instruction $u$ must be scheduled before instruction $v$. Thus, $G$ is a *data dependence graph* (DDG) that defines a partial ordering of instructions, whereas a schedule $s$ is a fully-ordered list of instructions that also satisfies the partial ordering specified by $G$. Let $S$ be the set of all legal schedules of $G$. $GOA(s)$ represents the overhead of the optimal memory layout for a schedule $s \in S$.

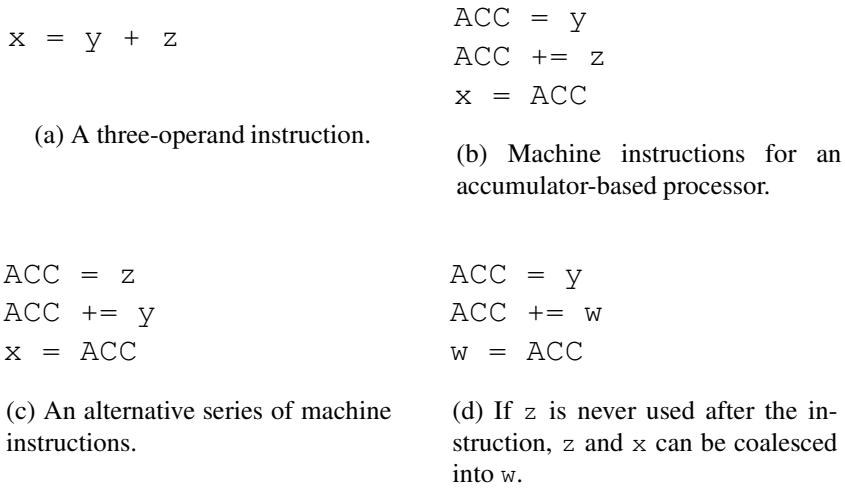
An instruction schedule for a basic block defines a sequence of three-address codes but does not necessarily define an access sequence. Variable extraction is required to identify the order in which variables in memory are accessed, thereby

defining an access sequence. Figure 1.2.2 shows how a single three-operand instruction can produce three different instruction sequences, and consequently, three different access sequences. Each variable in the instruction shown in Figure 1.2(a) represents a variable that must be stored, and accessed, in memory. In an accumulator-based machine, the instruction would typically be translated to a series of instructions as shown in Figure 1.2(b). The first source operand `y` is loaded, followed by a load of the second the source operand `z`. The newly computed result is then stored into the target operand `x`. Thus, the resulting access sequence is '`y z x`'. If the operator used in the instruction is commutative (such as addition), then the first two accesses in the sequence can be reversed so that the second source operand in the instruction is accessed before the first operand, as shown in Figure 1.2(c). Another optimization that can affect the access sequence is variable coalescing. Variable coalescing identifies a set of variables that can be assigned to a single memory location. In particular, if one variable is defined after the last use of another variable, the values of the two variables can safely be stored in the same memory location. For example, if `z` is never accessed after the instruction in Figure 1.2(a), then the variables `z` and `x` can be coalesced into one memory location, and represented by one variable `w`. The resultant machine instructions as shown in Figure 1.2(d), with an access sequence of '`y w w`'. Thus, variable coalescing changes the access sequence by reducing the number of unique variables accessed.

This thesis does not consider the optimization opportunities related to variable extraction. Thus, in the scope of this research, a schedule defines a unique access sequence. Additionally, the problem of generating an access sequence that ultimately results in a minimum overhead can then be defined. The problem of finding a *minimum-overhead access sequence* is to find a schedule $s \in S | GOA(s) \leq GOA(s'), \forall s' \in S$.

### 1.2.3 Address-Code Generation

After an instruction schedule and memory layout have been formed, addressing code must be generated. Addressing code dictates which variable *accesses* are to be addressed by each address register, and ultimately determines the address-




(a) A three-operand instruction.

(b) Machine instructions for an accumulator-based processor.

(c) An alternative series of machine instructions.

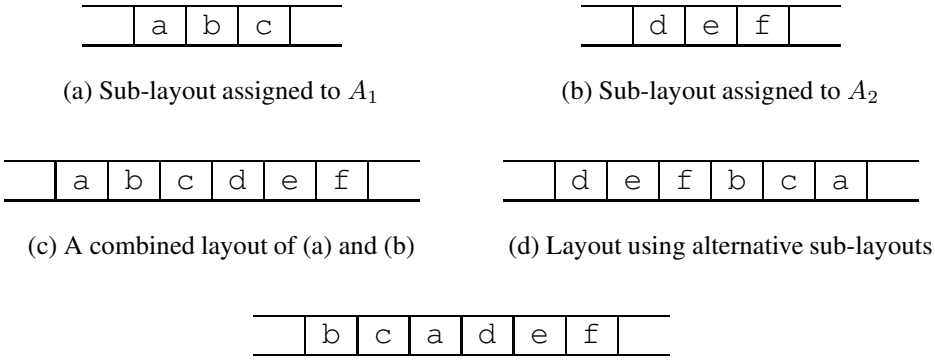
(d) If `z` is never used after the instruction, `z` and `x` can be coalesced into `w`.

Figure 1.2: A three-address instruction can be converted into three different series of machine instructions.

computation overhead for a basic block. Notice that the address-register assignment phase during offset assignment implicitly defines addressing code, and subsequently, the address-computation overhead value. However, address-register assignment can produce sub-optimal addressing code because it assigns *variables*, instead of accesses, to address registers. Instead of producing addressing-code based on address-register assignment, optimal address-code generation algorithms can be used. An assignment of accesses to address registers that produces the minimum overhead can be found in polynomial-time using the algorithms presented in Section 2.4. The existence of these algorithms creates a potentially large disparity between the overhead values modeled in traditional offset assignment and the actual overhead incurred by using optimal address-code generation.

## 1.3 Motivating Example

Using the processor model described in Section 1.1, consider a basic block of code that accesses 6 variables in memory. For illustrative purposes, assume the following fixed access sequence: '`a d b e c f b e c f a d`'. The traditional approach to minimize overhead is to perform offset assignment by partitioning the variables into disjoint sets. Each set is accessed exclusively by a single address reg-




(a) Sub-layout assigned to $A_1$

(b) Sub-layout assigned to $A_2$

(c) A combined layout of (a) and (b)

(d) Layout using alternative sub-layouts

(e) Swapping the sub-layouts

Figure 1.3: Several memory layouts.

ister. For example, variables {a,b,c} can be assigned to address register $A_1$, and variables {d,e,f} to address register $A_2$. The variables assigned to each address register can then be independently arranged in memory to form two independent *sub-layouts*, as shown in Figure 1.3. Address registers $A_1$ and $A_2$ independently access the variables, as shown in Figures 1.4(a) and 1.4(b). In this example, the address register assigned to each layout must each perform one initialization and one explicit address computation. Thus, the overhead for each address register is 3 cycles, for a total of 6 cycles of overhead.

In the traditional approach to the GOA problem, the sub-layouts in Figure 1.3 are considered "optimal" for two reasons. First, no ordering exists for either variable subset, {a,b,c} or {d,e,f}, with less than 3 cycles of overhead. Second, no partitioning of the 6 variables exists that can produce sub-layouts with a total overhead that is less than 6 cycles. Do these sub-layouts minimize the overhead of the input access sequence? No. The problem is the restriction that each set of variables be accessed by a single address register.

The memory layouts in Figures 1.3(a) and 1.3(b) can be placed contiguously in memory to form the single memory layout shown in Figure 1.3(c). Figure 1.5 shows that $A_1$ can be used for the last access of variable d (originally assigned to $A_2$) without requiring additional overhead. Similarly, $A_2$ is used for the last access of variable a (originally assigned to $A_1$). This solution has an overhead of 5 cycles instead of 6.

| variable accessed | offset | addressing code | overhead |
|---|---|---|---|
| | | $A_1$ = &a | 2 |
| a | 0 | $A_1$ += 1 | |
| b | 1 | $A_1$ += 1 | |
| c | 2 | $A_1$ -= 1 | |
| b | 1 | $A_1$ += 1 | |
| c | 2 | $A_1$ -= 2 | 1 |
| a | 0 | $A_1$ | |

(a) Instructions for address register $A_1$

| variable accessed | offset | addressing code | overhead |
|---|---|---|---|
| | | $A_2$ = &d | 2 |
| d | 0 | $A_2$ += 1 | |
| e | 1 | $A_2$ += 1 | |
| f | 2 | $A_2$ -= 1 | |
| e | 1 | $A_2$ += 1 | |
| f | 2 | $A_2$ -= 2 | 1 |
| d | 0 | $A_2$ | |

(b) Instructions for address register $A_2$

Figure 1.4: Instructions for address registers $A_1$ and $A_2$, using the layouts in Figures 1.3(a) and 1.3(b)

| variable accessed | offset | addressing code | overhead |
|---|---|---|---|
| | | $A_1$ = &a | 2 |
| | | $A_2$ = &d | 2 |
| a | 0 | $A_1$ += 1 | |
| d | 3 | $A_2$ += 1 | |
| b | 1 | $A_1$ += 1 | |
| e | 4 | $A_2$ += 1 | |
| c | 2 | $A_1$ -= 1 | |
| f | 5 | $A_2$ -= 1 | |
| b | 1 | $A_1$ += 1 | |
| e | 4 | $A_2$ += 1 | |
| c | 2 | $A_1$ += 1 | |
| f | 5 | $A_2$ -= 5 | 1 |
| a | 0 | $A_2$ | |
| d | 3 | $A_1$ | |

Figure 1.5: Accessing the variables in Layout 1.3(c) requires 5 cycles of overhead.

| variable accessed | offset | addressing code | overhead |
|---|---|---|---|
| | | $A_1$ = &a | 2 |
| | | $A_2$ = &b | 2 |
| a | 5 | $A_1$ -= 5 | 1 |
| d | 0 | $A_1$ += 1 | |
| b | 3 | $A_2$ += 1 | |
| e | 1 | $A_1$ += 1 | |
| c | 4 | $A_2$ -= 1 | |
| f | 2 | $A_1$ -= 1 | |
| b | 3 | $A_2$ += 1 | |
| e | 1 | $A_1$ += 1 | |
| c | 4 | $A_2$ += 1 | |
| f | 2 | $A_1$ | |
| a | 5 | $A_2$ -= 5 | 1 |
| d | 0 | $A_2$ | |

Figure 1.6: Accessing the variables in Layout 1.3(d) requires 6 cycles of overhead.

| variable accessed | offset | addressing code | overhead |
|---|---|---|---|
| | | $A_1$ = &b | 2 |
| | | $A_2$ = &a | 2 |
| a | 2 | $A_2$ += 1 | |
| d | 3 | $A_2$ += 1 | |
| b | 0 | $A_1$ += 1 | |
| e | 4 | $A_2$ += 1 | |
| c | 1 | $A_1$ -= 1 | |
| f | 5 | $A_2$ -= 1 | |
| b | 0 | $A_1$ += 1 | |
| e | 4 | $A_2$ += 1 | |
| c | 1 | $A_1$ += 1 | |
| f | 5 | $A_2$ | |
| a | 2 | $A_1$ += 1 | |
| d | 3 | $A_1$ | |

Figure 1.7: Layout 1.3(e) is optimal as the variables can be accessed with the minimum amount of overhead — 4 cycles.

Now, consider an alternative sub-layout to 1.3(a), with the variables ordered as {b,c,a}. Variables {d,e,f} are kept in the same order as shown in Figure 1.3(d). Similar to Figure 1.4, if variables {d,e,f} are assigned to $A_1$ and variables {b,c,a} are assigned to $A_2$, the total overhead is 6 cycles. If the two sub-layouts are placed contiguously in memory the minimum overhead is still 6 cycles, as shown in Figure 1.6.

However, if variables {b,c,a} are placed before variables {d,e,f}, producing the memory layout shown in Figure 1.3(e), and each variable can be accessed by more than one address register, then the address computation overhead is reduced to 4 cycles, as shown in Figure 1.7. Despite the similarities with memory layouts 1.3(c) and 1.3(d), the layout in Figure 1.3(e) is the only one that allows for the minimum amount of overhead.

Clearly, in order to minimize address-computation overhead for a basic block of code, the offset assignment problem must be solved. Unfortunately, solving the ARA and SOA problems alone is not sufficient to minimize overhead because of the restriction that each variable must be accessed by a single address register. Ultimately, address-computation overhead is dictated by the addressing code generated, which allows multiple address registers to access the same variable. Thus, if it is possible to generate optimal addressing code for a fixed access sequence and memory layout, the following questions are raised:

- Do different ARA and SOA heuristics affect the combined-layout overhead?
- How should sub-layouts be arranged to minimize address-computation costs?
- How do different access sequences (found through instruction scheduling) affect overhead?
- What is the minimum amount of address-computation overhead required for a given basic block of code?

This thesis addresses these questions through an exhaustive exploration of the solution space.

## 1.4 Contributions

The rest of this thesis is organized as follows. Previously proposed algorithms that affect address-computation overhead are presented in Chapter 2. The problems with previously proposed techniques are discussed in Chapter 3. Specifically, three new contributions are presented:

- the formulation of a new optimization problem, the Memory Layout Permutations problem, that has to be solved in order to minimize overhead when generating memory layouts using the traditional approach;
- an extension of the offset assignment problem, called the Conflict General Offset Assignment problem, that reflects the changes in address registers and memory in modern DSP architectures;
- a minimum-cost flow (MCF) model for the generation of addressing code for a fixed access sequence and memory layout. Let $L$ be the number of accesses and $N$ be the number of variables. This MCF model has at least $\frac{L}{N}$ fewer edges than Gebotys' minimum-cost circulation (MCC) model, as demonstrated in Section 3.2.

The rest of the thesis presents several experimental results. Chapter 4 examines the solution space for access-sequence generation and offset assignment. Examining the solution spaces demonstrate that optimal address-code generation alone is not sufficient to minimize address-computation overhead. Furthermore, the offset assignment has significant impact on overhead, while access-sequence generation does not. Chapter 5 presents an empirical evaluation of heuristic-based algorithms published in the literature for offset assignment and demonstrates that these algorithms produce poor approximations to the minimization of address-computation overhead. Alternative algorithms for offset assignment are presented and evaluated in Chapter 6. Lastly, conclusions on the study and future work are discussed in Chapter 7.

# Chapter 2

# Related Work

This chapter presents the different approaches and algorithms that can affect address-computation overhead. As mentioned in Section 1.2, overhead is reduced by addressing three issues: offset assignment, access-sequence generation, and address-code generation. Algorithms designed to approximate solutions for the offset assignment problem are presented in Sections 2.1 and 2.2. Section 2.3 presents research that affect how access sequences are generated, and Section 2.4 presents two algorithms to generate optimal addressing-code for a fixed offset assignment and access sequence.

## 2.1 Simple Offset Assignment

Bartley introduced the SOA problem in 1992 and solved it as a *maximum-weight Hamiltonian-path* problem [2]. A path in the access graph represents the ordering of variables in memory. Liao *et al.* refine the problem formulation to a *maximum-weight path-cover* problem (MWPC) [18]. They improve the run-time complexity of Bartley's algorithm to approximate a solution to the SOA problem. This improved algorithm begins by marking all edges of the access graph $G$ as *removable* and sorts them in decreasing order of weight. The algorithm then iterates the list of removable edges and marks each edge as *unremovable* or removes the edge from $G$. In each iteration of the algorithm, the heaviest removable edge $e$ is marked as *unremovable*. If the other unremovable edges in $G$ form a cycle with $e$, then $e$ is removed from $G$. If $e$ is incident to two other unremovable edges, then all *remov-*

*able* edges incident to $e$ are removed. The algorithm terminates when all edges have either been removed or marked as unremovable. The unremovable edges form an approximate maximum-weight path cover.

Leupers proposes to extend Liao's algorithm by using a tie-break function to decide between edges that have equal weights [16]. This function computes the sum of the weights of incident edges to the edge being evaluated. When a tie occurs, the edge with the lowest sum is selected. Using this tie-break function usually increases the weight of path cover, resulting in memory layouts with lower address-computation overhead [16, 13].

Sugino *et al.* propose to approximate a MWPC by iteratively removing edges from the access graph using a greedy heuristic [23]. Each edge $e = (u, v)$ in the access graph is evaluated using two metrics, the *fork* value of the endpoints $u, v$, and the $cycle$ value of the edge $e$. The $fork$ value of a vertex $v$ is defined as:

$$fork(v) = \max\{degree(v) - 2, 0\}$$

The $cycle$ value of an edge $e$ is defined as:

$$cycle(e) = \begin{cases} 1 & \text{if } e \text{ is part of a cycle} \\ 0 & \text{otherwise} \end{cases}$$

Using these two metrics, the *benefit* of each edge $e = (u, v)$ access graph is defined as:

$$benefit(e) = \frac{fork(u) + fork(v) + cycle(e)}{weight(e)}$$

In each iteration of the algorithm, the edge with the highest benefit is removed. The benefit of each edge is re-evaluated and the process continues until the access graph becomes a path.

For evaluation purposes (see Chapter 5), a naive and an optimal algorithm to find a memory layout for the SOA problem are also implemented. The naive algorithm produces a memory layout based on the declaration order of variables. Two variables, $u$ and $v$, are adjacent in memory if and only if there are no other variables that are declared between the declaration of $u$ and the declaration of $v$. The algorithm is known as the Order First Use (OFU) algorithm [13, 18]

Liao *et al.* propose an algorithm that finds an optimal layout for the SOA problem using the branch-and-bound technique [17]. The algorithm has an exponential running time but can compute the MWPC for graphs with 12 vertices in a reasonable amount of time. The algorithm is based on the observation that an access graph with $n$ variables has $n-1$ edges in a maximum-weight path cover. Given a partial path cover $p$ with $m < n-1$ edges, there is a set of *valid edges* that can be added to $p$. An edge $e$ is valid if adding $e$ to $p$ does not form a cycle in $p$ and does not cause a vertex in $p$ to have a degree greater than two. Let $p'$ be the partial cover $p$ augmented with $e$. An upper bound on a path cover subsuming $p'$ is the weight of $p'$ plus the $n-m$ heaviest valid edges. If the upper bound of $p'$ is greater than the current maximum weight path cover, the procedure is recursively called. Otherwise, $p'$ is discarded and another valid edge is added to $p$. When there are no more valid edges to add to $p$, the MWPC is found, producing an optimal memory layout for the SOA problem.

## 2.2 Address-Register Assignment

In the GOA problem, $k > 1$ address registers are used to access variables in memory. Liao *et al.* decompose the GOA problem into multiple instances of the SOA problem by assigning each variable to an address register $A_i$. Let $C(A_i)$ be the address-computation overhead for an optimal SOA solution to variables assigned to $A_i$. Liao *et al.* define the GOA problem as follows:

> Given an access sequence $S$, the set of variables $V$, and $k$ address registers, assign each $v \in V$ to an address register $A_i, 1 \leq i \leq k$, such that $\sum_{i=1}^{k} C(A_i)$ is minimum.

Solving this problem does not produce a memory layout — it only produces an assignment of variables to address registers. Additionally, as shown in the example in Section 1.3, assigning variables to address registers may not necessarily minimize the overall address-computation overhead. Thus, this problem should not be considered as the *real* GOA problem. In this thesis, the problem is called the Address-Register Assignment (ARA) problem. Finding an optimal solution to the

ARA problem is NP-hard because the SOA problem, which is NP-complete, is an instance of the ARA problem [5].

Three algorithms that approximate a solution to the ARA problem are examined in this thesis. In order to approximate the minimum overhead, an approximation of $C(A_i)$ is required. Any one of the SOA algorithms in Section 2.1 can be used as a sub-routine for the following ARA algorithms to approximate the overhead of assigning a variable to an address register.

Leupers and David propose to solve the ARA problem by using a greedy algorithm based on selecting edges [16]. Given an access graph $G$, the algorithm assigns the $k$ heaviest disjoint edges of $G$ to each address register. Each remaining variable $v \in V$ is assigned to the address register $A_i$ for which $v$ causes the minimum increase to $C(A_i)$.

Sugino *et al.* use an heuristic-based algorithm for the ARA problem [23]. Their algorithm requires two disjoint partitions of all the variables. They claim that starting with one partition with all variables and one partition with no variables works best. Each variable is initialized as *not-yet-moved*. The algorithm moves one variable at a time from one partition to the other. The *gain* of moving a variable from $A_i$ to $A_j$ is the reduction of $C(A_i) + C(A_j)$. At each iteration, the algorithm moves the not-yet-moved variable that yields the highest gain. The algorithm saves all the intermediate partitions and their costs. When all the variables have been moved, the intermediate partition configuration that has the lowest total cost is selected. If there are $k > 2$ address registers available, the procedure is repeated on each pair of address registers until no movement occurs.

