Evaluation of Offset Assignment Heuristics

Johnny Huynh, José Nelson Amaral, Paul Berube, Sid-Ahmed-Ali Touati

Abstract

In digital signal processors (DSPs) variables are accessed using k address registers. The problem of finding a memory layout, for a set of variables, that minimizes the address-computation overhead is known as the *General Offset Assignment* (GOA) Problem. The most common approach to this problem in the literature is to partition the set of variables into k partitions and to assign each partition to an address register. Thus effectively decomposing the GOA problem into several *Simple Offset Assignment* (SOA) problems.

The complementary problem of finding the addressing code that minimizes address-computation overhead for a *fixed* memory layout and a fixed instruction schedule has been solved by Gebotys in [6]. This paper implements Gebotys' solution using an integer linear programming formulation. To find the memory layouts that have the minimum address-computation overhead, the overhead for *all* possible memory layouts for a given sequence of instructions can be computed. Since the number of possible memory layouts grow exponentially, we can only find the memory layout with minimum overhead for access sequences with less than 12 variables. The quality of the solutions obtained with heuristic-based algorithms proposed in the literature [11, 14, 18, 22] are then compared with the set of all possible solutions.

1 Introduction

The extensive use of data in digital signal processing applications requires frequent memory accesses. Many digital signal processors (DSPs) provide dedicated address registers to facilitate the access of variables stored in memory through indirect addressing modes. These addressing modes often support post-incrementing and post-decrementing of the address stored in the address register. Post-incrementing and post-decrementing allows the processor to update the address register without additional cost. Thus, the placement of data in memory affects how effectively the post-increment or post-decrement addressing modes can be used. This placement is also called a memory layout. When two subsequent memory accesses indexed by the same address register are not adjacent in the memory layout, an extra address-computation instruction is required.

Given a memory layout and an instruction sequence, Gebotys' network-flow solution finds the optimal usage of address registers to access the data [6]. Although the technique minimizes the address computation overhead according to a fixed memory

layout, the initial memory layout greatly affects the final code performance. We discovered that even in small test cases that access 12 variables, some memory layouts require addressing code with twice as much address-computation overhead as other memory layouts.

Several heuristic algorithms have been proposed to generate a memory layout that minimizes the address computation overhead [12, 14, 18, 22]. These algorithms assume that all accesses to a single variable use the same address register. Thus, the variables accessed in a sequence of instructions are first partitioned, and each partition is assigned to an address register. We call this partitioning problem the Address Register Assignment (ARA) problem. Individual sub-layouts are generated for each variable partition by approximating a solution to the Simple Offset Assignment (SOA) problem [14]. The ordering that these sub-layouts are concatenated into a single layout is non-trivial and can have a significant impact on the solution generated by the network-flow technique. We call this ordering problem the Memory Layout Permutation (MLP) problem.

Although all the proposed algorithms find a memory layout using similar techniques, there are very few comparisons of the algorithms [12, 22]. Furthermore, the address-computation overhead of the memory layouts produced has only been measured using the cost models of the heuristic algorithms, and not by an optimal technique such as Geboty's network-flow formulation. The experiments reported in this paper show that different orderings of sub-layouts have the most significant impact on address-computation overhead, producing layouts with overhead ranging from optimal to very sub-optimal. Conversely, using different algorithms for the ARA and SOA problems does not significantly impact the overhead of the resulting memory layouts.

The main contributions of this paper are:

- a demonstration that existing heuristic solutions to the GOA problem produce poor approximations to the minimization of address-computation overhead;
- the formulation of a new optimization problem, the memory-layout permutation problem, that has to be solved in order to use a minimum-cost circulation (MCC) technique to evaluate the minimum address-computation overhead incurred in memory layouts produced by heuristic solutions to GOA;
- an experimental evaluation, based on the MCC technique, of heuristic-based ARA and SOA algorithms.

This paper is organized as follows. Sections 2 and 3 present the background to the offset assignment problem and a motivating example. Section 4 discusses how the address-computation overhead of a memory layout can be computed. Current algorithms used to find memory layouts are proposed in Section 5. The experimental evaluation of offset assignment algorithms is presented in Section 6. Finally, related work and conclusions are presented in Sections 7 and 8.

2 Background

2.1 Processor Model

Most DSPshave a set of address registers that is used to access variables stored in memory. Post-incrementing and post-decrementing addressing modes allow an address register r to access a variable v and modify the content of r by one word in the same instruction. Thus, if the next variable accessed using r is either v, the variable immediately before v, or the variable immediately after v, then r can be updated without any additional cost. However, if r has to access a variable that is farther from v, then an explicit address computation is necessary to update r. This additional address computation increases the code's final address-computation overhead.

The amount of computation overhead required to initialize or update an address register, through an address-computation operation, depends on the actual DSP. In the formulation of the address-register allocation and offset-assignment problems, these costs are parameterized by INIT and JUMP.

2.2 The Offset Assignment Problem

Given a set of variables stored contiguously in memory, the *memory layout* of these variables is the ordering of the variables in memory. Each basic block in a program accesses n variables. The order in which these variables are accessed by the instructions in the basic block defines an *access sequence*. The *Offset-Assignment Problem* can be stated as follows:

> Given k address registers and a basic block that accesses n variables, find a *memory layout* such that the address-computation overhead is minimum.

Memory layouts with minimum address-computation 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 as the *General Offset Assignment* (GOA).

In the Simple Offset-Assignment (SOA) problem, a single address register is available to access all the variables in the memory. Liao *et al.* [14] convert the access sequence to an *access graph*. The vertices of this undirected graph are the variables and the weight of the edges indicate the number of times two variables are adjacent in the access sequence. Liao *et al.* [14] show that the SOA problem can be solved by finding the maximum-weight path cover (MWPC) of the access graph. Because the MWPC problem is NP-Complete, they propose a heuristic to solve SOA in polynomial time (see section 5.1).

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 then be found for each sub-sequence. The sub-layouts cannot be computed independently from one another because a variable may appear in multiple address registers. Nonetheless, the union of all sub-layouts must still form a contiguous

layout. Liao *et al.* [14] propose to simplify the GOA problem by assigning *variables*, instead of *variable accesses*, to address registers. This simplification produces subsequences that access disjoint sets of variables. A memory layout can be obtained by solving the SOA problem for each sub-sequence. We call the problem of assigning *variables* to address registers the Address-Register Assignment (ARA) problem (see section 5.2).

Figure 1 illustrates the traditional approach to produce a memory layout for the access sequence of a basic block. The sequence of memory accesses for a basic block is produced by the instruction scheduler. Then the ARA problem is solved to produce sub-sequences. Offsets are assigned to these sub-sequences by solving several instances of the SOAproblem. This paper examines All the heuristic-based algorithms for the ARA and SOA problems examined in this paper generate approximate solutions. Alternative techniques to reduce address-computation overhead are discussed in Section 7.

3 Motivating Example

The processor model used in this paper is based on the TI C54X family of DSPs. Initializing an address register (INIT) has a latency of 2 cycles; modifying an address register by more than one word (JUMP) has a latency of 1 cycle. For this example, consider the access sequence with 6 variables, as shown in Figure 2. The offset assignment problem is to find a placement of the 6 variables in memory such that multiple address registers can access the variables with a minimum amount of overhead. We present 4 examples of how the variables can be accessed and the associated overhead of each solution.

The traditional approach to find an offset assignment requires partitioning the variables into disjoint sets. Each set can then be accessed exclusively with a single address register. 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 3. Address registers A_1 and A_2 can independently access the variables, as shown in Figures 4(a) and 4(b). In this example, the address register assigned to each layout must perform one initialization operation and one jump operation. Thus, the address-computation overhead is 3 cycles for each address register – resulting in a total of 6 cycles of overhead.

In the traditional approach to the GOA problem, the sub-layouts in Figure 3 are considered "optimal" for two reasons. First, there does not exist an ordering for each variable subset, {`a,b,c`} or {`d,e,f`}, that has less than 3 cycles of overhead. Second, there does not exist a partitioning of the 6 variables that can produce sub-layouts with a total overhead that is less than 6 cycles. Do these sub-layouts minimize the address-computation overhead of the input access sequence? The answer is no. The problem is that in this solution each set of variables must be accessed by a single address register.