Zhuang *et al.* propose a technique to simplify offset assignment problems using variable coalescing [29]. They propose an algorithm to assign variables to address registers that is independent of the variable coalescing technique. The algorithm assigns a single variable to a single address register at a time. Each unassigned variable $v \in V$ is added to each address register $A_i$ and the increase in $C(A_i)$ is computed. The assignment that results in the lowest increase is committed. If there is a tie, a weighted access graph $G$ is used. Let $weight(v, u)$ be the weight of the edge connecting $v$ and $u$ in $G$. Let ($v$,$A_i$) be an assignment of $v$ to $A_i$. For each

$(v,A_i)$ that is tied, the following score is computed:

$$w1(v, A_i) = \sum_{u \in A_j, j \neq i} weight(v, u)$$

The assignment with the maximum $w1$ score is selected. This process continues until all variables are assigned to an address register.

Although many approaches to the GOA problem address the ARA and SOA problems separately, some researchers have proposed algorithms that address both problems simultaneously using iterative approaches. Atri *et al.* propose an SOA algorithm that iteratively improves a given memory layout [1]. Similarly, Wess and Zeitlhofer propose to approximate a solution to the GOA problem by iteratively modifying offset assignments and address register assignments [26]. Leupers and David propose to find a memory layout for the GOA problem using a genetic algorithm [15], while Wess and Gotschlich propose to generate memory layouts using simulated annealing [15, 27]. These algorithms are excluded from the experiments in Chapter 5 because their performance is dependent on the initial memory layout produced. Additionally, simulation-based algorithms can be run for an unspecified number of iterations, making it difficult to compare to heuristic-based algorithms.

Despite the large number of algorithms proposed by researchers to generate an offset assignment and address-register assignment, only two comprehensive comparisons of the algorithms exist. Leupers presents the *OffsetStone* benchmark suite and application to evaluate algorithms for the SOA problem [13]. Huynh *et al.* use a minimum-cost circulation technique (discussed in Section 2.4) to accurately evaluate a combination of SOA and ARA heuristic-based algorithms [8].

## 2.3 Generating Access Sequences

As mentioned in Section 1.2, instruction scheduling and variable extraction can affect the access sequence used in the offset assignment problem. Many algorithms have been proposed to improve overhead through instruction scheduling or re-ordering variable accesses. Rao and Pande apply algebraic transformations (such as commutativity) on expression trees to produce a *least-cost access sequence* [22]; Lim *et al.* manipulate the entire instruction schedule [19]. Kandemir *et al.* propose

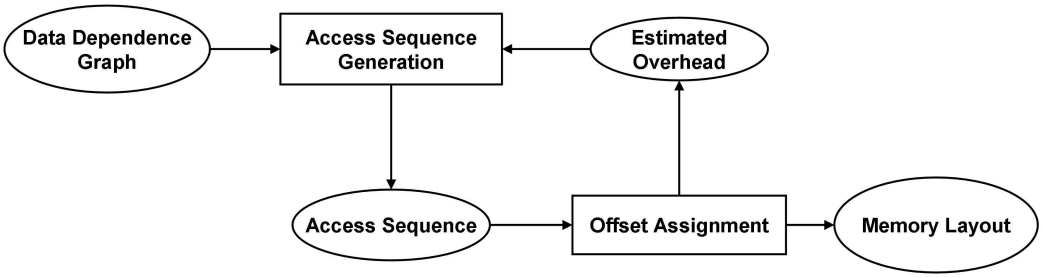



Figure 2.1: The relationship between access-sequence generation and offset assignment.

an algorithm to change the access sequence *after* a memory layout is formed for each basic block [9]. Unfortunately, these algorithms share two major drawbacks. First, the objective of the algorithms is to produce an access sequence that can be accessed with minimal cost by a *single* address register. That is, the algorithms are primarily designed to produce access sequences for the SOA problem. Second, the algorithms rely on an effective offset assignment algorithm to generate the access sequence, as shown in Figure 2.1. An offset assignment algorithm is used to determine the overhead of potential access sequences. Once an access sequence is generated, the offset assignment algorithm is invoked again to produce a memory layout. A unified algorithm for scheduling and offset assignment by Choi and Kim avoids both drawbacks. Choi and Kim observe that small changes in the instruction schedule produces localized changes in the access graph. Thus, it is possible to incrementally compute the overhead associated with each change in the access sequence. The main drawback to this technique is its iterative nature, which may not always be a feasible approach in a compiler, especially if each iteration is computationally intensive.

After an instruction schedule is found, there is another opportunity to decrease address-computation overhead through variable coalescing. Ottoni *et al.* coalesce variables and simultaneously find a memory layout for instances of the SOA problem [21]. Similarly, Zhuang *et al.* propose algorithms that coalesce variables for both SOA and GOA problems [29]. Although the coalescing algorithms in both works simultaneously find memory layouts, it is still possible to perform an addi-

tional offset assignment pass to further reduce address-computation overhead.

The majority of studies related to minimizing address-computation overhead have traditionally focused on offset assignment for *scalar* variables in *straight line code*. Few studies treat *array* accesses in *loops*. Leupers and David propose both a branch-and-bound and heuristic-based algorithm to improve AR usage for array accesses in loop bodies [14]. Cheng and Lin propose a graph-based address register allocation and data re-ordering algorithm to reduce overhead for loop execution [4]. Chen and Kandemir present a scheme to transform arrays and reschedule array accesses to reduce overhead using minimum cost traversals of reference graphs [3]. Zhang and Yang propose a procedure-level method of offset assignment that uses an access graph with information from a control-flow graph [28].

## 2.4 Address Code Generation

After an instruction schedule and a memory layout have been found, the final addressing code must be generated. Traditionally, the address-register assignment produced during offset assignment defines the addressing code. That is, if each variable in an access sequence has already been assigned an address register to access it, then no additional work is required. However, address-register assignment during offset assignment only assigns *variables* to address registers. As depicted in Figure 1.1 and demonstrated in Section 1.3, the address-computation overhead for a basic block of code is ultimately determined by the addressing code generated. Specifically, overhead for a given access sequence and memory layout is minimized by assigning *variable accesses* to address registers. There are two known techniques to generate optimal addressing code: minimum-cost circulation and minimum-weight perfect matching.

### 2.4.1 Minimum-Cost Circulation

An algorithm to find the optimal addressing code is proposed by Gebotys [6]. Gebotys shows that the assignment of *accesses* to address registers can be found in polynomial time by transforming $S$ and $M$ into a directed cyclic network-flow graph.

The minimum cost circulation (MCC) of the graph represents the optimal addressing code, and the cost of the circulation represents the minimum overhead for the given memory layout. The MCC for a fixed memory layout can be computed using integer linear programming where the constraint matrix is totally unimodular, and thus, can be solved in polynomial time. All memory layouts in this paper evaluate the quality of a memory layout using this technique. Gebotys' MCC technique is reproduced here for the reader's convenience.

Let $G = (V, E)$ be a network-flow graph with vertices $V$ and edges $E$. $V$ is composed of the accesses $a_i \in S$, and two special vertices, source $a_s$ and sink $a_k$. Let $(a_i, a_j)$ represent an access to $a_i$, immediately followed by an access to $a_j$. $E$ is composed of directed edges $(a_i, a_j)$, for all $a_i$ that are accessed before $a_j$. The cost, $c_{i,j}$, associated with each edge, $(a_i, a_j)$, represents the overhead for a single address register to consecutively access $a_i$ then $a_j$. $E$ also contains special edges that connect the source and sink vertices, $(a_s, a_i)$ and $(a_i, a_t), \forall a_i \in S$. These edges do not represent actual accesses in $S$, thus their cost is zero. $E$ also has a special edge connecting the sink to the source, $(a_t, a_s)$. Each unit-flow through this edge represents an address register initialization, thus its cost is $c_{t,s} = INIT$.

To find the minimum cost circulation of $G$, a set of linear constraints are placed on the flow through each edge in $E$. Let $e_{i,j}$ represent the amount of flow through edge $(a_i, a_j) \in E$.

Since $(a_t, a_s)$ represents address register initialization, the flow through this edge cannot exceed the number of available address registers, $r$:

$$0 \leq e_{t,s} \leq r$$

All other edges represent an access by a single address register, thus the flow through these edges must be non-negative and not greater than 1.

$$0 \leq e_{i,j} \leq 1, i \neq k, j \neq s$$

The minimum cost circulation must also satisfy the *conservation of flow* property [11], thus the total flow into a vertex must equal the total flow out of the vertex:

$$\sum_j e_{i,j} - \sum e_{j,k} = 0$$

Finally, the model must ensure that each access, $a_j \in S$, is executed exactly once. This condition can be expressed by adding a constraint on each directed edge ending at $a_j$.

$$\text{for each j} \neq \text{s,k}, \sum_{i \neq j} e_{i,j} = 1$$

Thus, the minimum cost circulation of $G$ is found by minimizing the total cost of the flows:

$$z = \sum_{e_{i,j} \in E} c_{i,j} e_{i,j}$$

subject to the constraints described above.

Since variables $e_{i,j}$ have non-negative integer bounds, the flows through the graph are guaranteed to be integers because [6, 11]. Thus, the MCC of a given memory layout can be solved in polynomial time with a linear programming library or solver.

### 2.4.2 Minimum-Weight Perfect Matching

An alternative technique to generate optimal addressing code is to use the minimum-weight perfect matching (MWPM) technique proposed by Udayanarayanan [25]. Given an access sequence $S$ and a memory layout $M$, the MWPM technique builds a bipartite graph as follows:

Let $G = (V, E)$ be a bipartite graph with vertices $V$ and edges $E$. $V$ is composed of pairs of vertices, $x_i$ and $y_i$, representing each access $a_i \in S$. $V$ also contains pairs of vertices, $s_r$ and $t_r$, $1 \leq r \leq k$, representing the $k$ address registers. $E$ is composed of four types of undirected edges:

- $(y_i, x_j)$, $i < j$. These edges model an AR accessing $a_i$ followed by $a_j$, thus the edge cost is $c_{i,j}$.
- $(y_i, s_r)$. These edges model an AR accessing $a_i$ first.
- $(x_i, t_r)$. These edges model an AR accessing $a_i$ last.
- $(s_r, t_r)$. These edges model unused ARs in addressing code.

The MWPM of $G$ represents addressing code, for the given $S$ and $M$, with minimum overhead. The advantage to using the minimum-weight perfect matching technique is a faster running time. Let $l$ be the number of accesses in $S$. The complexity of finding a minimum-weight perfect matching is $O((l + k)^3)$, while the running time to find a minimum-cost circulation is $O(l^4\ log\ l)$ [25]. However, since $k$ is small and bounded, the MWPM can be found in less time.

# Chapter 3

# Extending Current Models

Chapter 2 clearly shows that there has been a significant amount of research on problems that involve reducing address-computation overhead; however, there are still more opportunities to further reduce overhead in DSP codes and to do so using more precise and more efficient models and algorithms. This chapter presents three improvements in current problem models in order to more accurately model the problem of minimizing address-computation overhead. Specifically, Section 3.1 presents a new problem called the *memory-layout permutation* problem that arises from the separation of offset assignment and address-code generation. In Section 3.2, an alternative model for address-code generation is proposed. The new model is based on the minimum-cost flow technique presented in Section 2.4. This new model is more efficient and easier to implement. Lastly, in Section 3.3, a new variant of the offset assignment problem is introduced to reflect advancements in modern DSP architectures.

## 3.1 Memory-Layout Permutations

Figure 3.1 illustrates the traditional process of generating a memory layout. ARA produces a set of disjoint access sub-sequences that are solved as independent SOA problems. Solving each SOA instance produces a memory layout called an ARA sub-layout. Each ARA sub-layout is accessed independently through a single address register, thus, each sub-layout can be placed independently in memory. Additionally, each ARA sub-layout is a set of disjoint paths that cover an access graph

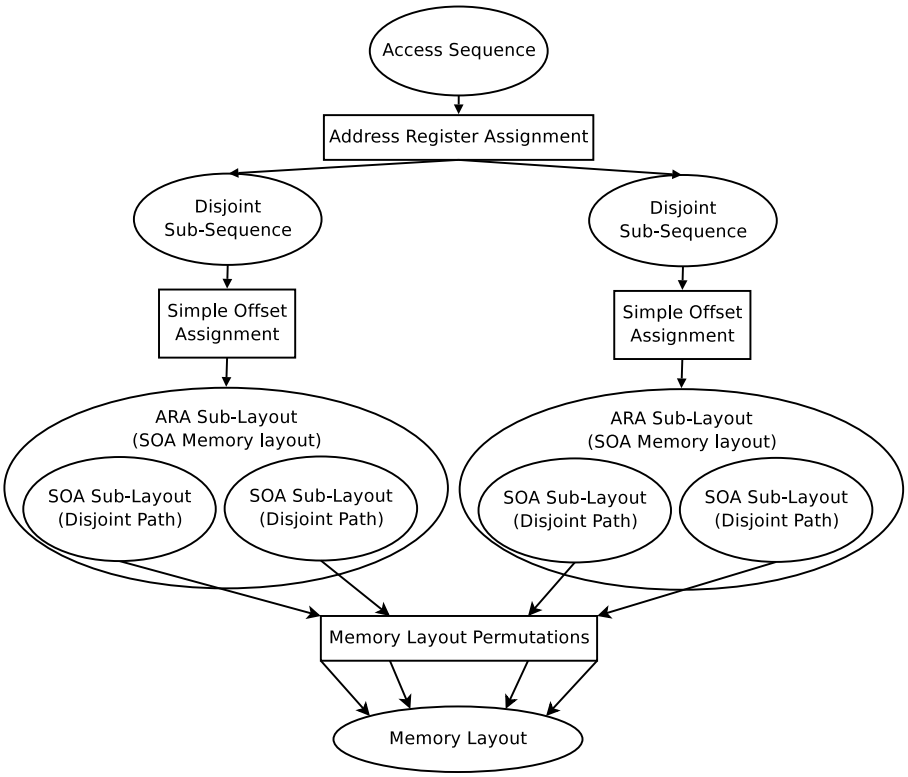



Figure 3.1: Performing address register assignment followed by simple offset assignment generates memory sub-layouts that must be placed in memory. The problem of finding a placement that minimizes overhead is called the memory layout permutations problem.

for an SOA problem. Each disjoint path is called an SOA sub-layout and defines a contiguous ordering of variables in memory. Since the SOA sub-layouts are disjoint and accessed by a single address register, it is not possible to use a post-increment (or post-decrement) addressing mode to consecutively access two variables in two different sub-layouts. Thus, SOA sub-layouts can also be placed independently in memory. Unless otherwise stated, the term *sub-layout* refers to an SOA sub-layout.

However, Section 1.3 demonstrates that if a variable can be accessed by multiple ARs, address-computation overhead may be reduced by placing the sub-layouts contiguously in memory. Since the MCC technique allows variables to be accessed by multiple ARs, the sub-layouts should no longer be placed independently in memory. Let $M_i$ be a sub-layout and $M_i^r$ be a sub-layout with the variables of $M_i$ in reverse order in memory. Let $(M_i|M_i^r)$ stand for an instance of either $M_i$ or $M_i^r$. The *memory layout permutations* (MLP) problem is introduced as follows:

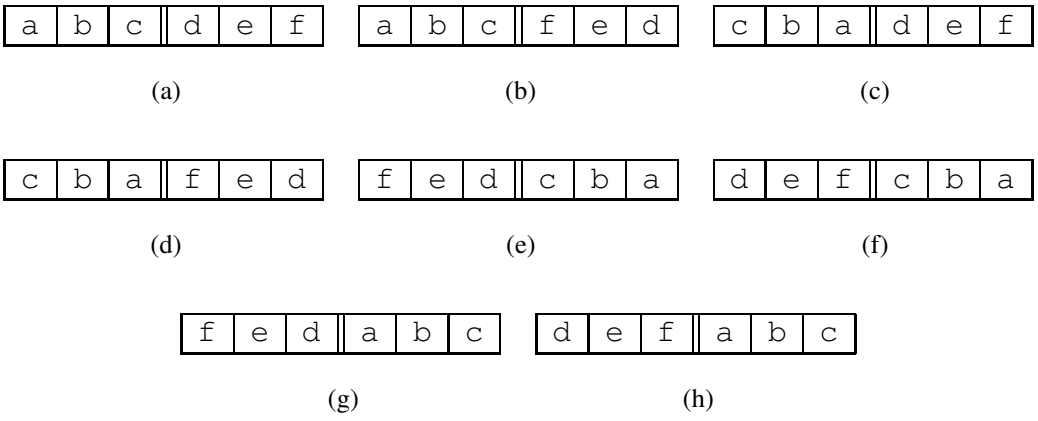



Figure 3.2: Permutations of two sub-layouts

> Given an access sequence $S$ and a set of $m$ disjoint memory sub-layouts, find an ordering of the sub-layouts $\{(M_1|M_1^r), \ldots, (M_m|M_m^r)\}$ such that address-computation overhead is minimum when the sub-layouts are placed contiguously in memory.

The MLP solution space is extremely large. Given $m$ sub-layouts, there are $m!$ permutations of sub-layouts. For each permutation, each sub-layout can be placed in memory as either $M_i$ or $M_i^r$. Thus, $m$ sub-layouts originate $(m!)(2^m)$ layouts. However, because all variables have the same relative offset to each other, an ordering of layouts $M_1, \ldots, M_m$ is equivalent to its reciprocal layout, $M_m^r, \ldots, M_1^r$, Thus, the MLP solution space is $\frac{(m!)(2^m)}{2}$ memory layouts. Figure 3.2 shows how 2 sub-layouts can form 8 possible layouts, half of which are reciprocals of another.

When reciprocals are considered, an offset assignment problem with $n$ variables has a solution space of $\frac{n!}{2}$ memory layouts. In the degenerate case, each sub-layout can be a single variable, $m = n$, and the MLP problem is reduced to the offset assignment problem. This reduction implies that if an algorithm solves the MLP problem, the same algorithm solves the offset assignment problem.

## 3.2 Minimum-Cost Flow

Although the MCC and MWPM techniques (see Section 2.4) can generate optimal addressing code in polynomial time, it is possible to further reduce the running time

to find optimal addressing code. This thesis proposes to use a minimum-cost flow (MCF) model to generate optimal addressing code. The MCF model has two practical advantages over the MCC and MWPM techniques. First, the MCF problem is a well-known problem with many algorithms and solvers implemented, such as `LEDA` [20]. The experiments performed in this thesis use Goldberg's efficient implementation of a scaling algorithm [7]. Second, the proposed MCF model uses a smaller network flow graph than the MCC and MWPM models, further reducing the execution time requirements for generating optimal addressing code.

The network-flow graph used in the MCF technique is very similar to the graph used by Gebotys, and can be constructed as follows. Let $G = (V, E)$ be a network-flow graph with vertices $V$ and edges $E$. $V$ is composed of the accesses $a_i \in S$, and two special vertices, source $a_s$ and sink $a_t$. $a_s$ has an excess supply equal to the number of address registers, and $a_t$ has a demand equal to the number of address registers. All other vertices in $V$ have a minimum and maximum flow capacity of 1. That is, each vertex must have one unit of flow through it. Let $(a_i, a_j)$ represent an access to $a_i$, immediately followed by an access to $a_j$. $E$ is composed of directed edges $(a_i, a_j)$, for all accesses $a_i \in S$ that are accessed before access $a_j \in S$. The cost, $c_{i,j}$, associated with each edge, $(a_i, a_j)$, represents the overhead for a single address register to consecutively access $a_i$ then $a_j$. $E$ also contains edges that connect the source and sink vertices, $(a_s, a_i)$ and $(a_i, a_t), \forall a_i \in S$. The edges $(a_s, a_i), \forall a_i \in S$ represent an address register initialization, and have cost $c_{s,i} = INIT$. Edges connected to the sink $a_t$ have cost $c_{i,t} = 0$. The final edge $(a_s, a_t)$ in $E$ connects the source to the sink and represents unused address-registers. It has cost $c_{s,t} = 0$ and infinite capacity. All other edges in $E$ have a capacity of 1. Each unit flow through $G$ represents an assignment of a single address register to a set of variable accesses; thus, a minimum-cost flow through $G$ represents an assignment of address registers to variable accesses with minimal overhead. In other words, the minimum-cost flow through $G$ represents an optimal addressing code for the given access sequence $S$ and memory layout $M$. An example of a minimum-cost flow graph for the example in Section 1.3 is shown in Figure 3.3. The network flow graph is built using the following access sequence

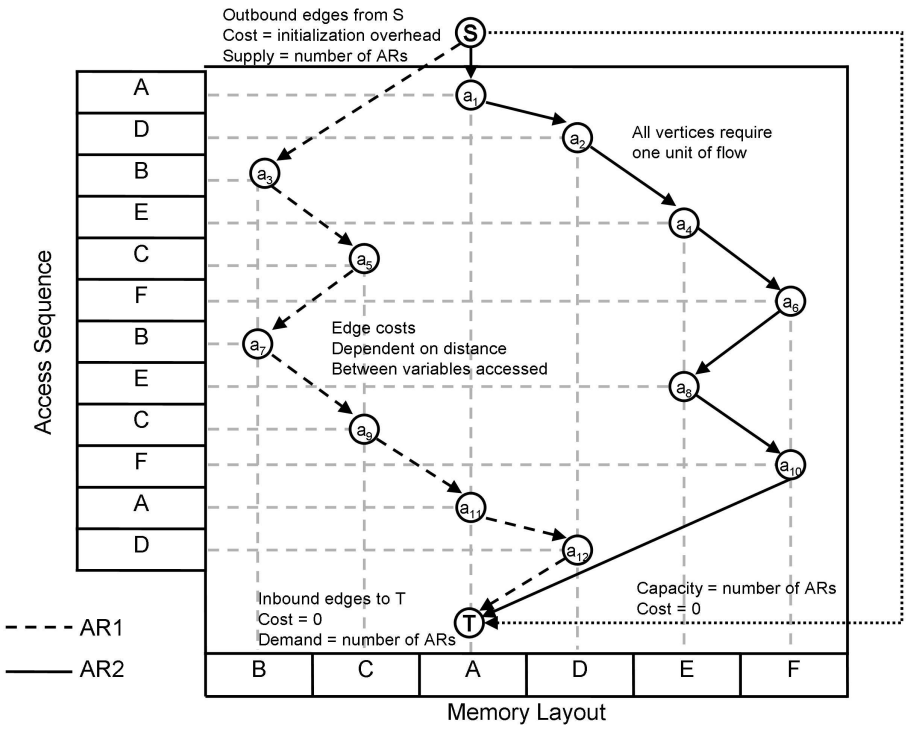



Figure 3.3: A minimum-cost flow representing the optimal addressing code for a fixed access sequence and memory layout. Only edges part of the minimum-cost flow are shown.