Figure 5 demonstrates that address-computation overhead can be reduced if a variable can be accessed by more than one address register. The memory layouts in Figure 3 can be placed in memory to form the single, contiguous memory layout shown in Fig-

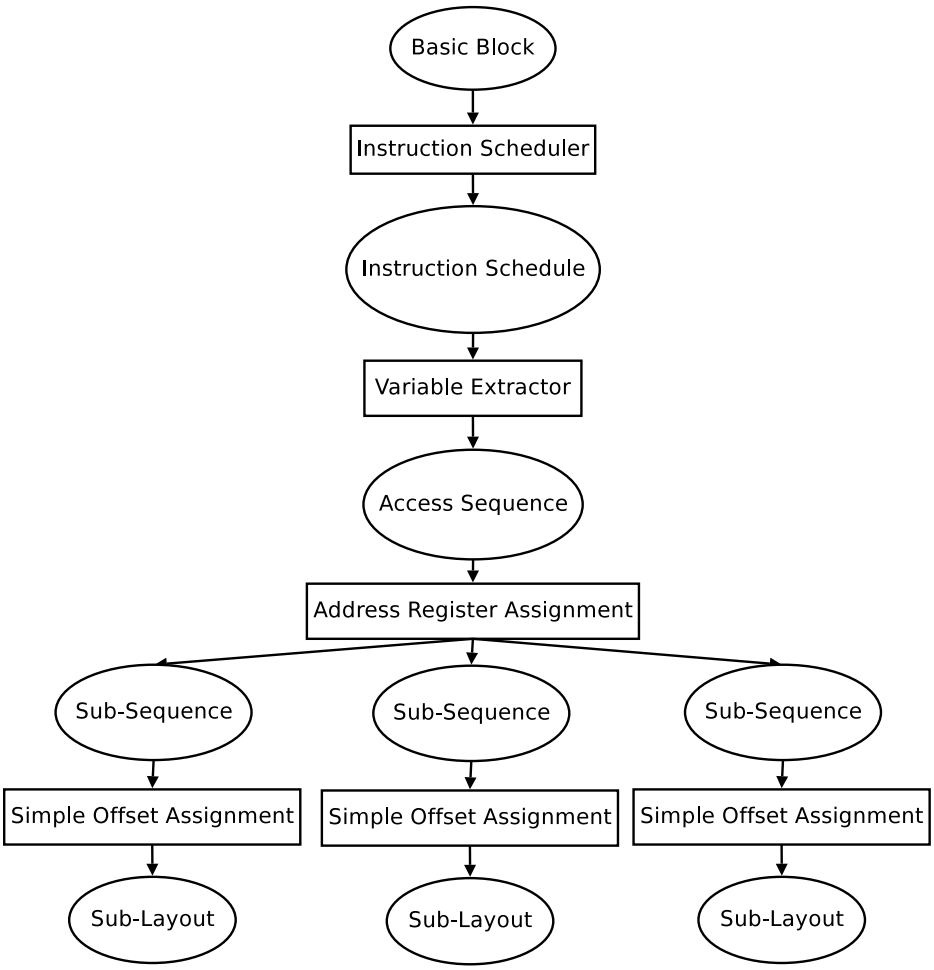
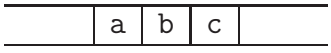


Figure 1: The traditional approach to generate a memory layout for the access sequence of a basic block. The sub-sequences generated by address register assignment access disjoint sets of variables. The resulting set of sub-layouts can then be placed independently in memory to form the final memory layout.

‘a d b e c f b e c f a d’

Figure 2: Memory access sequence



(a) Sub-layout assigned to A_1

(b) Sub-layout assigned to A_2

Figure 3: Two memory sub-layouts

access	instruction	overhead
	$A_1 = \&a$	2
a	$A_1 + = 1$	
b	$A_1 + = 1$	
c	$A_1 - = 1$	
b	$A_1 + = 1$	
c	$A_1 - = 2$	1
a	A_1	

(a) Instructions for address register A_1

access	instruction	overhead
	$A_2 = \&d$	2
d	$A_2 + = 1$	
e	$A_2 + = 1$	
f	$A_2 - = 1$	
e	$A_2 + = 1$	
f	$A_2 - = 2$	1
d	A_2	

(b) Instructions for address register A_2

Figure 4: Instructions for address registers A_1 and A_2

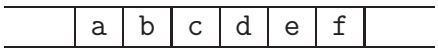
ure 5(a). A_1 is now used for the last access of variable `d` (originally assigned to A_2) without requiring additional overhead cycles. Similarly, A_2 is used for the last access of variable `a` (originally assigned to A_1). This solution has an address-computation overhead of 5 cycles, instead of 6.