‘a d b e c f b e c f a d’

and memory layout

[b c a d e f]

The network flow graph $G$ models *all* possible addressing codes for an input sequence $S$ and layout $M$. However, to produce optimal addressing code, only *one* minimum-cost flow in $G$ is required. Thus, this thesis proposes to reduce the size of the network flow graph, and subsequently improve the time required to produce optimal addressing code, by identifying and removing edges in $G$ that are not necessary to find a minimum-cost flow that flows through all variables. In particular, an edge $(a_i, a_k)$ in $G$ is classified as *redundant* if and only if it has the following properties:

- there exists an access $a_j$ that occurs between $a_i$ and $a_k$ in the access sequence, and forms edges $(a_i, a_j)$ and $(a_j, a_k)$ in $G$;
- $a_i, a_j$, and $a_k$ do not all access the same variable in memory;
- either, $a_i$ and $a_j$, or $a_j$ and $a_k$, are accesses to the same variable in memory.

Examples of redundant edges are shown in Figure 3.4.

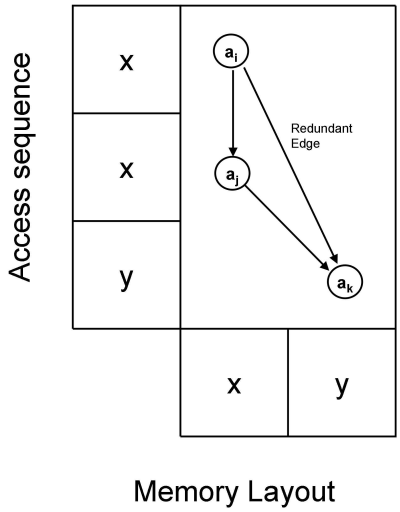
Let $MCF(G)$ represent the cost of the minimum-cost flow in $G$. Let graph $G'$ be $G-(a_i, a_k)$. This thesis shows that for any redundant edge $(a_i, a_k)$, $MCF(G') = MCF(G)$. In other words, a redundant edge in $G$ is an edge that can be removed without affecting the cost of the minimum-cost flow in $G$. Observe that $G'$ is a subgraph of $G$, and contains the same vertices and flow requirements as $G$. Any minimum-cost flow in $G'$ must also be a flow in $G$. Thus, it must always be the case that $MCF(G') \geq MCF(G)$. Therefore, to show that $MCF(G') = MCF(G)$, it is sufficient to show that $MCF(G') \leq MCF(G)$.

A redundant edge $(a_i, a_k)$ in $G$ can appear in a minimum-cost flow of $G$ under three different situations: thus, three cases must be examined:

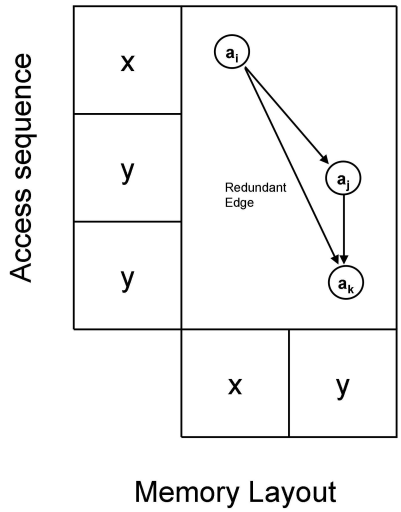
1. $(a_i, a_k)$ is not used in any minimum-cost flows of $G$;
2. $(a_i, a_k)$ is used in a minimum-cost flow flow of $G$ and has a cost $c_{i,k} = JUMP$;
3. $(a_i, a_k)$ is used in a minimum-cost flow flow of $G$ and has a cost $c_{i,k} = 0$.

When $(a_i, a_k)$ is not a part of any minimum-cost flows, it is trivial to demonstrate that $MCF(G') = MCF(G)$. Figures 3.5 and 3.6 illustrate the cases where $(a_i, a_k)$ is used in a minimum-cost flow and has cost $c_{i,k} = JUMP$, and $c_{i,k} = 0$, respectively. The figures illustrate that for every minimum-cost flow that uses a redundant edge, there exists another flow, with equal cost, that does not use a redundant edge; thus, the redundant edge can be removed from $G$.

First, consider the case where $(a_i, a_k)$ has cost $c_{i,k} = JUMP$, as shown in Figure 3.5. If $(a_i, a_k)$ is used in the minimum-cost flow, then there must exist a second flow to access $a_j$. Let $a_h$ be an access that occurs before $a_j$, and let $(a_h, a_j)$

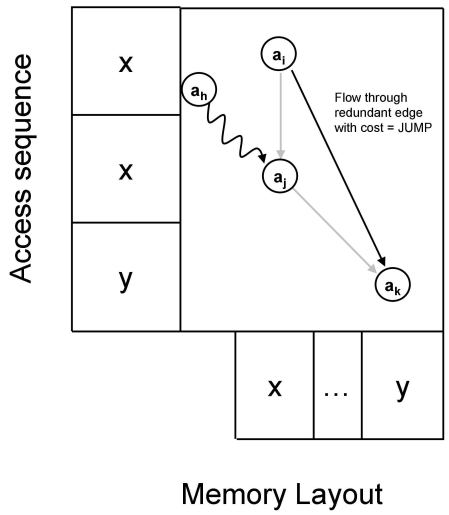



(a) An example of a redundant edge when $a_i$ and $a_j$ access the same memory location

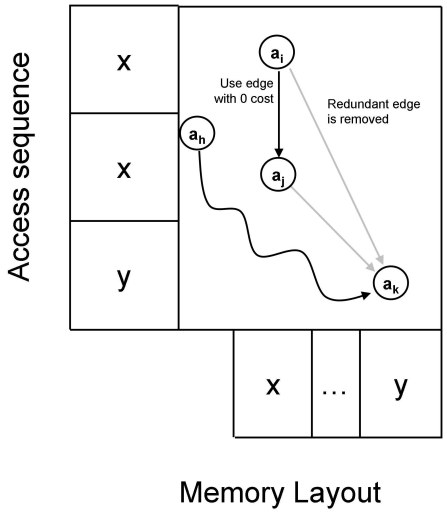



(b) An example of a redundant edge when $a_j$ and $a_k$ access the same memory location

Figure 3.4: Two examples of redundant edges in the network-flow graph.

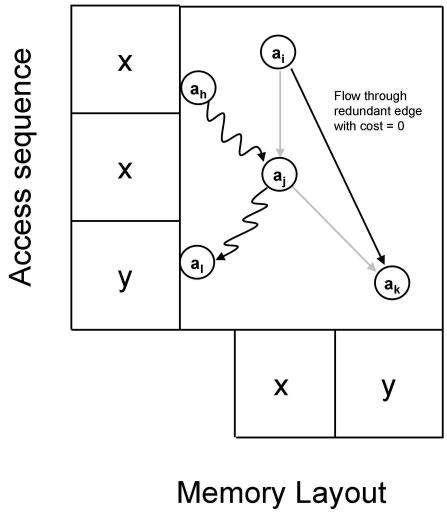



(a) The four accesses require at least JUMP cycles of overhead.

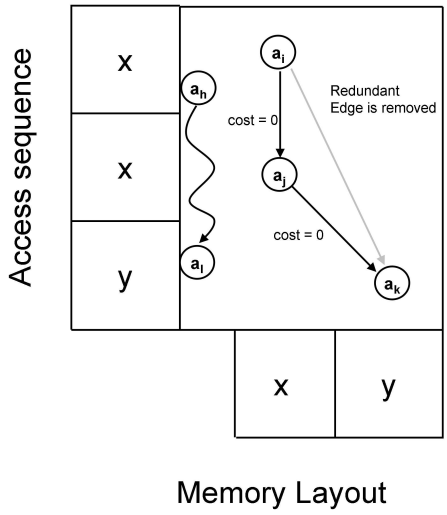



(b) After removing the redundant edge, the four accesses incur a cost of, at most, one JUMP.

Figure 3.5: Let the grey lines be edges with no flow; and black lines be edges with a unit flow. Removing a redundant edge with cost=JUMP in the network-flow graph does not increase the cost of the flow.




(a) The five accesses incur a cost of zero, one or two JUMPs.




(b) After removing the redundant edge, the five accesses incur a cost of, at most, one JUMP.

Figure 3.6: Let the grey lines be edges with no flow; and black lines be edges with a unit flow. Removing a redundant edge with cost=0 in the network-flow graph does not increase the cost of the flow.

be the edge used to access $a_j$, as shown in Figure 3.5(a). Regardless of the cost of edge $(a_h, a_j)$, at least JUMP cycles of overhead is required to perform the four accesses. If $(a_i, a_k)$ is removed, an alternative flow, consisting of two edges, can be used to perform the four variable accesses, as shown in in Figure 3.5(b). Given that $a_i$ and $a_j$ access the same memory location, the flow through $(a_i, a_j)$ has a cost of 0, and the flow through $(a_h, a_k)$ has a cost of, at most, JUMP cycles. Thus, removing $(a_i, a_k)$ does not increase the cost of the minimum-cost flow.

Second, consider the case where $(a_i, a_k)$ has cost $c_{i,k} = 0$, as show in Figure 3.6. If $(a_i, a_k)$ is used in the minimum-cost flow, then there must exist a second flow to access $a_j$. Let $a_h$ and $a_l$ be the accesses that occur before and after $a_j$, respectively. Let edges $(a_h, a_j)$ and $(a_j, a_l)$ be the edges that carry flow through $a_j$, as shown in Figure 3.6(a). Recall that $a_i$ and $a_j$ access the same memory location, while $a_i$ and $a_k$ access adjacent memory locations (because $c_{i,k} = 0$). Thus, after removing $(a_i, a_k)$, a zero-cost flow can be used to access $a_i, a_j$ and $a_k$. A second flow is then used to access $a_h$ and $a_l$ through $(a_h, a_l)$, as depicted in Figure 3.6(b). The cost of the flow through $(a_h, a_l)$ depends on the cost of edges $(a_h, a_j)$ and $(a_j, a_l)$:

- If edges $(a_h, a_j)$ and $(a_j, a_l)$ have a total cost of zero, then $a_h$ and $a_l$ must both access the same memory location (adjacent to the memory location accessed by $a_j$). Thus, the flow through $(a_h, a_l)$ has zero cost and the total cost of the flow in Figure 3.6(b) is zero;

- Alternatively, if $a_h$ and $a_l$ do not access the same memory location, either $(a_h, a_j)$ or $(a_j, a_k)$ (or both) have a cost of one JUMP. Thus, the total cost of the original flow (Figure 3.6(a)) is at least one JUMP. After removing the redundant edge, the flow through $(a_h, a_l)$ incurs, at most, a cost of one JUMP.

Thus, for all occurrences in the network flow graph where $a_i$ and $a_j$ access the same memory location, edge $(a_i, a_k)$ can be removed. Similarly, the same arguments can be used to show that if $a_j$ and $a_k$ access the same memory location, $(a_i, a_k)$ can be removed without increasing the cost of the minimum-cost flow.

Instead of removing all redundant edges from $G$, it is also possible to construct $G'$ directly. Specifically, for each access $a_i$, only the first subsequent access to all

other memory locations are used to form an edge. Algorithm 1 shows how the edges of the network flow graph can be found.

**Algorithm 1** Construct Network-Flow Edges

```
Input: AccessSequence S
EdgeArray Edges ← ()
for each V ∈ S do
   BooleanArray Flags ← ()
   for each V' ∈ S, V' occurs after V do
      if Flags.get(V') == false then
         Edges.add(V,V')
         Flags.set(V') ← True
      else
         continue
      end if
   end for
end for
return Edges
```

### 3.2.1 Impact of Removing Redundant Edges

The removal of redundant edges from $G$ significantly reduces the size of the network flow graph. Let $l$ be the length of the access sequence, and $n$ be the number of variables. In the original graph $G$, there is one edge between all pairs of accesses, resulting in $\frac{(l-1)(l-2)}{2}$ edges between the accesses. In this new MCF model, each access has, at most, $n$ out-going edges (one for every possible memory location). Thus, the total number of edges is, at most, $ln$. Thus, the new model has $O(\frac{l}{n})$ fewer edges in the network-flow graph. A reduction in the number edges results in a lower running time to compute the overhead.

Table 3.1 presents the size of the network-flow graph for the access sequences used in the experimental evaluation of this thesis (see Chapters 4 and 5). For these testcases, at least 39% of the edges are identified as redundant and can be removed from the network flow graph; however, due to the relatively small sizes of graphs, the running-time improvements to find a minimum-cost are not detectable.

To better demonstrate the advantages of removing redundant edges, Algorithm 2 is used to generate large access sequences. Each of the access sequences generated

| Testcase | Number of Vertices | Number of Edges (with redundant edges) | Number of Edges (no redudant edges) |
|---|---|---|---|
| iir_arr | 21 | 253 | 153 |
| iir_arr_swp | 33 | 595 | 356 |
| latnrm_arr_swp | 30 | 496 | 288 |
| latnrm_ptr | 30 | 496 | 276 |
| latnrm_ptr_swp | 30 | 496 | 287 |

Table 3.1: The size of the network-flow graph, produced by different access sequences, can be significantly reduced by removing redundant edges.

has between 25 and 125 variables, with each variable accessed an average of 4 times. Thus, each access sequence contains between 100 and 500 accesses. The resulting network-flow graphs have hundreds of vertices and thousands of edges, as listed in Table 3.2. To find the minimum-cost flost for each network-flow graph, each problem is modeled as a linear programming problem and solved using the GNU linear-programming solver, `glpsol`.

**Algorithm 2** Generate Random Access Sequence

```
Input: Integer NumVariables, Integer NumAccesses
Integer I
AccessSequence A
for I ← 0 to NumAccesses do
   Access V ← rand() mod NumVariables
   A.add(V)
end for
return A
```

Table 3.2 shows how removing redundant edges can significantly reduce the size of the graphs and improve the time required to find the minimum-cost flow. As the problem size increases, the benefits of removing redundant edges becomes more apparent. In the largest network-flow graph, removing the redundant edges reduces both, the size of the graph, and the execution time, by over 60%.

## 3.3 Conflict General Offset Assignment

The processor model used in this thesis, the `TMS320C54X` family of DSPs, has an additional architectural feature that affects the problem of minimizing address-computation overhead. The processor has a dual memory bus that allows it to access

| Random Testcase | Number of Vertices | With Redundant Edges | | No Redudant Edges | |
|---|---|---|---|---|---|
| | | Edges | Time (s) | Edges | Time (s) |
| Testcase 1 | 100 | 5,151 | 5.7 | 2,223 | 2.6 |
| Testcase 2 | 200 | 20,301 | 38.8 | 8,101 | 18.1 |
| Testcase 3 | 300 | 45,451 | 129.3 | 18,846 | 49.7 |
| Testcase 4 | 400 | 80,601 | 294.3 | 31,909 | 127.3 |
| Testcase 5 | 500 | 125,751 | 575.4 | 48,477 | 227.4 |

Table 3.2: The impact of removing redundant edges becomes more significant as the size of the access sequence, and resulting network-flow graph, increases.

```
x = y + z
```

(a) A 3-operand instruction

```
ACC = y
ACC += z
x = ACC

'y z x'
```

(b) Instance of GOA

```
ACC = y + z
x = ACC

'{z y} x'
```

(c) Instance of C-GOA

Figure 3.7: The `TMS320C54X` family of DSPs can encode a three-operand instruction in fewer machine instructions than tradition accumulator-based architectures.

two memory locations indexed by two distinct address registers simultaneously. Specifically, the processor has instructions that can use two different addressing modes on two different address registers to access two memory locations, perform an arithmetic operation and store the result to an accumulator. Figure 3.3 demonstrates that the dual-operand instructions create another opportunity to further reduce address-computation overhead. Traditionally, when converting three-address codes into accumulator-based assembly instructions, two instructions are required to read and operate on the two source operands, and a third instruction is required to write the result to memory, as shown in Figure 3.7(b). In the `TMS320C54X` family of DSPs, if two address registers are available to store the addresses of both source operands, only two instructions are required: one instruction to read and operate on the both source operands, and one instruction to store the result to memory, as shown in Figure 3.7(c).

In order to effectively use the dual-memory bus and associated dual-operand instructions, current models for address-computation must be revised. The in-

troduction of dual-operand instructions affects three processes when minimizing address-computations: access-sequence generation, address-register assignment, and address-code generation.

Given an instruction schedule of three-address codes that use commutative operations, it is possible to generate many valid access sequences (see Section 1.2). With dual-operand instructions, the access sequence can be augmented with additional information to represent opportunities in the instruction schedule where two variables can be simultaneously loaded from memory. The *Augmented Access Sequence* (AAS) uses matching pairs of delimiters, such as { }, to denote when two variables can be accessed simultaneously. For example, the augmented access sequence for the instructions in Figure 3.7(c) is denoted as '{z y} x'. However, dual accesses to the same variable in an instruction, such as `z = x` **op** `x`, use a single address register, thus, can be represented by a single access to `x` in the access sequence.

Augmented access sequences impose additional constraints on the offset assignment problem. In particular, during address-register assignment, two variables that are accessed simultaneously are said to be in *conflict* because they cannot be assigned to the same address register, and will not appear in the same sub-layout. This new variant of the offset assignment problem is called the *Conflict General Offset Assignment* (C-GOA) problem.

The third process affected by dual-operand instructions is address-code generation. The models for generating addressing code must ensure that if two different variables are accessed, two different address registers will be used. Fortunately, the three models for address-code generation (MCF, MCC, MWPM) already ensure two simultaneous accesses will be accessed by two different address registers. Let $a_i$ and $b_i$ represent two accesses to different memory locations that occur simultaneously in the $i^{th}$ instruction of a schedule. The current models do not create any edges between $a_i$ and $b_i$ because $i = i$, implying that a single address register cannot consecutively access $a_i$ and then $b_i$, or vice versa. Moreover, in all models, there are no paths connecting $a_i$ and $b_i$, implying that the two accesses must be done through two different address registers.

The ability to generate optimal addressing code in the presence of dual-operand instructions creates an interesting problem. To use existing offset assignment algorithms (or variants of it), one must consider that certain pairs of variables cannot be accessed by the same address register; however, there is no actual constraint on the placement of variables in memory.

## 3.4 Summary

The three contributions presented in this chapter can be used to more accurately model the problem of minimzing address-computation overhead. First, the minimum-cost flow technique can efficiently produce optimal addressing code. Second, using optimal addressing code in conjunction with traditional offset assignment algorithms introduces a new problem, the Memory Layout Permutations problem, that must be considered when generating memory layouts. Third, the MCF technique can model addressing overhead in DSP architectures with a dual memory bus. Thus, in order to effectively minimize address-computation overhead, memory layouts must be evaluated using the models presented in this chapter.

# Chapter 4

# Understanding the Solution Space

With the introduction of optimal address-code generation, as presented in Sections 2.4 and 3.2, the problem of minimizing overhead becomes difficult to model and solve. The only method to accurately evaluate the overhead of an access sequence and memory layout is to generate and solve a network-flow problem. To better understand the effects of access-sequence generation and offset assignment on overhead, the solution space for several problem instances are exhaustively evaluated. That is, for a given basic block, all valid access sequences and memory layouts are generated, and their overheads are evaluated using the previously presented network-flow techniques. Section 4.2 presents the impact of offset assignment on overhead, and Section 4.3 reveals how different access sequences make a negligible impact on address-computation overhead. Section 4.4 presents how the topology and characteristics of the solution space can be evaluated and used to guide future algorithms to generate low-overhead memory layouts. The results of the exhaustive evaluation is the only known method to find an accesses sequence and memory layout with minimum overhead, and is used in in Chapter 5 to determine the efficiency of current algorithms.

## 4.1 Experimental Methodology

As presented thus far, the minimization of address-register computation overhead is a difficult problem. Although the problem of generating addressing code has been solved, the impact of scheduling and offset assignment on the final overhead

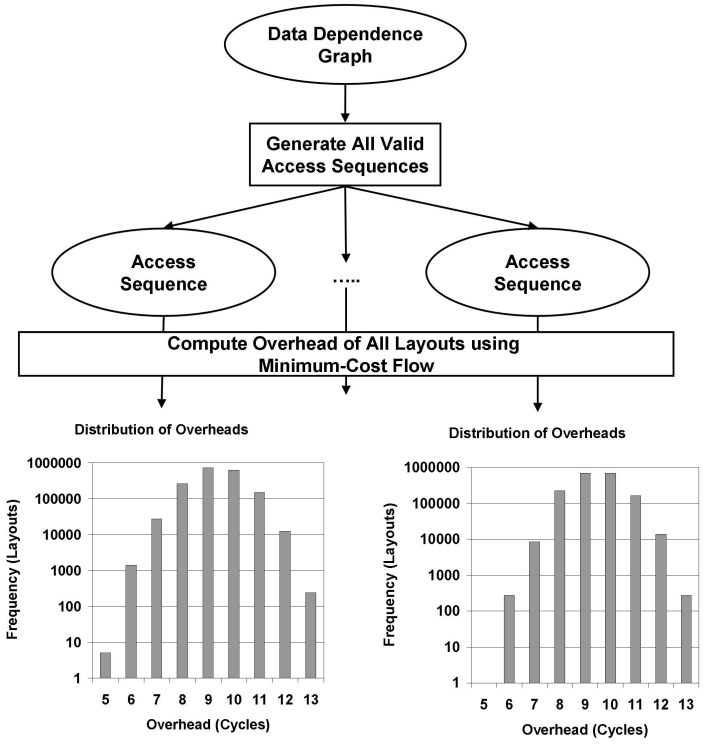



Figure 4.1: Methodology used to exhaustively evaluate the solution space.

is not clear. The problem formulations proposed by other researchers use assumptions that are invalidated by the introduction of optimal address-code generation. To determine the significance of the instruction scheduling and offset assignment problems in the presence of optimal address-code generation, several experiments were conducted to obtain the solution space for some DSP benchmark kernels. Figure 4.1 depicts the methodology used to exhaustively evaluate the solution space.

Given $n$ *variables*, there are $n!$ possible memory layouts. However, there are pairs of memory layouts that are guaranteed to have equivalent overheads. Specifically, for every memory layout $M$, there is a reciprocal layout $M'$, with variables in reverse order, that has the same overhead. For a fixed access sequence, each post-increment used to access a variable in $M$ can be replaced with a post-decrement to access the same variable in $M'$. Figure 4.1 shows how two memory layouts can be accessed with the same overhead.

The testcases used for the experiments are obtained by compiling kernels in the UTDSP benchmark suite [12]. Each kernel is compiled with `gcc` version 3.3.2

‘x y z y x’

(a) An access sequence

| access | instruction |
|---|---|
| | AR = &x |
| x | AR += 1 |
| y | AR += 1 |
| z | AR -= 1 |
| y | AR -= 1 |
| x | AR |

(b) Layout [x y z]

| access | instruction |
|---|---|
| | AR = &x |
| x | AR -= 1 |
| y | AR -= 1 |
| z | AR += 1 |
| y | AR += 1 |
| x | AR |

(c) Layout [z y x]

Figure 4.2: Two memory layouts that are reciprocals of each other are considered equivalent because they have the same overhead.

| Kernel | Number of Accesses | Number of Variables | Dual-Operand Accesses | Possible Memory Layouts |
|---|---|---|---|---|
| iir_arr | 21 | 8 | 2 | 20,160 |
| iir_arr_swp | 33 | 12 | 6 | 239,500,800 |
| latnrm_arr_swp | 30 | 10 | 6 | 1,824,400 |
| latnrm_ptr | 30 | 10 | 6 | 1,824,400 |
| latnrm_ptr_swp | 30 | 10 | 6 | 1,824,400 |

Table 4.1: Size of problem and solution space for selected kernels

using -O2 optimization. The most frequently executed basic block (usually from the inner-most loop) is identified, and the block's DDG and the default instruction schedule prior to register allocation are extracted. In order for exhaustive evaluations of the solution spaces to be feasible, only basic blocks with 12 variables or less were examined. The kernels are presented in Table 4.1.

The results of these experiments support the following conclusions:

- Instruction scheduling has a very minor impact on overhead. For the two testcases examined, all valid schedules produce similar distributions of overhead values. Specifically, 99% of all distributions have the exact same range of possible overhead values. Conversely, for any given memory layout, using all valid schedules usually produce a smaller range of possible overhead values.

- Contrary to the conjectures of other authors [6], the selection of memory layout has a significant impact on address-computation overhead. Less than

| Access Sequence | overhead (cycles) | GOA Number of Layouts | GOA % of Layouts | C-GOA Number of Layouts | C-GOA % of Layouts |
|---|---|---|---|---|---|
| iir_arr | 4 | 5 | 0.02% | 0 | 0.00% |
| | 5 | 281 | 1.39% | 134 | 0.66% |
| | 6 | 5707 | 28.31% | 3928 | 19.48% |
| | 7 | 10526 | 52.21% | 10346 | 51.32% |
| | 8 | 3641 | 18.06% | 5390 | 26.74% |
| | 9 | 0 | 0.00% | 362 | 1.80% |
| Average overhead | | 6.87 | | 7.09 | |

Table 4.2: Number of layouts with a specific address-computation overhead, for the GOA and CGOA solution spaces, for the iir_arr benchmark kernel.

0.1% of all memory layouts for any given access sequences have the minimum overhead; and more than 98% of all memory layouts have an overhead that is 50% higher than minimum. Thus, using optimal address-code generation alone is not sufficient to minimize overhead.

## 4.2 Offset-Assignment Solution Space

The distribution of overhead values for each benchmark kernel are shown in Tables 4.2 through 4.6. The access sequences are obtained using the schedule produced by `gcc` before register allocation using `-O2` optimization. The schedule of three-address codes is translated into two access sequences. One sequence contains parallel variable accesses, representing a C-GOA problem instance. To generate GOA problem instances, each operand in a parallel access is separated into two distinct accesses (as described in Section 1.2).

The most notable observation is that despite the ability to generate optimal addressing code, offset assignment has a significant impact on the address-computation overhead. In the GOA version of the access sequences, some memory layouts have twice as much overhead as the optimal layouts. In the C-GOA problem instances, some layouts have 2.5 times the overhead of the optimal layouts. The significance of offset assignment also increases with the problem size.

The wider range of possible overhead values in the C-GOA problem instances is due to the constraints caused by dual-operand accesses. Although the potentially

| Access Sequence | overhead (cycles) | GOA Number of Layouts | GOA % of Layouts | C-GOA Number of Layouts | C-GOA % of Layouts |
|---|---|---|---|---|---|
| iir_arr_swp | 6 | 144 | 0.00% | 56 | 0.00% |
| | 7 | 19557 | 0.01% | 7117 | 0.00% |
| | 8 | 1514918 | 0.63% | 551740 | 0.23% |
| | 9 | 21757157 | 9.08% | 10228196 | 4.27% |
| | 10 | 90478895 | 37.78% | 59796598 | 24.97% |
| | 11 | 104101226 | 43.47% | 109480502 | 45.71% |
| | 12 | 21628904 | 9.03% | 52373319 | 21.87% |
| | 13 | 0 | 0.00% | 6818622 | 2.85% |
| | 14 | 0 | 0.00% | 243532 | 0.10% |
| | 15 | 0 | 0.00% | 1118 | 0.00% |
| Average overhead | | 10.51 | | 10.94 | |

Table 4.3: Number of layouts with a specific address-computation overhead, for the GOA and CGOA solution spaces, for the iir_arr_swp benchmark kernel.

| Access Sequence | overhead (cycles) | GOA Number of Layouts | GOA % of Layouts | C-GOA Number of Layouts | C-GOA % of Layouts |
|---|---|---|---|---|---|
| latnrm_arr_swp | 6 | 323 | 0.02% | 19 | 0.00% |
| | 7 | 10785 | 0.59% | 1671 | 0.09% |
| | 8 | 253379 | 13.96% | 112718 | 6.21% |
| | 9 | 918134 | 50.60% | 561629 | 30.95% |
| | 10 | 631779 | 34.82% | 890592 | 49.08% |
| | 11 | 0 | 0.00% | 229645 | 12.66% |
| | 12 | 0 | 0.00% | 17974 | 0.99% |
| | 13 | 0 | 0.00% | 152 | 0.01% |
| Average overhead | | 9.20 | | 9.71 | |

Table 4.4: Number of layouts with a specific address-computation overhead, for the GOA and CGOA solution spaces, for the latnrm_arr_swp benchmark kernel.

| Access Sequence | overhead (cycles) | GOA Number of Layouts | GOA % of Layouts | C-GOA Number of Layouts | C-GOA % of Layouts |
|---|---|---|---|---|---|
| latnrm_ptr | 6 | 1449 | 0.08% | 494 | 0.03% |
| | 7 | 29682 | 1.64% | 10349 | 0.57% |
| | 8 | 456647 | 25.17% | 241667 | 13.32% |
| | 9 | 929244 | 51.21% | 721325 | 39.76% |
| | 10 | 397378 | 21.90% | 690065 | 38.03% |
| | 11 | 0 | 0.00% | 138406 | 7.63% |
| | 12 | 0 | 0.00% | 11950 | 0.66% |
| | 13 | 0 | 0.00% | 144 | 0.01% |
| Average overhead | | 8.93 | | 9.41 | |

Table 4.5: Number of layouts with a specific address-computation overhead, for the GOA and CGOA solution spaces, for the latnrm_ptr benchmark kernel.

| Access Sequence | overhead (cycles) | GOA Number of Layouts | GOA % of Layouts | C-GOA Number of Layouts | C-GOA % of Layouts |
|---|---|---|---|---|---|
| latnrm_ptr_swp | 6 | 323 | 0.02% | 2 | 0.00% |
| | 7 | 7706 | 0.42% | 1359 | 0.07% |
| | 8 | 225109 | 12.41% | 113103 | 6.23% |
| | 9 | 905303 | 49.90% | 581575 | 32.05% |
| | 10 | 675959 | 37.26% | 889366 | 49.02% |
| | 11 | 0 | 0.00% | 212410 | 11.71% |
| | 12 | 0 | 0.00% | 16399 | 0.90% |
| | 13 | 0 | 0.00% | 186 | 0.01% |
| Average overhead | | 9.24 | | 9.69 | |

Table 4.6: Number of layouts with a specific address-computation overhead, for the GOA and CGOA solution spaces, for the latnrm_ptr_swp benchmark kernel.

higher overhead can negate the benefits of using dual-operand instructions, a net improvement in overhead is still expected by using dual-operand instructions instead of the traditional register-memory instructions. In all kernels, using dual operand instructions increases the worst-case overhead; however, using dual operand instructions still reduces the overall number of instructions. For example, in four of the kernels, using dual-operand instructions to access the C-GOA access sequence can require up to 13 cycles of overhead, while using single-operand instructions to access the GOA access sequence requires no more than 10 cycles of overhead. However, all of the kernels have 6 dual-operand accesses (see Table 4.1), thus, there are 6 fewer instructions to encode when using the C-GOA access sequence.

## 4.3 Instruction-Scheduling Solution Space

In order to understand how instruction scheduling affects overhead, the solution space of two kernels were examined. The methodology for examining the solution space is shown in Figure 4.1. Two kernels are selected and compiled with `-O2` optimization in `gcc`. Before the compiler's register allocation phase, a DDG of the most frequently executed basic block is extracted. All possible schedules for the DDG are found using a listing algorithm [10]. As mentioned in Section 2.3, each instruction in a schedule of three-address codes can potentially double the number of possible access sequences. If there are $n$ instructions that use a commutable operator on two variables in memory, then there are $2^n$ possible GOA access sequences for each possible schedule. Conversely, a schedule of three-address codes precisely translates to a single *augmented access sequence*, and hence, a single C-GOA problem instance. To keep the computational requirements of this experiment feasible, only the C-GOA problem space is examined. Thus, for this section, schedules and access sequences can be used interchangeably. For each access sequence, all possible memory layouts are found and the overhead is computed. The final result of the experiment is a distribution of possible overhead values for each access sequence. In the rest of this thesis, an access sequence $S$ is said to "admit" an overhead of $c$ cycles, if there exists a memory layout with $c$ cycles of overhead when optimal

| kernel | Number of Instructions | Number of Schedules | Layouts per Schedule | Range of Overheads |
|---|---|---|---|---|
| iir_arr | 11 | 864 | 20,160 | 4-9 |
| latnrm_ptr | 14 | 3120 | 1,814,400 | 5-13 |

Table 4.7: Scheduling statistics for two benchmark kernels

addressing code is generated.

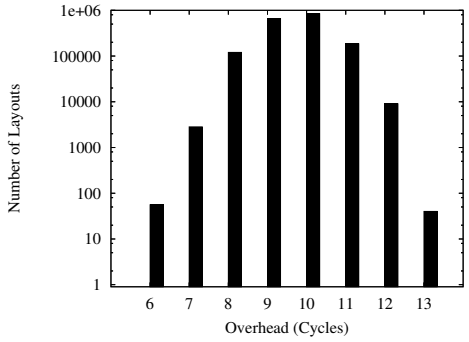
The two DDGs used in the experiments are presented in Table 4.7. Due to the large number of access sequences evaluated, it is impractical to present the distribution of overhead values for each sequence; however, the distributions can be summarized quite succinctly. In the iir_arr benchmark, *all* access sequences admit overheads between 5 and 9 cycles. Additionally, 540 out of 864 sequences also admit the minimum overhead of 4 cycles. In the latnrm_ptr kernel, *all* sequences admit overheads between 6 and 13 cycles, and 19 out of 3120 sequences admit the minimum overhead of 5 cycles. That is, for the two kernels examined, all schedules admit an overhead value that is within 1 cycle of the minimum possible overhead, suggesting that the scheduling has a negligible impact on overhead.

To better understand how an access sequence affects overhead, metrics for evaluating the distribution of overheads are presented. Specifically, a method for classifying *desirable* and *undesirable* access sequences is required. An access sequence can be classified as *undesirable* in three ways:

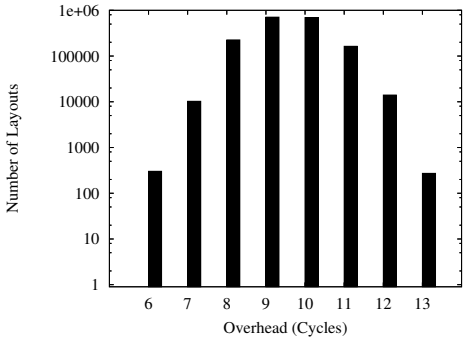
- it has the highest average overhead value;
- it has the most number of layouts with the maximum overhead;
- it has the fewest number of layouts with the minimum overhead.

Similarly, a *desirable* access sequence can be identified in the following three ways:

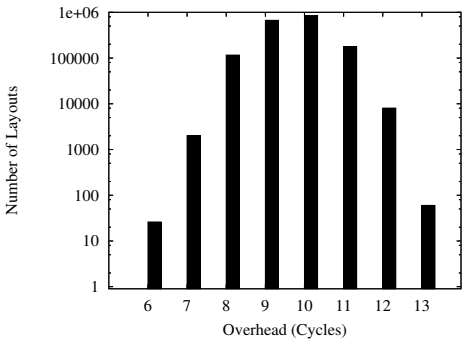
- it has the lowest average overhead value;
- it has the most number of layouts with the minimum overhead;
- it has the fewest number of layouts with the maximum overhead.




(a) Distribution with highest average overhead




(b) Distribution with most number of high-overhead layouts

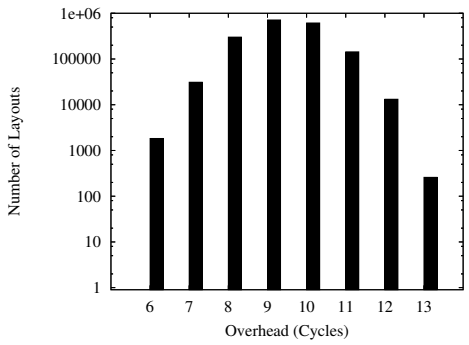



(c) Distribution with least number of low-overhead layouts

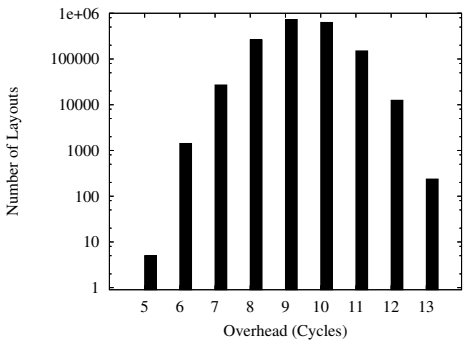
Figure 4.3: Distribution of overhead values produced by three C-GOA access sequences from the latnrm_ptr kernel that can be considered undesirable.

The distribution of overheads in three undesirable and three desirable access sequences for the latnrm_ptr kernel are shown in Figures 4.3 and 4.4 respectively. The distribution of overheads between all of the layouts are very similar, indicating that the difference between a desirable and undesirable sequence is negligible. Additionally, many schedules can be classified in both categories. In particular, many schedules that admit a high overhead for many layouts also admit a low overhead for many other layouts. The difficulties in classifying access sequences as desirable or undesirable further suggest that scheduling has a minimal impact on the final overhead.

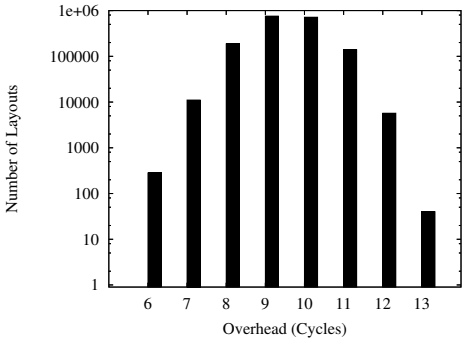
A complementary analysis of the schedules is to examine the impact of schedul-




(a) Distribution with lowest average overhead




(b) Distribution with most number of low-overhead layouts




(c) Distribution with least number of high-overhead layouts

Figure 4.4: Distribution of overhead values produced by three C-GOA access sequences from the latnrm_ptr kernel that can be considered desirable.

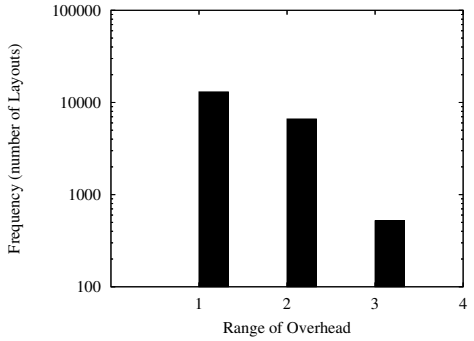
ing for a fixed memory layout. For each memory layout $M$, the overhead of accessing $M$ using all valid access sequences is recorded. For the iir_arr benchmark, there are over 20,000 distributions of overhead values to examine. For the latnrm_ptr benchmark, there are over 1.8 million distributions to examine. Thus, it is impractical to present all distributions; however, Figure 4.5 presents the number of layouts that have a particular range of overhead values. For example, in the iir_arr benchmark (shown in Figure 4.5(a)), over 10,000 memory layouts (out of a possible 20,160 layouts) have a range of one overhead; that is, the overhead for each of the 10,000 layouts has a fixed overhead value regardless of the schedule used to access the layout. Thus, a memory layout that is optimal for any particular access sequence is optimal or near-optimal for *all* other valid access sequences for the given DDG.

The range of overhead values for different memory layouts in the latnrm_ptr benchmark is presented in Figure 4.5(b). Due to technical limitations, it is not easy to obtain and summarize 1.8 million distributions of overhead values. Instead, 20,160 memory layouts were randomly selected and their range of overheads were obtained. Unlike the iir_arr benchmark, there are memory layouts that are heavily influenced by scheduling. There exist memory layouts where changing the schedule affects overhead by up to 7 cycles, while the entire solution space has a range of 8 cycles. However, the overhead for a majority of layouts is less influenced by overhead. In 89% of the layouts examined, scheduling can only affect the overhead of any particular layout by, at most, 4 cycles.