Now, consider an alternative sub-layout to 3(b), with the variables ordered as {`b,c,a`}. Variables {`d,e,f`} are kept in the same order. Similar to Figure 4, if variables {`d,e,f`} are assigned to A_1 and variables {`b,c,a`} are assigned to A_2, the total overhead of both address registers is 6 cycles. If the two sub-layouts are placed contiguously in memory, as shown in Figure 6, the minimum address-computation is still 6 cycles.

However, if variables {`b,c,a`} are placed before variables {`d,e,f`}, producing the memory layout shown in Figure 6(a), 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 7(b). Despite the similarities with memory layouts 5(a) and 6(a), the layout in Figure fig:ex-layouto is that only one that allows for the minimum amount of address-computation overhead.

The objective of offset assignment is to minimize address-computation overhead. The traditional approach to solving GOA by using address register assignment and SOA does not minimize overhead because of the restriction that each variable must be accessed exclusively by a single address register. However, the network-flow technique (see Section 4) can reduce the overhead of a memory layout by allowing multiple address registers to access a variable. When the network-flow technique is applied to find a solution to GOA the following questions arise:

- How do different methods of address-register assignment and simple offset assignment affect the overhead of the combined layouts?
- How to arrange a set of sub-layouts in memory to minimize the address-computation costs?
- How to determine whether a given memory layout is optimal?



(a) Layout formed by 3(a) and 3(b)

access	instruction	overhead
	$A_1 = \&a$	2
	$A_2 = \&d$	2
a	$A_1 += 1$	
d	$A_2 += 1$	
b	$A_1 += 1$	
e	$A_2 += 1$	
c	$A_1 -= 1$	
f	$A_2 -= 1$	
b	$A_1 += 1$	
e	$A_2 += 1$	
c	$A_1 += 1$	
f	$A_2 -= 5$	1
a	A_2	
d	A_1	

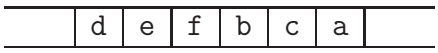
(b) Accessing variables with A_1 and A_2

Figure 5: When the sub-layouts of Figure 3 are placed contiguously in memory, variables `d` and `a` can be accessed by multiple address registers, reducing the address-computation overhead to 5 cycles.

4 Computing Address-Computation Overhead

The example in Section 3 demonstrates that although the address-computation overhead of an access sequence S is influenced by the memory layout M, the overhead is ultimately determined by the addressing code used to access S. The traditional method of finding an addressing code is to assign *variables* to address registers. However, Section 3 also demonstrates that in order to find an addressing code with minimum overhead, multiple address registers need to access the same set of variables. Thus, an *optimal* addressing code for M is an assignment of *accesses* to address registers such that S can be accessed with the minimum possible overhead. In order to evaluate the overhead of memory layouts, the optimal addressing code is required.

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.

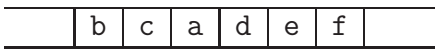


(a) Layout formed using alternative sub-layouts

access	instruction	overhead
	$A_1 = \&a$	2
	$A_2 = \&b$	2
a	$A_1 - = 5$	1
d	$A_1 + = 1$	
b	$A_2 + = 1$	
e	$A_1 + = 1$	
c	$A_2 - = 1$	
f	$A_1 - = 1$	
b	$A_2 + = 1$	
e	$A_1 + = 1$	
c	$A_2 + = 1$	
f	A_1	
a	$A_2 - = 5$	1
d	A_2	

(b) Accessing variables with A_1 and A_2

Figure 6: Placing sub-layouts contiguously in memory does not always guarantee a reduction in overhead. Variables `d` and `a` are accessed by both address registers, but the overhead is still 6 cycles.



(a) An alternative layout with 6 variables

access	instruction	overhead
	$A_1 = \&b$	2
	$A_2 = \&a$	2
a	$A_2 + = 1$	
d	$A_2 + = 1$	
b	$A_1 + = 1$	
e	$A_2 + = 1$	
c	$A_1 - = 1$	
f	$A_2 - = 1$	
b	$A_1 + = 1$	
e	$A_2 + = 1$	
c	$A_1 + = 1$	
f	A_2	
a	$A_1 + = 1$	
d	A_1	

(b) Optimal Usage of A_1 and A_2

Figure 7: An optimal memory layout

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 [8], 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 are guaranteed to be integers as well [6, 8]. Thus, the MCC of a given memory layout can be solved in polynomial time with a linear programming library or solver.