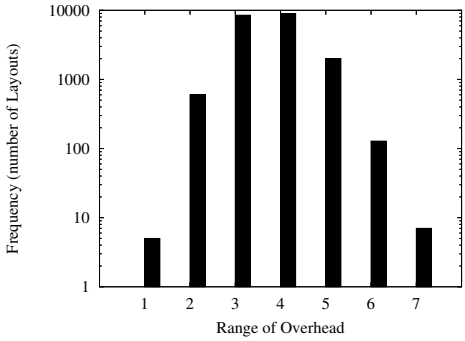
The conclusion from these analyses is that scheduling has a less significant impact on overhead than offset assignment. In particular, if an optimal offset assignment can be found for any given schedule, an improved schedule can only reduce overhead by, at most, one additional cycle.

## 4.4 Features

After evaluating the overhead for all memory layouts for a fixed access sequence, the data can be used to attempt to characterize the solution space. In particular, identifying *features* in the access graph and memory layout that correlate to overhead




(a) Distribution of overhead ranges for iir_arr




(b) Distribution of overhead ranges for latnrm_ptr

Figure 4.5: Frequency of layouts that have a specified range of possible overhead values.

values may assist in producing a heuristic to generate a memory layout with low overhead quickly. A feature $f$ defines a function $\phi_f$, that maps an access sequence $S$ and memory layout $M$ to a single value. Ideally, a feature is computationally inexpensive to compute and is used to guide a compiler's search for an optimal memory layout. Algorithm 3 shows a simple method to record the correlation between features and overhead. Given a fixed access sequence, every memory layout is evaluated using two metrics:

1. the overhead is computed using the minimum-cost flow technique;

2. the expressiveness of a feature $f$ is computed using the function $\phi_f$.

The two values are used to index into a matrix to increase a counter. After all memory layouts are evaluated, the matrix reveals the number of layouts for each possible feature-overhead value-pair. The amount of correlation between feature and overhead values can help characterize the solution space for the offset assignment problem, and identify possible ways to find low-overhead memory layouts.

The rest of this section will present several different features, and the motivation for evaluating each feature. The feature and overhead values for each of the the five C-GOA access sequences shown in Table 4.1 are evaluated to look for correlations.

**Algorithm 3** Feature-Search Framework

```
Input: AccessSequence S
Integer x,y
Integer[][] count
for each memory layout M do
    x ← overhead(S, M)
    y ← feature(S, M)
    count[x,y]++
end for
```

### 4.4.1 Transition Count

For each pair of adjacent variables in $M$, the *transitions* feature measures the number of times an address register would have to move between two memory locations. The algorithm for measuring transitions is given in Algorithm 4. A high number of transitions is expected to be detrimental to overhead values because an address register may have to move between two memory locations often, thus reducing the opportunities for the address register to access other variables.

**Algorithm 4** Transitions

```
Input: AccessSequence S, MemoryLayout M
AccessSequence S'
Integer T ← 0
for each pair of adjacent variables u,v in M do
    S' ← S composed with only u and v
    T ← T + number of times 'u v' or 'v u' occurs in S'
end for
return total
```

### 4.4.2 Path Weight

The traditional approaches for generating memory layouts rely on finding path covers on an access graph. The *path-weight* feature measures the weight of a path (representing a given memory layout) in the access graph. Algorithm 5 is used to measure this feature. If this feature was evaluated on SOA problem instances, there would be a perfect correlation between the weight of the path cover and the overhead of a layout, as the SOA problem has been reduced to the maximum-weight path cover problem [18]. However, the usefulness of this feature for finding general

offset assignments is unclear because the GOA and C-GOA problems cannot be reduced to the maximum-weight path cover problem,

**Algorithm 5** Path Weight

```
Input: AccessSequence S, MemoryLayout M
AccessGraph G ← AccessGraph(S)
Integer T ← 0
for each pair of adjacent variables u,v in M do
   T ← T + weight of (u,v) in G
end for
return T
```

### 4.4.3 Distance Measurement

The *distance* feature measures the number of words between two consecutively accessed memory locations. That is, for each consecutive pair of accesses $x$ and $y$ in the access sequence, the distance is the number of words between $x$ and $y$ in memory. Algorithm 6 is used to measure distance. Although the overhead incurred by accessing two non-adjacent variables is fixed, regardless of distance, smaller jump distance can potentially keep an address register pointing to a more confined region of memory. Memory layouts with more spatial locality can potentially have lower overhead values.

**Algorithm 6** Distance Measurement

```
Input: AccessSequence S, MemoryLayout M
Integer T ← 0
for each pair of consecutive variables u,v in S do
   T ← T + distance(u,v) in M
end for
return T
```

### 4.4.4 Interleavings

Given two variables in an access sequence, there are many possible interleavings of accesses to the two variables. The *interleavings* feature reveals which types of interleavings of accesses are prevalent between pairs of adjacent variables in

memory. Given two variables in memory, $x$ and $y$, three types of variable access interleavings are identified:

- No interleavings. Accesses to one variable are not *interleaved* with accesses to another. That is, multiple consecutive accesses to one variable occurs before any accesses to the second variable, *i.e.* the access sequence contains the pattern '`x x y y`'.
- Full interleavings. One variable is accessed multiple times between two accesses of another variable, *i.e.* the access sequence contains the pattern '`x y y x`'.
- Partial interleavings. Accesses alternate between the two variables under consideration, *i.e.* the access sequence contains the pattern '`x y x y`'.

Note that pairs of variables are not restricted to one classification of interleavings; if both variables are accessed many times, multiple types of interleavings may appear several times. This feature is similar to counting *transitions*; no interleavings (one transition) is preferred over full interleavings (two transitions), and both are preferred over partial interleavings (three transitions). The advantage of using the interleavings feature over transitions is the ability to differentiate between specific types of variable access interleavings. The general methodology for identifying interleavings is presented in Algorithm 7.

### 4.4.5 Live Ranges

A generalization of the interleavings feature is to categorize two variables $x$ and $y$ based on interference between their live ranges. Let $birth(x)$ represent the point in the access sequence when $x$ is defined; let $death(x)$ represent the last use of $x$ in the access sequence; and let $e_1 < e_2$ denote an ordering of events where event $e_1$ occurs before event $e_2$. Similar to the interleavings feature, the interference of live ranges for two variables can fall into three categories.

- No Interference. No live-range interference occurs if $x$ and $y$ are never live simultaneously, *i.e.* the order of births and deaths for $x$ and $y$ is $birth(x) < death(x) < birth(y) < death(y)$.

**Algorithm 7** Interleavings

```
Input: AccessSequence S, MemoryLayout M
Integer N,F,P ← 0
AccessSequence S',S"
for each pair of adjacent variables u,v in M do
  S' ← S composed with only u and v
  for each occurrence of 'u u v v' in S' do
    N ← N + 1
  end for
  for each occurrence of 'u v v u' in S' do
    F ← F + 1
  end for
  for each occurrence of 'u v u v' in S' do
    P ← P + 1
  end for
end for
return N,F,P
```

- Full Interference. The live-range of one variable occurs within the live-range of another variable, *i.e.* the order of births and deaths is $birth(x) < birth(y) < death(y) < death(x)$.

- Partial Interference. The live-range of the two variables partially overlap. That is, one variable's birth, but not its death, occurs during the live-range of another, *i.e.* the order of events is $birth(x) < birth(y) < death(x) < death(y)$.

In access sequences where variables and defined and used exactly once, the live-range feature is equivalent to the interleavings feature. As such, layouts with higher occurrences of non-interfering live-ranges are expected to have lower overhead values. Algorithm 8 is used to empirically evaluate the live-range feature.

### 4.4.6 Conflicts

The last feature examined in this thesis is only applicable to C-GOA problem instances. Recall from Section 3.3 that two variables are *conflicting* if they are accessed simultaneously in an instruction and cannot be accessed by the same address register. The *conflict* feature reveals the number of times two adjacent variables in

**Algorithm 8** Live Ranges

```
Input: AccessSequence S, MemoryLayout M
Integer N,F,P ← 0
for each pair of adjacent variables u,v in M do
  if birth(u) < death(u) < birth(v) < death(v) then
    N ← N + 1
  end if
  if birth(u) < birth(v) < death(v) < death(u) then
    F ← F + 1
  end if
  if birth(u) < birth(v) < death(u) < death(v) then
    P ← P + 1
  end if
end for
return N,F,P
```

memory are accessed simultaneously in the access sequence. The function used to evaluate the conflict feature is shown in Algorithm 9. Two variables accessed simultaneously likely do not need to be adjacent in memory because the two variables must be accessed by two separate address registers at some point in the access sequence. Thus, layouts with low overhead are expected to have fewer pairs of adjacent variables that are accessed in simultaneously.

**Algorithm 9** Conflicts

```
Input: AccessSequence S, MemoryLayout M
Integer T ← 0
for each pair of adjacent variables u,v in M do
  T ← T + no. of times u and v accessed in parallel in S
end for
return T
```

### 4.4.7 Evaluating Features

All of the features presented are evaluated for each of the C-GOA problem instances in Table 4.1. For each access sequence, the overhead value was plotted against the feature value, as shown in Algorithm 3. Examples of these plots can be seen in Figures 4.6 and 4.8. Fifty matrices were generated because each of the five access sequences has ten features to correlate with the overhead.

For readability, the number of layouts in each cell have been converted to a percentage of the total number of layouts. For example, when reading the plot for the *path weights* feature for the iir_arr_swp kernel in Figure 4.6, row 11, column 4 has the value 12. This means that 12% of all memory layouts have an overhead of 11 cycles and a path weight of 4. Entries with '0' mean that less than 1% of layouts satisfy an [overhead,feature] value-pair; entries with '.' have zero layouts satisfying the value-pair. For features with larger matrices, a graphical version (such as Figure 4.8) is easier to read. White represents zero layouts, and darker shades of gray represent higher percentages of layouts satisfying a value-pair.

An ideal feature is one that has perfect correlation with overhead values. That is, all layouts that fully express some feature have a very low overhead value. Conversely, all layouts with very low overhead values have high feature values. Thus, ideal plots of the correlation matrices will form lines with a defined, non-zero slope. Non-useful features will have a wide range of overhead values that overlap for each value of a feature.

Unfortunately, none of the features examined have a perfect (or even strong) correlation to overhead values. Figure 4.6 shows the correlation between overhead and *path weights* for the iir_arr_swp kernel. Layouts with low path weights generally have a lower range of overhead values than layouts with a high path weight. For example, layouts with a path weight of 1 have overhead values between 8 and 15 (maximum) cycles, while layouts with path weight of 10 have overhead values between 6 (minimum) and 13 cycles. Although there is a general trend, the wide range of possible overhead values for a particular path weight means that this feature cannot be used to accurately predict a layout's overhead. Also notice that layouts with the minimum overhead (6 cycles) do not have the maximum path weight (12), further indicating that generating a memory layout by maximizing the path weight of an access graph cover is not optimal.

Another feature with a weak correlation is the *conflicts* feature. Figure 4.7 shows the relationship between overhead values and conflicts for the iir_arr_swp kernel. Layouts with the minimum overhead have a low, but not necessarily minimal, number of adjacent variables that are accessed simultaneously. Additionally,

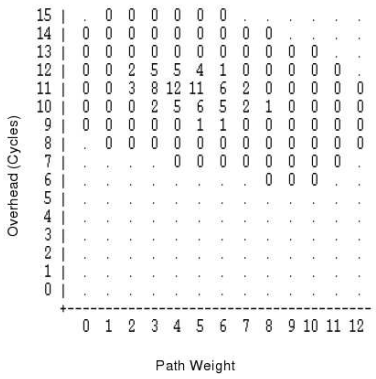



Figure 4.6: Overhead vs Path Weight for iir_arr_swp

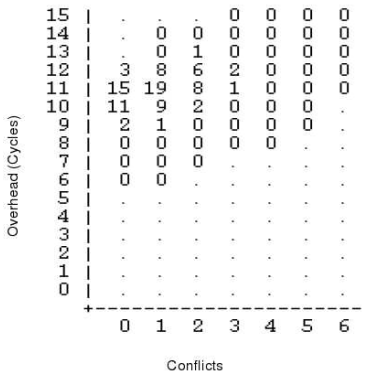



Figure 4.7: Overhead vs Conflicts for iir_arr_swp

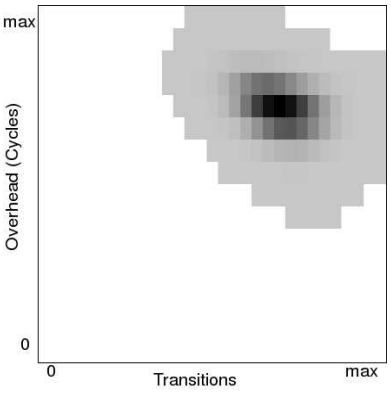



Figure 4.8: Overhead vs Transitions for iir_arr_swp

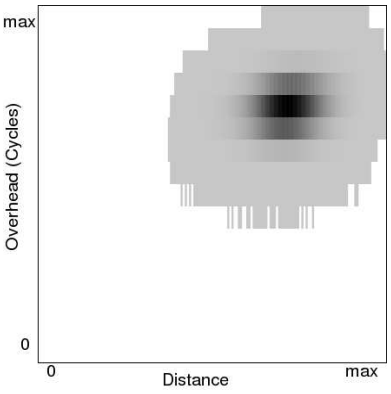



Figure 4.9: Overhead vs Distance for iir_arr_swp

minimizing the number of adjacent variables that are accessed simultaneously does not guarantee a minimum overhead. This result agrees with the original hypothesis that variables accessed simultaneously do not need to be adjacent in memory; however, generating a memory layout using this feature alone does not minimize overhead.

Two other features of the iir_arr_swp kernel, *transitions* and *distance*, are shown graphically in Figures 4.8 and 4.9. All other features evaluated for all of the kernels exhibit similarly shaped plots. The plots indicate that the features have very little correlation to overhead values. In all cases, optimal layouts express feature values that are also found in a large number of non-optimal memory layouts. In some cases, the feature value is exhibited in layouts with the maximum overhead as well.

A common trend observed is that layouts that exhibit a maximum or minimum

feature value usually have a small range of overhead values that are neither maximum or minimum. Thus, for these test cases, finding a layout that maximizes or minimizes a particular feature does not give a good indication to the actual overhead of the layout. More specifically, the experiments indicate that it is not possible to generate layouts with minimum overhead simply by minimizing or maximizing the features presented in this thesis.

## 4.5 Summary

The exhaustive evaluation of the search space for several test cases show that the offset assignment problem still has a significant impact on address-computation overhead. Surprisingly, the results also suggest that access-sequence generation has a very insignificant impact on overhead. For the test cases examined, there always exists a memory layout with an overhead within one cycle of the minimum overhead, for all valid access sequences. Additionally, the solution space of the offst assignment problem is difficult to characterize. Ten different features of memory layouts were evaluated; however, none of the features had any correlation to overhead values.

The exhaustive evaluation of the offset assignment search space also reveals the minimum and maximum overhead values possible. This data enables us to evaluate the efficiency of current offset assignment algorithms.

# Chapter 5

# Evaluating Offset Assignment Algorithms

The results of the exhaustive evaluation of the search space from Chapter 4 reveals that address-computation overhead is best addressed through offset assignment. Although many algorithms have been proposed (see Sections 2.1 and 2.2), the generated memory layouts have never been evaluated using optimal address-code generation. Additionally, using the network-flow techniques to evaluate overhead introduces a new problem, the *memory layout permutations* problem, which has never been considered. This chapter presents an empirical evaluation of available heuristics for offset assignment and presents the impact of the MLP problem. The results of these experiments support the following conclusions:

- Current offset assignment algorithms seldom produce memory sub-layouts that admit MLP solutions with the minimum possible overhead. For some access sequences, none of the algorithms produce sub-layouts that can form an optimal solution.
- Using different ARA algorithms greatly impacts the quantity and quality of memory layout permutations. Conversely, using different SOA algorithms has little impact.

## 5.1 Experimental Methodology

Figure 5.1 outlines the experimental methodology to evaluate current offset assignment algorithms. Each of the five benchmark kernels in Table 4.1 is compiled with `gcc` and `-O2` optimizations to obtain five access sequences. For each access sequence, heuristic solutions to the offset assignment problem are found by using all combinations of three ARA and five SOA algorithms presented in Sections 2.2 and 2.1 respectively. Each combination produces a set of memory sub-layouts (see Figure 3.1). Recall that if $m$ sub-layouts are produced, then there are $p = \frac{(m!)(2^m)}{2}$ possible memory layouts. The overhead of each memory layout is computed using the minimum-cost flow technique (see Section 3.2). The results of this empirical evaluation are examined in terms of the *distribution* of overhead values for the layouts produced by each combination of ARA and SOA algorithms.

## 5.2 Efficiency of Offset Assignment Algorithms

Table 5.1 shows a summary of the address-computation overhead for all memory layouts evaluated in this study. The *Exhaustive* column shows the number of memory layouts with a particular overhead in the solution space for each GOA problem. The average overhead of all layouts in each GOA problem ranges from 49% to 75% higher than minimum. Additionally, at least 98% of all layouts have an overhead 33% to 100% higher than minimum. Thus, even when the MCC technique is used to find optimal addressing code, the selection of memory layout has a significant impact on address-computation overhead.

The *Algorithmic* column of Table 5.1 shows the *combined* distribution and average address-computation overhead for memory layouts produced by *all 15 combinations* of the ARA and SOA algorithms. The distribution of the overheads obtained using the heuristic-based algorithms presented in Sections 2.2 and 2.1 indicates that, in general, the algorithms are not very effective at minimizing overhead. The average overhead of layouts produced by the algorithms for each access sequence ranges from 40% to 60% higher than minimum and is only slightly lower than the average overhead of all layouts in the solution space. Moreover, the layouts formed

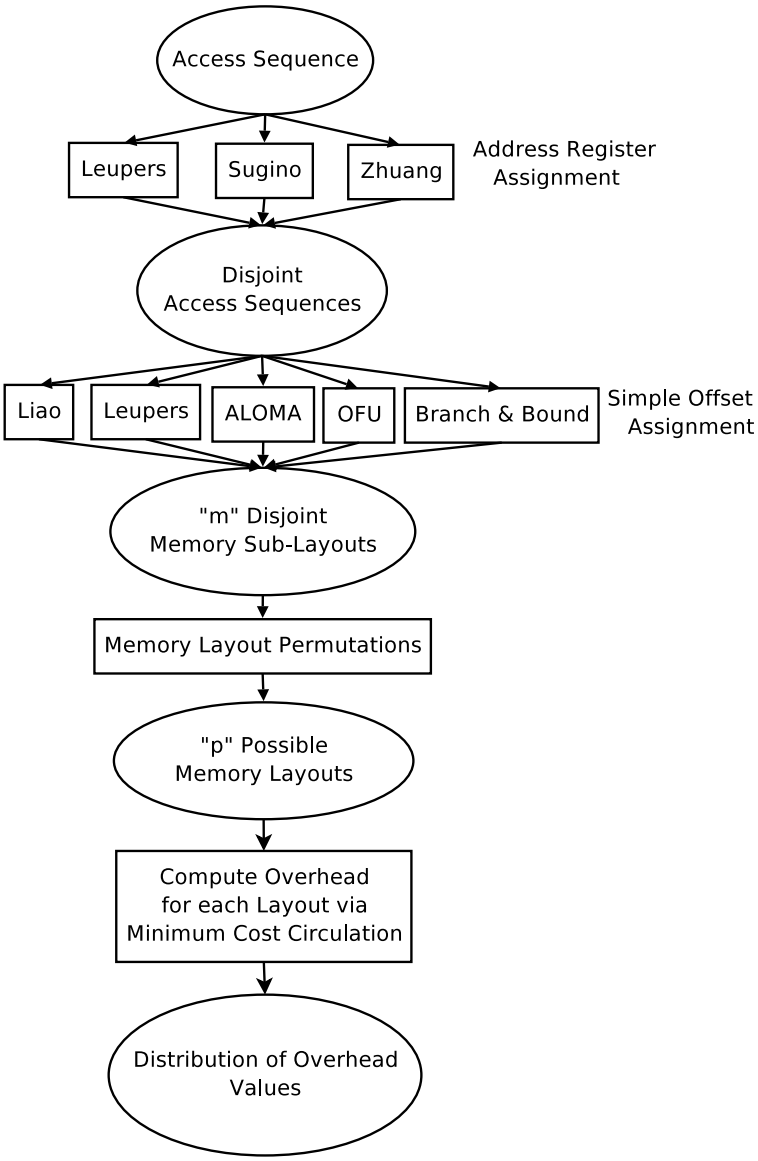



Figure 5.1: Procedure for evaluating offset assignment algorithms. There are 15 paths in the chart, for the 15 combinations of ARA and SOA algorithms.

by combining sub-layouts produced by these heuristic-based algorithms have overheads that can range from the best (minimum) to worst (maximum) possible values in the entire solution space.