5 Offset Assignment Algorithms

All heuristic-based algorithms that find a memory layout for the GOA problem use the approach illustrated in Section 3. Given an access sequence S and k address registers,

a memory layout can be found by finding answers for three problems: (1) First, the *address-register assignment* (ARA) problem assigns each variable $v \in S$ to a single address register $A_i, 1 \leq i \leq k$ (see Section 5.2). (2) Next, the SOA problem, which is reducible to the *maximum-weight-path cover* problem [14], finds a memory sub-layout m_i for the variables assigned to each address register A_i (see Section 5.1). (3) Last, the *memory-layout permutation* (MLP) problem combines all sub-layouts $m_1 \ldots m_k$ into a single contiguous memory layout (see Section 5.3).

The SOA and ARA problems and their proposed algorithms are presented in Sections 5.1 and 5.2, respectively. The MLP problem, presented in Section 5.3, is not discussed in the literature because all previous formulations of the GOA problem impose the constraint that each variable be accessed exclusively by a single address register.

5.1 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) [14]. They improve the run-time complexity of Bartley's algorithm to approximate a solution to the SOA problem. This improved algorithm marks all edges of the access graph as *removable* and sorts them in decreasing order of weight. If the heaviest removable edge, e, is part of a cycle of *unremovable* edges, edge e is removed; otherwise, e is marked unremovable. If e is incident to two unremovable edges, then all *removable* 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 [12]. 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 [12, 9].

Sugino *et al.* propose a greedy algorithm to approximate a MWPC called ALOMA [18]. The ALOMA algorithm greedily removes edges from the access graph until a path cover is found. Each edge $e = (u, v)$ 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 the 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 Section 6), we also implemented a naive and an optimal algorithm to find a memory layout for the SOA problem. 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 [9, 14]

Liao *et al.* propose an algorithm that finds an optimal layout for the SOA problem using the branch-and-bound technique [13]. 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.

5.2 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 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. We call this problem the Address-Register Assignment (ARA) problem. Since the SOA problem is NP-complete, we conjecture that the ARA problem is NP-hard (since SOA is an instance of ARA).

We examine several algorithms that approximate a solution to the ARA problem. In order to approximate the minimum overhead, an approximation of $C(A_i)$ is required. Any one of the SOA algorithms in Section 5.1 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 [12]. 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 [18]. 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 [22]. 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.

5.3 Memory Layout Combinations

After performing simple offset assignment for the variables assigned to each address register, the variables must be placed in memory, as shown in Figure 8. Address register assignment produces a set of disjoint access sub-sequences that can be treated as independent SOA problems. The memory layout produced by solving each SOA problem is called the ARA sub-layout; that is, a sub-layout resulting from address register assignment. However, finding an ARA sub-layout for each SOA problem involves finding a maximum-weight path cover. By definition, the path cover can be a set of disjoint paths. Each path represents an ordering of variables in memory, which we call the SOA sub-layout. Unless otherwise stated, the term *sub-layout* refers to an SOA sub-layout. An address register accessing one variable at the end of sub-layout will never subsequently access a variable at the end of another sub-layout; if such an access occurred, the two sub-layouts could form a single sub-layout. Thus, the traditional approach to generating a memory layout implies that each sub-layout can be placed independently in memory without affecting address-computation overhead.