## 5.3 Efficiency of ARA Algorithms

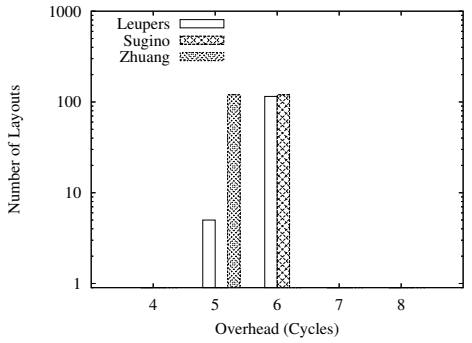
Each of the three ARA algorithms — Leupers, Sugino, and Zhuang — can be combined with five SOA algorithms (Figure 5.1) to produce a memory layout. All of the layouts produced by an ARA algorithm are combined into a set. The distribution of overhead values for the possible layouts produced by each ARA algorithm are shown in Figure 5.2. For instance, Figure 5.2(a) shows that Leupers' ARA algorithm can admit over 100 layouts with 6 cycles of overhead and 5 layouts with 5 cycles of overhead. Each of these layouts are obtained by using different SOA and MLP solutions, but all use Leupers' ARA algorithm.

For each access sequence, the total number of layouts varies between each ARA algorithm because each algorithm may use a different number of ARs, yielding a different number of permutations (see Section 3.1). Figure 5.2 indicates that ARA algorithms producing fewer layouts, such as Sugino's, tend to produce better layouts. This result indicates that it is frequently disadvantageous to use all available ARs. For instance, in Figure 5.2(b), Leupers and Marwedel's ARA algorithm yields a total of 9600 possible layouts, 2 of which have a 7-cycle overhead. Alternatively, the ARA algorithm proposed by Sugino *et al.* generates a total of 2688 possible layouts, with 61 7-cycle-overhead layouts. Similar distributions occur for the other access sequences.

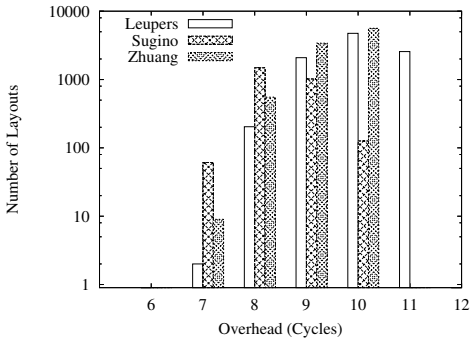
The results also suggest that locally optimal sub-layouts do not lead to globally optimal memory layouts. An ARA algorithm using more ARs assigns fewer variables to each register. In the case of Leupers and Marwedel's algorithm, and occasionally Zhuang's algorithm, as few as two variables may be assigned to an AR. Two variables can be trivially accessed without incurring JUMP overhead and are locally optimal. However, if the two variables are not adjacent in the optimal memory layouts, then the MLP solution space will never contain an optimal layout.

| Access Sequence | overhead (cycles) | Exhaustive: Number of Layouts | Exhaustive: % of Layouts | Algorithmic: Number of Layouts | Algorithmic: % of Layouts |
|---|---|---|---|---|---|
| | 4 | 5 | 0.02% | 0 | 0.00% |
| | 5 | 281 | 1.39% | 125 | 34.72% |
| iir_arr | 6 | 5707 | 28.31% | 235 | 65.28% |
| | 7 | 10526 | 52.21% | 0 | 0.00% |
| | 8 | 3641 | 18.06% | 0 | 0.00% |
| Average overhead | | 6.87 | | 5.65 | |
| | 6 | 144 | 0.00% | 0 | 0.00% |
| | 7 | 19557 | 0.01% | 72 | 0.33% |
| | 8 | 1514917 | 0.63% | 2240 | 10.23% |
| iir_arr_swp | 9 | 21757157 | 9.08% | 6515 | 29.77% |
| | 10 | 90478895 | 37.78% | 10496 | 47.95% |
| | 11 | 104101226 | 43.47% | 2565 | 11.72% |
| | 12 | 21628904 | 9.03% | 0 | 0.00% |
| Average overhead | | 10.51 | | 9.60 | |
| | 6 | 323 | 0.02% | 117 | 0.60% |
| | 7 | 10785 | 0.59% | 303 | 1.55% |
| latnrm_arr_swp | 8 | 253379 | 13.96% | 7067 | 36.26% |
| | 9 | 918134 | 50.60% | 8198 | 42.07% |
| | 10 | 631779 | 34.82% | 3803 | 19.51% |
| Average overhead | | 9.20 | | 8.78 | |
| | 6 | 1449 | 0.08% | 28 | 0.21% |
| | 7 | 29682 | 1.64% | 481 | 3.68% |
| latnrm_ptr | 8 | 456647 | 25.17% | 6093 | 46.58% |
| | 9 | 929244 | 51.21% | 6268 | 47.92% |
| | 10 | 397378 | 21.90% | 210 | 1.61% |
| Average overhead | | 8.93 | | 8.47 | |
| | 6 | 323 | 0.02% | 5 | 0.04% |
| | 7 | 7706 | 0.42% | 138 | 1.04% |
| latnrm_ptr_swp | 8 | 225109 | 12.41% | 3734 | 28.19% |
| | 9 | 905303 | 49.90% | 5881 | 44.39% |
| | 10 | 675959 | 37.26% | 3490 | 26.34% |
| Average overhead | | 9.24 | | 8.96 | |

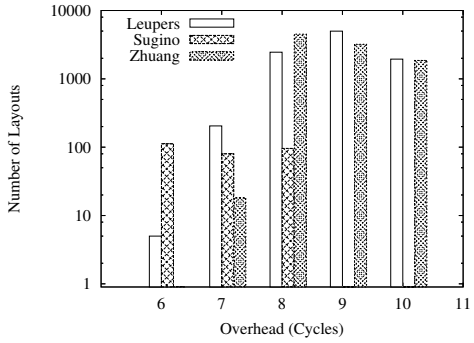
Table 5.1: Number of layouts with a specific address-computation overhead, for the entire solution space. The *Exhaustive* column shows distribution of memory layouts in the solution space. The *Algorithmic* column shows the combined distribution of layouts produced by the 15 different ARA and SOA combinations.



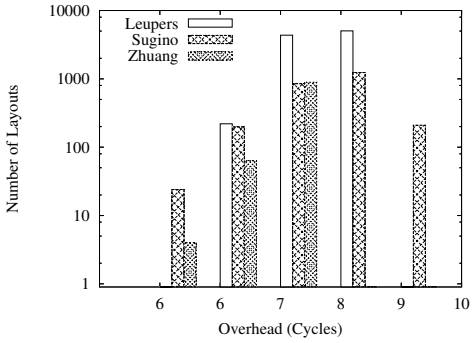

(a) iir_arr



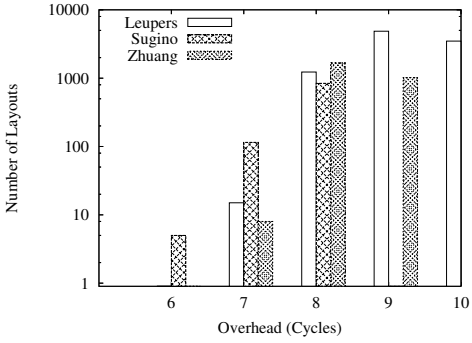

(b) iir_arr_swp




(c) latnrm_arr_swp




(d) latnrm_ptr




(e) latnrm_ptr_swp

Figure 5.2: Distribution of overhead values produced by each ARA algorithm on different test cases. The number of layouts shown for each algorithm is the union of 5 sets of layouts, each produced with one of the 5 different SOA algorithms, but using the same ARA algorithm. The layouts are plotted against the full range of overhead values obtained by exhaustive search.

| Access Sequence | overhead (cycles) | No. of Memory Layouts Leupers | Sugino | Zhuang |
|---|---|---|---|---|
| iir_arr | 4 | 0 | 0 | 0 |
| | 5 | 5 | 0 | 120 |
| | 6 | 115 | 120 | 0 |
| | 7 | 0 | 0 | 0 |
| | 8 | 0 | 0 | 0 |
| Average overhead | | 5.96 | 6.00 | 5.00 |
| iir_arr_swp | 6 | 0 | 0 | 0 |
| | 7 | 2 | 61 | 9 |
| | 8 | 204 | 1483 | 553 |
| | 9 | 2089 | 1018 | 3408 |
| | 10 | 4740 | 126 | 5630 |
| | 11 | 2565 | 0 | 0 |
| | 12 | 0 | 0 | 0 |
| Average overhead | | 10.01 | 8.45 | 9.53 |
| latnrm_arr_swp | 6 | 5 | 112 | 0 |
| | 7 | 205 | 80 | 18 |
| | 8 | 2455 | 96 | 4516 |
| | 9 | 4990 | 0 | 3208 |
| | 10 | 1945 | 0 | 1858 |
| Average overhead | | 8.90 | 6.94 | 8.72 |
| latnrm_ptr | 6 | 0 | 24 | 4 |
| | 7 | 220 | 198 | 63 |
| | 8 | 4350 | 850 | 893 |
| | 9 | 5030 | 1238 | 0 |
| | 10 | 0 | 210 | 0 |
| Average overhead | | 8.50 | 8.56 | 7.93 |
| latnrm_ptr_swp | 6 | 0 | 5 | 0 |
| | 7 | 15 | 115 | 8 |
| | 8 | 1230 | 840 | 1664 |
| | 9 | 4865 | 0 | 1016 |
| | 10 | 3490 | 0 | 0 |
| Average overhead | | 9.23 | 7.87 | 8.38 |

Table 5.2: Number of memory layouts produced by each ARA algorithm, with the specified overhead. Each column is the combined distribution of 5 sets of layouts, each produced with 5 different SOA algorithms, but using the same ARA algorithm. The layouts are plotted against the full range of overhead values obtained by exhaustive search.

## 5.4 Efficiency of SOA Algorithms

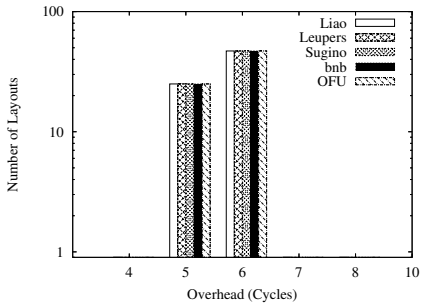
The distributions in Figure 5.3 are complementary to those in Figure 5.2, but focused on the layouts produced by each of the five SOA algorithms. For instance, Figure 5.3(b) shows that the SOA algorithm designed by Sugino *et al.* can admit over 1000 layouts with 9 cycles of overhead. Each of these layouts are obtained by combining Sugino *et al.*'s ARA algorithm with one of the three SOA algorithms.

SOA algorithms are used to estimate increases in overhead when variables are assigned to ARs; which, in turn, affects the number of sub-layouts produced by the ARA algorithms. Consequently, the total number of layouts varies between each SOA algorithm for each access sequence in Figure 5.3. Low variability between the algorithms can be partly attributed to the problem sizes. The access sequences only access 8 to 12 variables, and the ARA algorithms assign at most 6 variables to each address register. Thus, each SOA sub-problem is very small and the algorithms are likely to produce similar, and possibly optimal, sub-layouts. Specifically, no SOA algorithm consistently produces sub-layouts that admit the greatest number of optimal or near-optimal layouts. In two access sequences, OFU admits the most number of low-overhead layouts, while in one other sequence, Sugino *et al.*'s SOA algorithm admits the most number of optimal layouts.

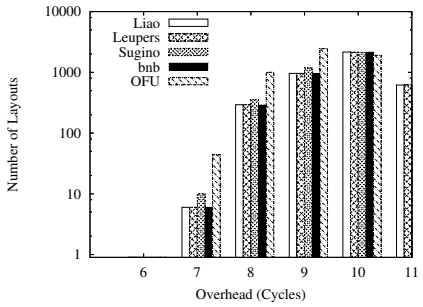
Figure 5.3 also further supports previous suggestions that combining optimal sub-layouts does not result in optimal layouts. For instance, in Figure 5.3(e), the OFU algorithm generates sub-layouts that can be combined to form optimal memory layouts, while the Branch-and-Bound algorithm, which finds optimal sub-layouts, does not allow for the creation of any optimal memory layouts.

## 5.5 Summary

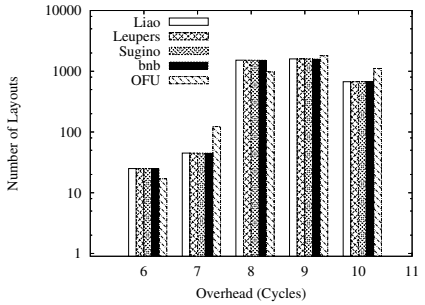
The results presented in this chapter indicate that current offset assignment algorithms are not effective at minimizing address-computation overhead. For the access sequences evaluated, changing the ARA and SOA algorithm does not make a *significant* impact on overhead. Furthermore, in order for current algorithms to *consistently* generate low-overhead memory layouts, the Memory Layout Permuta-



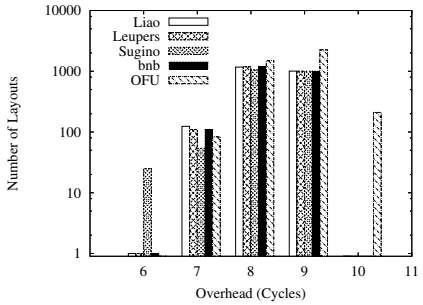

(a) iir_arr



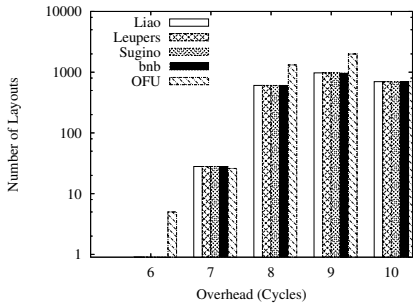

(b) iir_arr_swp




(c) latnrm_arr_swp




(d) latnrm_ptr




(e) latnrm_ptr_swp

Figure 5.3: Distribution of overhead values produced by each SOA algorithm on different test cases. The number of layouts shown for each algorithm is the union of 3 sets of layouts, each produced with one of the 3 different ARA algorithms, but using the same SOA algorithm. The layouts are plotted against the full range of overhead values obtained by exhaustive search.

| Access Sequence | overhead (cycles) | No. of Memory Layouts | | | | |
|---|---|---|---|---|---|---|
| | | Liao | Leupers | Sugino | B&B | OFU |
| iir_arr | 4 | 0 | 0 | 0 | 0 | 0 |
| | 5 | 25 | 25 | 25 | 25 | 25 |
| | 6 | 47 | 47 | 47 | 47 | 47 |
| | 7 | 0 | 0 | 0 | 0 | 0 |
| | 8 | 0 | 0 | 0 | 0 | 0 |
| Average overhead | | 5.65 | 5.65 | 5.65 | 5.65 | 5.65 |
| iir_arr_swp | 6 | 0 | 0 | 0 | 0 | 0 |
| | 7 | 6 | 6 | 10 | 6 | 44 |
| | 8 | 293 | 293 | 357 | 293 | 1004 |
| | 9 | 960 | 960 | 1187 | 960 | 2448 |
| | 10 | 2154 | 2154 | 2124 | 2154 | 1910 |
| | 11 | 619 | 619 | 354 | 619 | 354 |
| Average overhead | | 9.77 | 9.77 | 9.61 | 9.77 | 9.26 |
| latnrm_arr_swp | 6 | 25 | 25 | 25 | 25 | 17 |
| | 7 | 45 | 45 | 45 | 45 | 123 |
| | 8 | 1523 | 1523 | 1523 | 1523 | 975 |
| | 9 | 1598 | 1598 | 1598 | 1598 | 1806 |
| | 10 | 673 | 673 | 673 | 673 | 1111 |
| Average overhead | | 8.74 | 8.74 | 8.74 | 8.74 | 8.96 |
| latnrm_ptr | 6 | 1 | 1 | 25 | 1 | 0 |
| | 7 | 124 | 110 | 54 | 110 | 83 |
| | 8 | 1173 | 1187 | 1051 | 1187 | 1495 |
| | 9 | 1006 | 1006 | 1006 | 1006 | 2244 |
| | 10 | 0 | 0 | 0 | 0 | 210 |
| Average overhead | | 8.38 | 8.39 | 8.42 | 8.39 | 8.64 |
| latnrm_ptr_swp | 6 | 0 | 0 | 0 | 0 | 5 |
| | 7 | 28 | 28 | 28 | 28 | 26 |
| | 8 | 605 | 605 | 605 | 605 | 1314 |
| | 9 | 973 | 973 | 973 | 973 | 1989 |
| | 10 | 698 | 698 | 698 | 698 | 698 |
| Average overhead | | 9.02 | 9.02 | 9.02 | 9.02 | 8.83 |

Table 5.3: Number of memory layouts, produced by each SOA algorithm, with the specified overhead. Each column is the combined distribution of 3 sets of layouts, each produced with 3 different ARA algorithms, but using the same SOA algorithm. The layouts are plotted against the full range of overhead values obtained by exhaustive search.

tions problem must be addressed. Most importantly, the experiments show that even if the MLP problem can be solved, the traditional approach to generating memory layouts is not sufficient to find *optimal* memory layouts. Thus, alternative methods to generating memory layouts should be explored.

# Chapter 6

# Evaluating Alternative Algorithms

Current offset assignment algorithms are ineffective at minimizing overhead. It may be possible to improve current algorithms if the MLP problem can be addressed. However, as stated in Section 3.1, the MLP problem can be reduced to the GOA problem, and any algorithm used to solve the MLP problem can be used to directly solve the GOA problem itself. Due to the complex relation between offset assignment, address-code generation, and overhead, the *real* GOA problem is largely unsolved. In this chapter, alternative approaches and algorithms are presented. All of the approaches attempt to generate a memory layout with the consideration for the layout's overhead value as computed by the network-flow techniques. Section 6.1 presents a best-first search approach to finding a memory layout; Section 6.2 presents an algorithm that iteratively builds a memory layout using a greedy heuristic; and the last algorithm, presented in Section 6.3, uses a path cover on an *Augmented Interference Graph*. An experimental evaluation of the three approaches reveals that the simple, greedy-based algorithm consistently produces layouts with below-average overheads without excessive computations.

## 6.1 Best-First Search

One method of generating a memory layout is to systematically search the solution space by evaluating one layout at a time. Algorithm 10 describes a best-first search (BFS) algorithm used to find low-overhead memory layouts as well as to explore the topology of the offset-assignment solution space. During each iteration of the

search, the layout $L$ with the lowest known overhead is selected. Transformations are applied to $L$ to obtain a set of successor layouts. The overhead of each successor layout is evaluated and added to the queue of layouts to be transformed in future iterations of the algorithm.

**Algorithm 10** Best-First Search

```
Input: AccessSequence S
Layout L,L'
PriorityQueue Q ← {}
Overhead M ← ∞
while L ← Q.pop() do
  if L.overhead() > M then
    return M
  else
    M ← L.overhead()
  end if
  while L' ← L.successor() do
    Q.push ( L' )
  end while
end while
```

In order to generate successor layouts, a transformation is required. The transformation must be defined such that there is a series of transformations that can transform any given layout into any other valid layout in the solution space. In particular, the transformation used in this thesis transforms a layout by moving a single variable to any other valid position in the layout, as shown in Algorithm 11.

**Algorithm 11** Transformation (Successor) Function

```
Input: Layout L
LayoutSet S
Layout T1,T2
Variable V
for each V in L do
  T1 ← L.remove(V)
  for each position in T1 do
    T2 ← T1.insert(V)
    S ← T2
  end for
end for
return S
```

A potential problem with using best-first search to find a memory layout is the problem of defining a terminating condition for the search. That is, during the search, how is an optimal memory layout recognized? If the minimum overhead value is somehow known in advance, then the search can simply terminate when a layout with the minimum overhead value is found. Alternatively, if the minimum overhead value is not known, then the search may only find layouts with locally minimal overhead values, as shown in Algorithm 10. A locally minimal overhead is identified during the search when the minimal overhead among all queued layouts is larger than the lowest overhead value previously found.

Two sets of experiments are conducted to evaluate the efficiency of best-first search — one where the minimum overhead value, found through exhaustive search, is made known to the BFS algorithm, and one where the minimum overhead is not made known. Each set of experiments is conducted on C-GOA and GOA problem instances for the kernels described in Table 4.1. For each access sequence, 100 random initial layouts are generated, and 100 best-first searches are performed. To ensure that the results can be attributed to the best-first search, the experiments are compared to randomly generated layouts. Specifically, for each best-first search that terminates after evaluating $x$ number of layouts, two random searches are performed. One random search determines the number of layouts evaluated before a layout with the minimum overhead is found; a second random search determines the lowest overhead found while evaluating $x$ random layouts.

Tables 6.1 and 6.2 present the search results for the GOA and C-GOA problems, respectively, where the minimum overhead value is known in advance. In these experiments, BFS can identify an optimal layout by evaluating, on average, less than 1% of the solution space. Additionally, over 94% of the optimal layouts are found by a series of layouts that has monotonically decreasing overhead values. This observation suggests that it is possible for locally minimal overhead values to also be globally minimum, and that Algorithm 10 should be effective at finding optimal or near-optimal memory layouts. As additional evidence that using best-first search is effective, the BFS results are compared against a random search of the solution space. If layouts are generated randomly, as much as 49% of the solution

| GOA Access Sequence | Minimum Overhead | BFS Expansions | Random Expansions | Average Random Overhead |
|---|---|---|---|---|
| iir_arr | 4 | 1178 | 3,602 | 4.73 |
| iir_arr_swp | 6 | 4027 | 1,613,950 | 7.71 |
| latnrm_arr_swp | 6 | 521 | 6,371 | 6.89 |
| latnrm_ptr | 6 | 327 | 1,088 | 6.73 |
| latnrm_ptr_swp | 6 | 428 | 5,180 | 7.01 |

Table 6.1: Efficiency of best-first search and random search for GOA problems where the minimum overhead values are known.

| C-GOA Access Sequence | Minimum Overhead | BFS Expansions | Random Expansions | Average Random Overhead |
|---|---|---|---|---|
| iir_arr | 5 | 110 | 133 | 5.41 |
| iir_arr_swp | 6 | 3929 | 5,201,400 | 7.90 |
| latnrm_arr_swp | 6 | 2005 | 89,149 | 7.13 |
| latnrm_ptr | 6 | 463 | 3,819 | 6.91 |
| latnrm_ptr_swp | 6 | 9988 | 898,250 | 6.99 |

Table 6.2: Efficiency of best-first search and random search for C-GOA problems where the minimum overhead value is known.

space needs to be evaluated before an optimal layout is identified. Similarly, if the random search is restricted to evaluating the same number of layouts as BFS, the average overhead of the best layout found is between 8.2% and 31.7% higher than the minimum overhead.

Tables 6.3 and 6.4 present the search results for the GOA and C-GOA problems, respectively, where the minimum overhead value is *not* known in advance, and thus, the algorithm searches for layouts with locally minimal overheads. In six of the access sequences, using best-first search to find locally minimal overhead values often produced optimal layouts; however, the search is required to evaluate *more* layouts that necessary in order to detect the locally minimal overhead. When the local minima is also globally minimum, more layouts are evaluated because the search must continue until the minimum value in the search queue increases. Conversely, in the remaining four access sequences, using best-first search to find layouts with locally minimal overheads requires evaluating *fewer* layouts than the searches executed when the minimum overhead value was already known. Fewer layouts are evaluated because many layouts with locally minimal overheads are

| GOA Access Sequence | Average Overhead | Average BFS Expansions | Average Random Overhead |
|---|---|---|---|
| iir_arr | 4.06 | 1334 | 4.72 |
| iir_arr_swp | 6.17 | 3683 | 7.72 |
| latnrm_arr_swp | 6.01 | 1360 | 6.76 |
| latnrm_ptr | 6.00 | 5041 | 6.02 |
| latnrm_ptr_swp | 6.00 | 2284 | 6.63 |

Table 6.3: Efficiency of best-first search and random search for GOA problems where the minimum overhead value *is not* known.

| C-GOA Access Sequence | Average Overhead | Average BFS Expansions | Average Random Overhead |
|---|---|---|---|
| iir_arr | 5.00 | 1131 | 5.00 |
| iir_arr_swp | 6.11 | 3673 | 7.90 |
| latnrm_arr_swp | 6.29 | 1428 | 7.31 |
| latnrm_ptr | 6.02 | 3737 | 6.36 |
| latnrm_ptr_swp | 6.54 | 3923 | 7.03 |

Table 6.4: Efficiency of best-first search and random search for C-GOA problems where the minimum overhead value *is not* known.

*not* optimal layouts and simply require less effort to identify. However, this also results in an increase in the average overhead of the layouts found. For comparison purposes, if a random search is restricted to evaluating the same number of layouts as BFS in these experiments, the average overhead of the best layout found is up to 25.1% higher than the minimum overhead.

The results of the best-first searches indicate that the transformation described in Algorithm 11 describes a potentially viable search. The observation that Algorithm 10 consistently evaluates fewer layouts and finds lower overheads than a random search suggests that the search space has a topology that can be exploited. Specifically, the search space contains locally optimal solutions that are often also globally optimal or near-optimal. Although using BFS requires evaluating significantly fewer layouts than an exhaustive search, the overhead for thousands of layouts are still evaluated, which may not be practical for a compiler.

## 6.2 Greedy Construction

The results from the best-first search shows that many memory layouts can be improved incrementally and greedily, to find a memory layout with an overhead within one cycle of the minimum. Thus, it may be possible to also *build* a low-overhead memory layout greedily. Let $m$ be the number of variables currently in a memory layout $M$. Let $n$ be the number of *variables* in the access sequence $S$. A memory layout can be built as shown in Algorithm 12. $M$ is initially empty, and each iteration of the algorithm selects a variable $v$ that appears in $S$, but not in $M$. $v$ can be inserted in $m + 1$ locations in $M$. The overhead of inserting $v$ at each location is evaluated and the insertion with the lowest overhead is committed. The process continues until all variables are inserted into $M$ exactly once. There are $m$ variables to be inserted, and each insertion requires up to $m$ overhead evaluations, so the algorithm has a running time of $O(m^2)$.

**Algorithm 12** Greedy Layout Construction

```
Input: AccessSequence S
Variable V
MemoryLayout L,Ltemp,Lmin
Index I
for all V ∈ S do
  if L.contains ( V ) then
    continue
  end if
  Lmin ← NULL
  for I : 0 → L.length() do
    Ltemp ← L
    Ltemp.insertAt ( V, I )
    if Ltemp.overhead() < Lmin.overhead() then
      Lmin ← Ltemp
    end if
  end for
  L ← Lmin
end for
return L
```

As with many greedy algorithms, the problem with Algorithm 12 is that greedy decisions are locally optimal and not necessarily globally optimal. For example,

| Access Sequence | Minimum Overhead | Greedy Overhead |
|---|---|---|
| iir_arr | 5 | 5 |
| iir_arr_swp | 6 | 8 |
| latnrm_arr_swp | 6 | 8 |
| latnrm_ptr | 6 | 6 |
| latnrm_ptr_swp | 6 | 7 |

Table 6.5: Efficiency of the greedy algorithm for generating layouts for C-GOA problems.

| Access Sequence | Minimum Overhead | Greedy Overhead |
|---|---|---|
| iir_arr | 4 | 5 |
| iir_arr_swp | 6 | 9 |
| latnrm_arr_swp | 6 | 7 |
| latnrm_ptr | 6 | 6 |
| latnrm_ptr_swp | 6 | 6 |

Table 6.6: Efficiency of the greedy algorithm for generating layouts for GOA problems.

suppose the optimal layout does *not* have variable $y$ between variables $x$ and $z$. At some point in the algorithm before $y$ is inserted, the layout [x z] may have minimal overhead. It is possible that inserting $y$ to generate [x y z] also has a minimal overhead at this point. However, regardless of how variables are subsequently inserted in the layout, the final layout will be sub-optimal because $y$ appears between $x$ and $z$.

Regardless of the inherent drawbacks of a greedy algorithm, Algorithm 12 is used to generate layouts for the access sequences in Table 4.1. The resulting layouts have an overhead ranging from minimum to 50% higher than minimum, as shown in Tables 6.5 and 6.6.

## 6.3 AIG Path Cover

Current algorithms used to reduce address-computation overhead fail to be effective because they are designed using assumptions that are not true in the network-flow models. In this section, a new algorithm and data structure are presented to generate a memory layout. The algorithm and data structure are motivated by the network-

‘A B A C D C D E E E D E D’

Figure 6.1: An access sequence.

flow graph presented in Section 3.2. A memory layout is built by finding a path cover on a data structure called an *Augmented Interference Graph* (AIG), which captures information about how pairs of variables are accessed in relation to one another. For the rest of the section, the access sequence shown in Figure 6.1 will be used to demonstrate how the AIG is constructed and used to generate a memory layout.

### 6.3.1 Variable Access Patterns

Recall that in the network-flow graph, an edge represents the potential for an address register to access two variables in memory. Every pair of variables in the memory layouts has a set of directed edges between them in the network-flow graph that dictate how a single address register would access the two variables. Four different patterns to access two variables are identified and used in the algorithm to identify which pairs of variables should be adjacent in memory. Let $x$ and $y$ represent two variables in memory. The four classifications of access patterns to the variables are:

- Strong pass-through (SP). All accesses to one variable occur before all accesses to the second variable, *i.e.*‘x x y y’. Such a sequence of accesses allows an address register to “pass through” memory locations by performing all required accesses to one location and then permanently moving to the next memory location.

- Weak pass-through (WP). The lifetimes of two variables partially interfere with each other; if one variable $x$ is accessed first in the sequence, then the other variable $y$ is accessed last, *i.e.*‘x y x y’.

- Strong return (SR). All accesses to one variable occur between two consecutive accesses to the second variable, *i.e.*‘x y y x’. In this access sequence,

an address register can access one variable in memory, move to another variable, but eventually "return" to the original memory location.

- Weak return (WR). An access to one variable $x$ is followed by accesses to both variables $x$ and $y$. Eventually, the address register accessing to the two variables must return to, and access, $x$, *i.e.*'`x y x y x`'.

The strong pass-through and strong return relationships are particularly interesting. If multiple variables all have SP relationships with each other, then there is an ordering of variables in memory that allows an address register to access all variables without incurring overhead. Moreover, if adjacent variables in memory have an SP relationship, assumptions can be made about the pattern of accesses made by an address register. In particular, an address register can access one variable, continue to the next and never have to access the first variable again. Similarly, if adjacent variables in memory have an SR relationship, the address register can access one variable, move to the next, and will eventually access the first again.

### 6.3.2 Augmented Interference Graph

The relationships between all variables can be represented as an *Augmented Interference Graph* (AIG) $G = (V, E)$. Each vertex in $V$ represents a variable and each directed edge $(u, v)$ in $E$ represents a pair of variables where $u$ occurs before $v$ in the access sequence. Each edge is also augmented with type information $(u, v)_T$ to denote one of the four identified relationships. Figure 6.2 shows an example of an AIG for the access sequence shown in Figure 6.1. Each of the five variables represent a vertex in the AIG. There are four types of edges representing the four possible access patterns between each pair of variables.

Although any path in $G$ can represent a memory layout, certain paths can represent variables accessed by a single address register without overhead. Specifically, there are four types of paths that represent memory layouts that can be traversed in memory in a predictable pattern:

- P-paths. Paths that are only composed of pass-through edges, with no two consecutive edges being WP edges. An address register can access all vari-

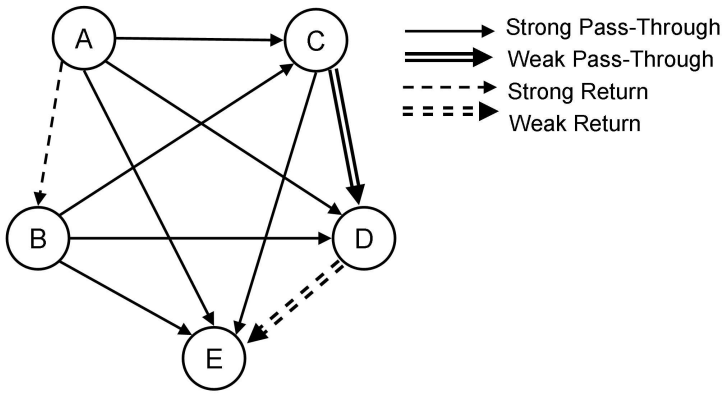



Figure 6.2: An Augmented Interference Graph (AIG) for the access sequence in Figure 6.1.

ables in the path by making one sweep through the memory layout, using only single-word post-increment (or post-decrement) addressing modes. An example of a P-path, formed by '`A C E`', is highlighted in Figure 6.3.

- R-paths. Paths that are only composed of SR edges. The last edge in the path can be an edge of type WR. An address register can access all variables in the path by stepping over the memory layout twice. Post-increment (or post-decrement) addressing can be used to access the first half of all accesses to each variable, followed by using post-decrement (or post-increment) addressing to access the second half of accesses to each variable. An example of an R-path, formed by '`A B`', is highlighted in Figure 6.4.

- PR-paths. Paths that are composed of a P-path appended with an R-path. After an address register steps through a memory layout formed by a P-path, the same register can be used to access a layout formed by an R-path. An example of a PR-path is highlighted in Figure 6.5. Variables '`A D`' form a P-path and variables '`D E`' form an R-path. The R-path is appended to the end of the P-path to form the PR-path.

- PR'-paths. Paths that are composed of a P or PR path, sharing the same root vertex with an R-path. After an address register accesses a memory layout

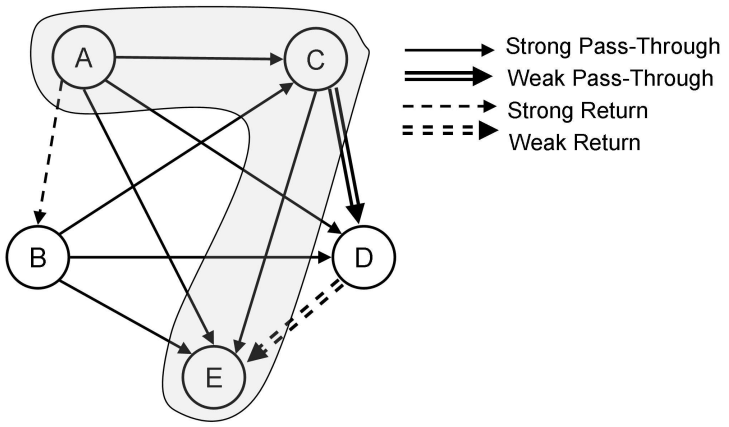



Figure 6.3: A P-path is a directed path composed of *pass-through* edges.

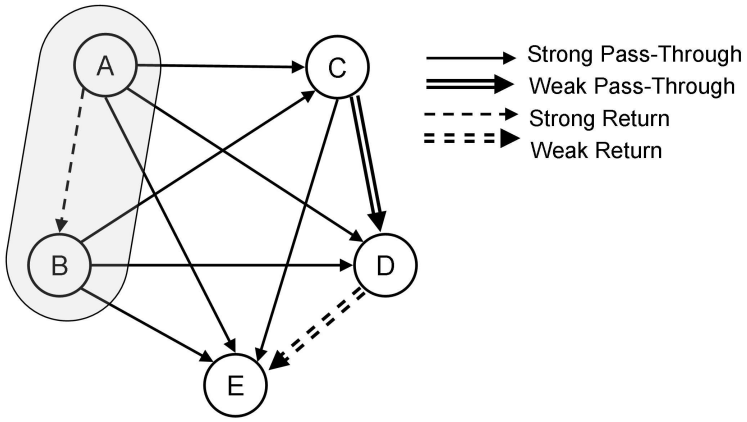



Figure 6.4: An R-path is a directed path composed of *return* edges.

formed by an R-path, the register will return to the first memory location where it can begin accessing the memory layout formed by a P or PR-path. An example of a PR'-path is highlighted in Figure 6.6. Variables '`A E`' form a P-path and variables '`A B`' form an R-path. Both paths start from the same variable and are combined to form a PR'-path.

Additionally, when combining paths to form a PR-path or PR'-path, the newly formed path must not have two incident edges that are both weak-type edges, such as the path '`C D E`'.

In order to facilitate variables being accessed by two address registers, the con-

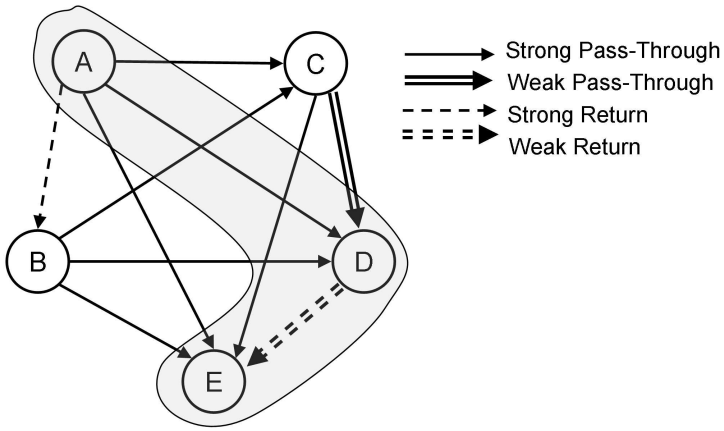



Figure 6.5: A PR-path is a directed path composed of a P-path, followed by an R-path.

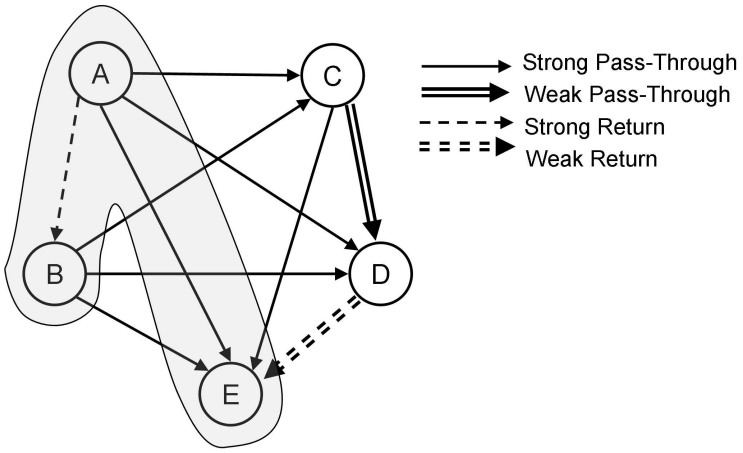



Figure 6.6: A PR'-path is composed of a PR-path and R-path, using the same root.

cept of *splitting* is introduced. A sequence of accesses to a variable can be conceptually divided such that the first half of accesses to a variable is performed through one address register, while the second half of accesses can be performed by another. As a result, all accesses to variable $v$, are replaced by two *split-variables* $v_1$ and $v_2$. When generating an AIG, split-variables can be treated as regular variables; however, in order for an AIG path cover to represent a memory layout, the two halves of a variable that has been split can only appear at the endpoint of two disjoint paths, or only one of the two halves can be in a path cover.

Additionally, the AIG can accommodate C-GOA problem instances. In the C-GOA problem, two variables cannot be accessed simultaneously by the same address register. Two simultaneously accessed variables cannot appear in the same path because each path in a AIG represents an access sequence for a single address register. To ensure that two simultaneously accessed variables are not in the same path, edges between simultaneously accessed variables are simply removed from the AIG.

### 6.3.3 Minimum Path Cover for AIGs

A path in an AIG $G$ represents a sequence of zero-overhead variable accesses by a single address register, as well as a memory layout. Thus, one method to find a memory layout with minimal overhead, is to find a minimum path cover on $G$. That is, to find the minimum number of paths (as described in Sub-section 6.3.2) that covers the vertices of $G$. In general, the minimum path cover problem is NP-complete (as it is reducible to the Hamiltonian Path problem) [5]; however, the problem is solvable in polynomial time for directed, acyclic graphs [5]. Unfortunately, due to the additional constraints imposed by path types and split-variables, traditional minimum path cover algorithms are not applicable to the AIG.

One algorithm to generate a path cover of $G$ is to iteratively cover the vertices of $G$ using greedily selected paths, as presented in Algorithm 13. On each iteration of the algorithm, the longest path (as defined by the rules in Sub-section 6.3.2) from a given variable is found using a depth-first search. The order in which variables are picked to start a path cover is based on the order the variables appear in the access

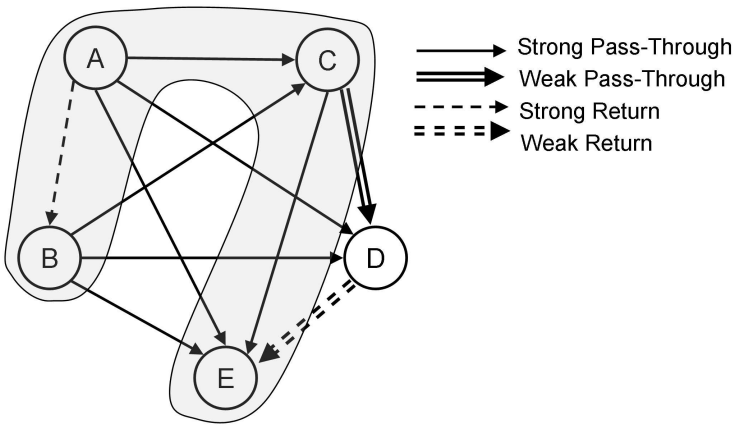



Figure 6.7: The longest path cover found for the given AIG.

sequence.

The algorithm used to find a path using depth-first searches is presented in Algorithm 14. Given a variable $v$, the algorithm recursively finds and traverses all valid outbound edges. When no more edges can be followed, the current path and its length are noted. The algorithm returns the longest P-path found. A second version of the algorithm identifies the longest R-path. The algorithm to find an R-path is executed on two different starting variables: one execution of the algorithm is used to find the longest R-path starting from $v$. The second execution finds the longest R-path starting from the last variable in the P-path, to extend the P-path (forming a PR-path). In Figure 6.2, the longest P-path, among others, is '`A C E`'; while the longest R-path is '`A B`'.

After both paths are identified, they can be combined to form a single PR'-path if they have only the variable $v$ in common. Otherwise, the longer of the two paths is selected to form a path cover. Figure 6.7 shows how the longest P-path ('`A C E`') and longest R-path ('`A B`') can be combined to form a single path.

The algorithm continues to find longest paths and add them to the AIG path cover until there are two or fewer unselected variables remaining. Two variables can always be accessed by a single address register without jump overhead, but using an additional address register requires initialization overhead. Instead, the algorithm attempts to insert the remaining variables into existing path covers. For

**Algorithm 13** AIG Path Cover

```
Input: AccessSequence S, AIG G
Variable V
Path P,R
Cover C
for each V ∈ S, V ∉ C do
   P ← AIG-DFS-P ( G, V )
   R ← AIG-DFS-R ( G, P.LastVariable )
   if P,R share exactly 1 variable then
      P ← P ∪ R
   end if
   R ← AIG-DFS-R ( G, V )
   if P,R share exactly 1 variable then
      C.add ( P ∪ R )
   else if P is longer than R then
      C.add ( P )
   else
      C.add ( R )
   end if
   if |G - C| ≤ 2 then
      break
   end if
end for
Boolean B ← false
for each V ∈ S, V ∉ C do
   Split-Nodes V1,V2
   if can-split( V ) then
      B ← true
      V1,V2 ← split( V )
      G.remove( V )
      G.add( V1, V2 )
   end if
end for
if B then
   restart algorithm
end if
C.add( remaining-variables )
C.merge-all-paths()
return C
```

**Algorithm 14** AIG Depth-First Search

```
Input: AIG G, Variable V
(Global) Path P,T
(Global) LastEdgeType E
T.append ( V )
if |T| > |P| then
   P ← T
end if
for each SP-edge or WP-edge (V,U) ∈ G do
   if E = WP then
      continue
   end if
   E ← TypeOf( V, U )
   G.remove ( V )
   AIG-DFS-P ( G, U )
end for
return P
```

example, in Figure 6.7, instead of simply assigning $D$ to another address register, the algorithm attempts to find a new cover by modifying the access sequence and AIG.

If an incomplete path cover is found, the unassigned variables are considered for splitting. All accesses to a variable $v$ are split into two non-empty sequences of accesses, $v_1$ and $v_2$, such that all accesses in $v_1$ occur before the accesses in $v_2$. Let $v_{idle}$ be the set of accesses to other variables between the last access in $v_1$ and the first access in $v_2$. The accesses to $v$ are split such that $|v_{idle}|$ is maximal. The purpose of splitting is to present an opportunity for $v$ to be accessed by two different address registers. If the maximum $|v_{idle}|$ value is small, the address register used to access $v_1$ can simply be used to access $v_2$, defeating the purpose of splitting. Thus, variables with $v_{idle} \leq 2$ are considered *unsplittable*. To keep the running-time complexity of the algorithm low, variables are only split once. That is, after splitting variable $v$ into $v_1$ and $v_2$, both $v_1$ and $v_2$ are marked as *unsplittable* for the remainder of the algorithm. For the access sequence in Figure 6.1, the unassigned variable $D$ is split into two groups of accesses, $D_1$ and $D_2$ to produce Figure 6.8. After splitting variable $D$, a new AIG can be generated, as shown in Figure 6.9.

After variables are split, the path cover algorithm is restarted. For all successive

‘A B A C D C $D_1$ E E E $D_2$ E $D_2$’

Figure 6.8: An access sequence based on Figure 6.1, but with variable D split into $D_1$ and $D_2$.

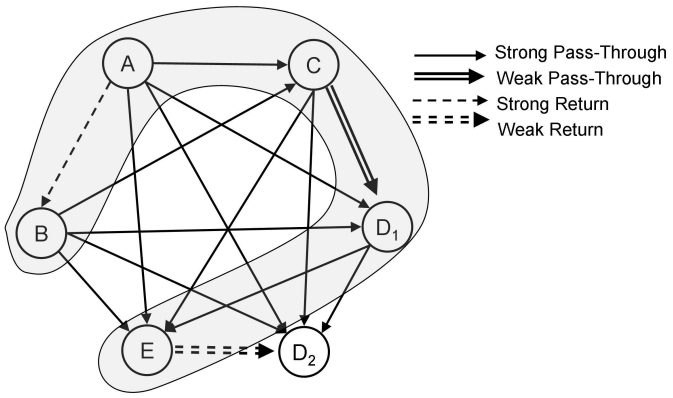



Figure 6.9: The AIG and path cover for the access sequence in Figure 6.8.

iterations of the algorithm, the constraints related to split-variables in paths must be observed, as mentioned in Sub-section 6.3.2. Specifically, two halves of a variable that has been split cannot appear in the middle of two different paths. However, the variables may appear at the endpoints of two different paths. The algorithm continues to split variables until either, a path cover on the entire graph is found, or there are no splittable variables remaining. If there are no more variables that can be split, any variables not assigned to the path cover form a separate path. When the algorithm is restarted for Figure 6.9, a new path is found using variable $D_1$, but variable $D_2$ remains unassigned. $D_2$ is marked *unsplittable* and cannot be split again because variable $D_2$ was generated by splitting $D$. With all variables assigned to a path cover, or marked as unsplittable, the algorithm proceeds to construct the final memory layout.

To construct the final memory layout, two paths that have split nodes from the same variable are joined. All other paths can be considered independent. However, the network-flow techniques for address-code generation ultimately determine the code’s overhead, and may assign accesses to address registers in a much different fashion than intended by the AIG path cover. Thus each path in the path cover is

connected to form a single path, representing a memory layout. Paths that have endpoints with SP or SR edges between them are connected first. Remaining paths are connected arbitrarily. For the example in Figure 6.9, only one path was formed and all variables are used in the path; thus the algorithm terminates with the memory layout [`B A C D E`].

## 6.4 Efficiency of Alternative Algorithms

Tables 6.7 and 6.8 present a comparison of overhead values for the different approaches presented in this chapter. The minimum overhead value for the C-GOA and GOA problem instances for each kernel in Table 4.1 is obtained by the exhaustive evaluation in Chapter 4. For these problem instances, using a best-first search to find a memory layout with a locally minimum overhead frequently produces a memory layout that has a globally minimum overhead. However, finding such layouts requires computing the overhead of thousands of memory layouts. Alternatively, finding a path cover for an AIG requires multiple iterations of Algorithm 13, but only requires computing the overhead once. Unfortunately, the memory layouts produced by this approach have an overhead value near the median value for the entire solution space. The greedy-based algorithm from Section 6.2 appears to be a good compromise between the number of overhead evaluations and quality of memory layouts. As mentioned in Section 6.2, generating a memory layout of $m$ variables using the greedy algorithms requires $O(m^2)$ overhead computations. The greedy algorithm consistently produces layouts that have lower overhead than the AIG path cover, although rarely produces a memory layout with minimum overhead.

For larger access sequences and memory layouts, an exhaustive evaluation is not possible, even in an academic study. However, layouts with locally minimal overhead values found by best-first search are often optimal or near-optimal. Thus, best-first search can be used to approximate the minimum and maximum overhead values for larger access sequences. Four additional GOA and four additional C-GOA problem instances from the UTDSP benchmark suite are identified and eval-

| Access Sequence | Overhead | | | | |
|---|---|---|---|---|---|
| | min | max | AIG cover | Greedy Construction | Best-First Search |
| iir_arr | 5 | 9 | 5 | 5 | 5 |
| iir_arr_swp | 6 | 15 | 9 | 8 | 6 |
| latnrm_arr_swp | 6 | 13 | 8 | 8 | 6 |
| latnrm_ptr | 6 | 13 | 9 | 6 | 6 |
| latnrm_ptr_swp | 6 | 13 | 11 | 7 | 6 |

Table 6.7: Efficiency of the three alternative algorithms for generating a memory layout for C-GOA problem. The minimum and maximum overhead values are found by exhaustive evaluation of the entire offset assignment solution space.

| Access Sequence | Overhead | | | | |
|---|---|---|---|---|---|
| | min | max | AIG cover | Greedy Construction | Best-First Search |
| iir_arr | 4 | 8 | 5 | 5 | 4 |
| iir_arr_swp | 6 | 12 | 9 | 9 | 6 |
| latnrm_arr_swp | 6 | 10 | 10 | 7 | 6 |
| latnrm_ptr | 6 | 10 | 8 | 6 | 6 |
| latnrm_ptr_swp | 6 | 10 | 9 | 6 | 6 |

Table 6.8: Efficiency of the three alternative algorithms for generating a memory layout for GOA problems. The minimum and maximum overhead values are found by exhaustive evaluation of the entire offset assignment solution space.

| Access Sequence | Number of Variables | Overhead | | | |
|---|---|---|---|---|---|
| | | AIG | Greedy Construction | BFS (Min) | BFS (Max) |
| fft_arr (C-GOA) | 24 | 12 | 12 | 8 | 23 |
| fft_ptr (C-GOA) | 16 | 9 | 9 | 6 | 18 |
| iir_ptr (C-GOA) | 14 | 11 | 8 | 7 | 16 |
| iir_ptr_swp (C-GOA) | 14 | 9 | 9 | 7 | 16 |

Table 6.9: Efficiency of the three alternative algorithms for generating a memory layout for C-GOA problems.

| Access Sequence | Number of Variables | Overhead | | | |
|---|---|---|---|---|---|
| | | AIG | Greedy Construction | BFS (Min) | BFS (Max) |
| fft_arr (GOA) | 24 | 12 | 11 | 10 | 22 |
| fft_ptr (GOA) | 16 | 8 | 9 | 6 | 16 |
| iir_ptr (GOA) | 14 | 9 | 8 | 7 | 14 |
| iir_ptr_swp (GOA) | 14 | 11 | 10 | 7 | 14 |

Table 6.10: Efficiency of the three alternative algorithms for generating a memory layout for GOA problems.

uated in Tables 6.9 and 6.10, respectively. In these access sequences, the AIG path cover produces layouts with an overhead that is between 29% and 50% higher than the minimum overhead found by best-first search. Although the memory layouts have overheads that are significantly lower than the worst-case overhead, the quality of memory layouts can be further improved by constructing a memory layout using a greedy-based algorithm. With the exception of the GOA version of the fft_ptr access sequence, constructing a layout greedily (see Algorithm 12) produced memory layouts with overhead values equal to, or less than, the overhead of layouts produced by the AIG path cover.

# Chapter 7

# Conclusions

Traditionally, the problem of minimizing address-computation overhead has been approached by approximating solutions to several problems independently. Of particular importance is the fact that access sequence generation and offset assignment are not formulated or solved with consideration for optimal address-code generation. Algorithms generate access sequences and offset assignments under the assumption that *variables* are assigned to address registers while optimal address-code generation removes this assumption and assigns *accesses* to address registers. Thus, the relationship between access sequence generation, offset assignment, and address-computation overhead has never been accurately explored.

This thesis shows that, despite the availability of algorithms to optimally solve the address-code generation problem, memory layouts still have a significant impact on address-computation overhead. Conversely, changing the access sequence through instruction scheduling appears to have a negligible impact on the final overhead. Specifically, for the benchmark kernels examined, it is always possible to produce memory layouts with very low and very high overhead values, regardless of the instruction schedule. Additionally, the experiments performed in this thesis indicate that current offset assignment algorithms can produce layouts with overhead values that span the full range of possible values in the solution space itself. Thus, in order for current algorithms to generate only low-overhead layouts, a new combinatorial problem, the memory layout permutations problem, must be solved.

The efficiency of current offset assignemtn algorithms are evaluated using five small access sequences. The access sequences were restricted to 12 variables or less

because larger problem instances cannot be exhaustively evaluated in a reasonable amount of time. Although the number of test cases were limited, it is important to note that the access sequences were obtained from the inner loops of commonly run DSP programs, so it is reasonable to assume that the results can be extrapolated to other similar DSP programs.

Layouts generated by different ARA algorithms have different distributions of overhead values. Distributions with fewer memory layouts (due to ARA using fewer ARs) consistently produce more low-overhead layouts. Thus, the average overhead of memory layouts produced by Sugino's ARA algorithm is usually the lowest. When an ARA algorithm uses more address registers, optimal sub-layouts are more easily found. However, locally optimal sub-layouts do not necessarily produce globally optimal memory layouts. For example, there are instances where the naive OFU algorithm produces sub-layouts that can be combined to form optimal layouts, while the branch-and-bound algorithm produces optimal sub-layouts that cannot be combined into optimal layouts.

Conversely, heuristic-based SOA algorithms have very little impact on either layout quantity or quality. However, the minimal differences between the SOA algorithms in this study may be attributed to the small problem sizes. The SOA algorithms are given problem instances with 6 variables or less, and the same path cover is usually found regardless of the algorithm. For GOA problems with 12 variables or fewer, the results of this thesis suggest that an ARA algorithm that generates fewer sub-layouts combined with any SOA algorithm has the greatest chance of producing sub-layouts that combine to form memory layouts with low or minimum overhead.

This thesis shows that regardless of the ARA and SOA algorithms used, placing the resulting sub-layouts contiguously in memory is a necessary optimization problem that must be solved in order to minimize address-computation overhead in a basic block. This new problem is called the memory layout permutations (MLP) problem. The order of sub-layouts in memory has a significant impact on overhead, especially when the number of sub-layouts is high. Additionally, as more variables are assigned to individual sub-layouts, the MLP problem is reduced to the GOA

problem itself. Thus, if an algorithm can be found to address the MLP problem, the same algorithm can be used to solve the GOA problem.

The MLP problem only arises because of the inherent disconnect between traditional offset assignment and address-code generation. An alternative approach to the offset assignment problem is to generate layouts with consideration for the overhead values produced by optimal address-code generation techniques. This thesis explores three alternative algorithms to generate memory layouts. First, a best-first search algorithm is presented to find layouts with locally minimal overhead values in the search space. For small access sequences, these layouts are also optimal layouts. In larger access sequences, these layouts are more often near-optimal layouts. However, the drawback of the best-first search is that executing the search requires evaluating the overhead for several thousands of memory layouts, which may not be feasible in a compiler. The second algorithm presented incrementally builds an access sequence by adding variables in a greedy fashion. To generate a layout of size $n$, the algorithm evaluates the overhead $O(n^2)$ times. Although the algorithm requires significantly less computational time, the algorithm only produced an optimal layout in the two smallest access sequences evaluated. The third algorithm presented generates a memory layout by finding a minimum path cover on an augmented interference graph. The graph is used to summarize how pairs of variables can be accessed by an address register, and the minimum path cover represents how multiple address registers can access variables in memory without incurring *jump* overheads. Unlike the other two alternative algorithms, the path cover approach does explicitly require evaluating the overhead value of any layouts; however, the algorithm also produces the layouts with the highest overhead values among the three alternative algorithms.

In conclusion, there is a large disconnect between the different optimizations used to minimize address-computation overhead. Ultimately, address-computation overhead is dictated by the final addressing code; unfortunately, current algorithms simply do not generate access sequences or memory layouts with consideration for address-code generation. This thesis shows that access sequence generation has a minor impact on the final overhead, while offset assignment is a significant,

but largely unsolved problem. In order for current offset assignment algorithms to be effective, the MLP problem must be solved. Ironically, the MLP problem can be reduced to the GOA problem itself, indicating the ineffectiveness of current approaches. Alternative approaches presented in this thesis are either impractical to implement in a compiler, or only produce layouts with moderately low overhead values.

## 7.1 Future Work

The results presented in this thesis suggest that the offset assignment problem remains largely unsolved. Although current algorithms could be used to generate a memory layout, a solution to the MLP problem is required. Alternatively, memory layouts should be generated to minimize the overhead introduced by address-code generation. One reason the offset assignment problem is so difficult is that the impact of offset assignment on address-computation overhead is not well understood.

After an offset assignment algorithm can be found to reduce overhead for a single basic block, there are two logical extensions to the offset assignment problem. One direction for future work is to examine the impact of offset assignment and optimal address-code generation in loop bodies. This is a particularly important problem because many DSP programs contain loops to continuously iterate and process large amounts of data. The second future direction for this work is to examine the impact of offset assignment and optimal address-code generation at the procedure level; that is, across basic blocks. Although there have been previous research studies to minimize overhead in loop bodies and across basic blocks [3, 28], these studies do not consider the existence of optimal address-code generation. This thesis has already shown that at the basic-block level, optimal address-code generation significantly alters how address-computation overhead can be minimized; thus, further research is required to understand the impact of optimal address-code generation on overhead in loops and procedures.

# Bibliography


[1] Sunil Atri, J. Ramanujam, and Mahmut Kandemir. Improving offset assignment for embedded processors. *Lecture Notes in Computer Science*, 2017:158–172, 2001.

[2] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software – Practice & Experience*, 22(2):101–110, February 1992.

[3] Guilin Chen and Mahmut Kandemir. Optimizing address code generation for array-intensive dsp applications. In *CGO '05: Proceedings of the international symposium on code generation and optimization*, pages 141–152, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Wei-Kai Cheng and Youn-Long Lin. Addressing optimization for loop execution targeting dsp with auto-increment/decrement architecture. In *ISSS '98: Proceedings of the 11th international symposium on system synthesis*, pages 15–20, Washington, DC, USA, 1998. IEEE Computer Society.

[5] Thomas H. Cormen, E. Leiserson, Charles, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[6] Catherine Gebotys. DSP address optimization using a minimum cost circulation technique. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 100–103, Washington, DC, USA, 1997. IEEE Computer Society.

[7] Andrew V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22(1):1–29, 1997.

[8] Johnny Huynh, José Nelson Amaral, Paul Berube, and Sid-Ahmed-Ali Touati. Evaluation of offset assignment heuristics. In *HiPEAC '07: Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, Ghent, Belgium, January 2007. Springer-Verlag.

[9] Mahmut T. Kandemir, Mary Jane Irwin, Guilin Chen, and J. Ramanujam. Address register assignment for reducing code size. In *CC '03: Proceedings of the 12th international conference on Compiler Construction*, pages 273–289, 2003.

[10] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: a Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[11] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids*. Dover Publications, 1976.

[12] Corinna Lee and Mark Stoodley. UTDSP benchmark suite. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, 1992.

[13] Rainer Leupers. Offset assignment showdown: Evaluation of dsp address code optimization algorithms. In *CC '03: Proceedings of the 12th international conference on Compiler Construction*, pages 290–302, 2003.

[14] Rainer Leupers, Anupam Basu, and Peter Marwedel. Optimized array index computation in DSP programs. In *Asia and South Pacific Design Automation Conference*, pages 87–92, 1998.

[15] Rainer Leupers and Fabian David. A uniform optimization technique for offset assignment problems. In *ISSS '98: Proceedings of the 11th international symposium on System synthesis*, pages 3–8, Washington, DC, USA, 1998. IEEE Computer Society.

[16] Rainer Leupers and Peter Marwedel. Algorithms for address assignment in DSP code generation. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 109–112, 1996.

[17] Stan Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, Massachusetts Institute of Technology, 1996.

[18] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tjiang, and Albert Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, 1996.

[19] Sungtaek Lim, Jihong Kim, and Kiyoung Choi. Scheduling-based code size reduction in processors with indirect addressing mode. In *CODES '01: Proceedings of the 9th international symposium on Hardware/software codesign*, pages 165–169, New York, NY, USA, 2001. ACM Press.

[20] Kurt Mehlhorn, Stefan Naher, and Christian Uhrig. The LEDA platform of combinatorial and geometric computing. In *Automata, Languages and Programming*, pages 7–16, 1997.

[21] Desiree Ottoni, Guilherme Ottoni, Guido Araujo, and Rainer Leupers. Improving offset assignment through simultaneous variable coalescing. In *7th International Workshop on Software and Compilers for Embedded Systems*, pages 285–297, 2003.

[22] Amit Rao and Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 128–138, New York, NY, USA, 1999. ACM Press.

[23] Nobuhiko Sugino, Satoshi Iimuro, Akinori Nishihara, and Nobuo Jujii. DSP code optimization utilizing memory addressing operation. *IEICE Trans Fundamentals*, (8):1217–1223, Aug 1996.

[24] Texas Instruments. *TMS320C54X DSP Reference Set: CPU and Peripherals*, 1991.

[25] Sathishkumar Udayanarayanan. Energy efficient code generation for DSPs. Master's thesis, Arizona State University, 2000.

[26] Bernhard Wess and Thomas Zeitlhofer. On the phase coupling problem between data memory layout generation and address pointer assignment. In *SCOPES*, pages 152–166, 2004.

[27] Bernhard R Wess and Martin Gotschlich. Minimization of data address computation overhead in dsp programs. In *Proc. ICASSP98*, pages 3093–3096, 1998.

[28] Youtao Zhang and Jun Yang. Procedural level address offset assignment of dsp applications with loops. *32nd International Conference on Parallel Processing*, 00:21, 2003.

[29] Xiaotong Zhuang, ChokSheak Lau, and Santosh Pande. Storage assignment optimizations through variable coalescence for embedded processors. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 220–231, New York, NY, USA, 2003. ACM Press.