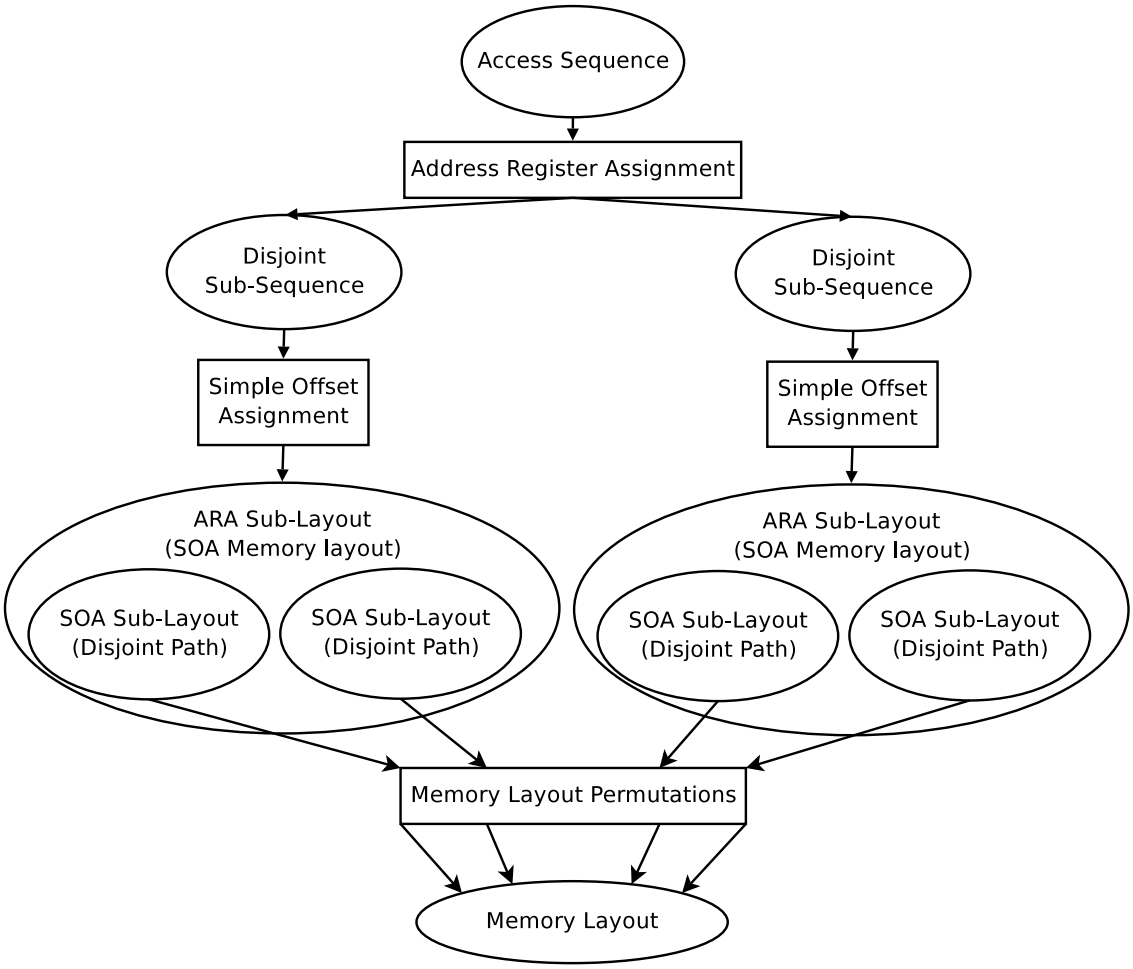


Figure 8: Performing address register assignment followed by simple offset assignment generates memory sub-layouts that must be placed in memory. The problem of finding a placement that minimizes overhead is called the memory-layout permutation problem.

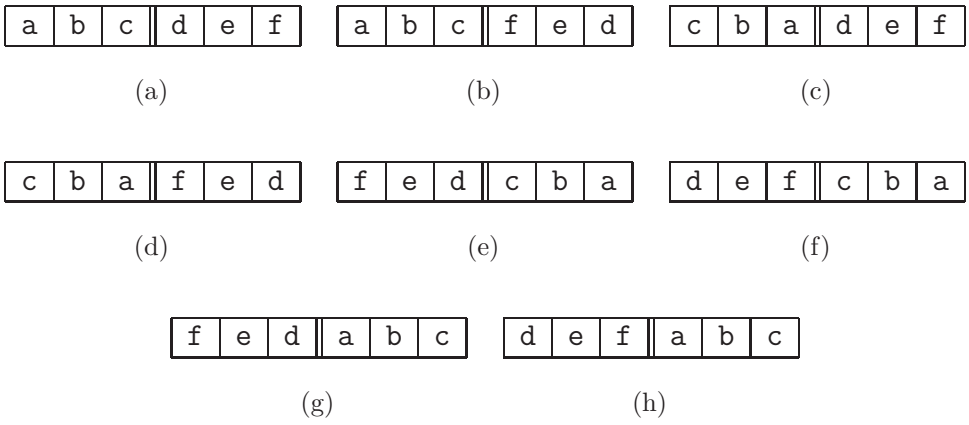


Figure 9: Permutations of two sub-layouts

However, the example in Section 3 demonstrates that if a variable can be accessed by multiple address registers, address-computation overhead may be reduced by placing each memory sub-layout contiguously in memory. Since the MCC technique allows variables to be accessed by multiple address registers, the sub-layouts can 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. We introduce the *memory-layout permutation* (MLP) problem as follows:

> 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 place contiguously in memory.

The solution space for the MLP problem is extremely large. Given m sub-layouts, there are $m!$ permutations of sub-layouts. For each permutation, each sub-layouts can be place in memory in two ways, as M_i or M_i^r. Thus, there are a total of $(m!)(2^m)$ possible memory layouts using m sub-layouts. However, given an ordering of layouts $M_1, \ldots, M_m$, a reciprocal layout can be produced by $M_m^r, \ldots, M_1^r$. The address-computation overhead of a layout is the same as its reciprocal since all variables have the same relative offset to each other. Since each layout has a reciprocal layout with the same address-computation overhead, the MLP solution space is effectively $\frac{(m!)(2^m)}{2}$ memory layouts. Figure 9 shows how 2 sub-layouts can form 8 possible layouts, half of which are reciprocals of another.

An offset assignment problem, as described in Section 2.2, with n variables, has a solution space of $n!$ memory layouts. The existence of reciprocals effectively reduces the solution space to $\frac{n!}{2}$ memory layouts. If we let each variable be a sub-layout, then $m = n$ and the MLP problem is reduced to the offset assignment problem. This implies that if an algorithm can solve the MLP problem, the same algorithm can be used to solve the offset assignment problem.

In practice, a sub-layout usually consists of at least two variables, because a single address register can access any two adjacent memory locations without any JUMP

overhead. Thus, m is usually less than $\frac{n}{2}$. Furthermore, in small GOA problems, address-register assignment may produce SOA problems that result in only a single SOA sub-layout for each ARA sub-layout. In problems where an SOA algorithm only produces a single sub-layout, m is the number of address registers used by an ARA algorithm, which cannot exceed the number of address registers in a DSP. Thus, it may be possible to exhaustively search the entire MLP solution space when the GOA problem has few variables, or the SOA algorithms only produce a single SOA sub-layout (disjoint path). Indeed, exhaustive search is used to find optimal solutions for the MLPproblem in the comparison of offset assignment algorithms in Section 6. For each set of m sub-layouts produced by an ARA and SOA algorithm, the overhead of $\frac{(m!)(2^m)}{2}$ memory layout permutations are evaluated using the MCC technique.

6 Evaluating Offset Assignment Algorithms

This section presents results from an extensive empirical evaluation of the available algorithms that produce approximated solutions to the offset-assignment problem. The main findings of this evaluation are:

- Contrary to the conjectures of other authors [6], the selection of memory layout has a significant impact on the address-computation overhead. Amongst the access sequences examined, less than 0.1% of the possible memory layouts result in the minimum overhead. Thus a system that only solves MCC may result in the generation of sub-optimal code.
- Existing heuristic algorithms seldom produce memory sub-layouts that can be ordered in memory to produce layouts with a minimum address-computation overhead. For some access sequences, none of the algorithms ever produces sub-layouts that can form an optimal solution.
- The choice of algorithm for the ARA problem has significant impact on the quantity and quality of possible memory layouts permutations. On the other hand, the choice of algorithm for the SOA problem has very little impact.

6.1 Experimental Methodology

An outline of the experimental methodology is shown in Figure 10. For each access sequence, a combination of algorithms from Sections 5.1 and 5.2 are used to approximate solutions to the ARA and SOA problems. The 3 ARA and 5 SOA algorithms can be combined to produce 15 heuristic solutions to the offset assignment problem. Each combination produces a set of memory sub-layouts (see Figure 8). If m sub-layouts are produced, then there are $p = \frac{(m!)(2^m)}{2}$ possible memory layout (see Section 5.3). The address-computation overhead of each memory layout is computed using the MCC method described in Section 4. 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.

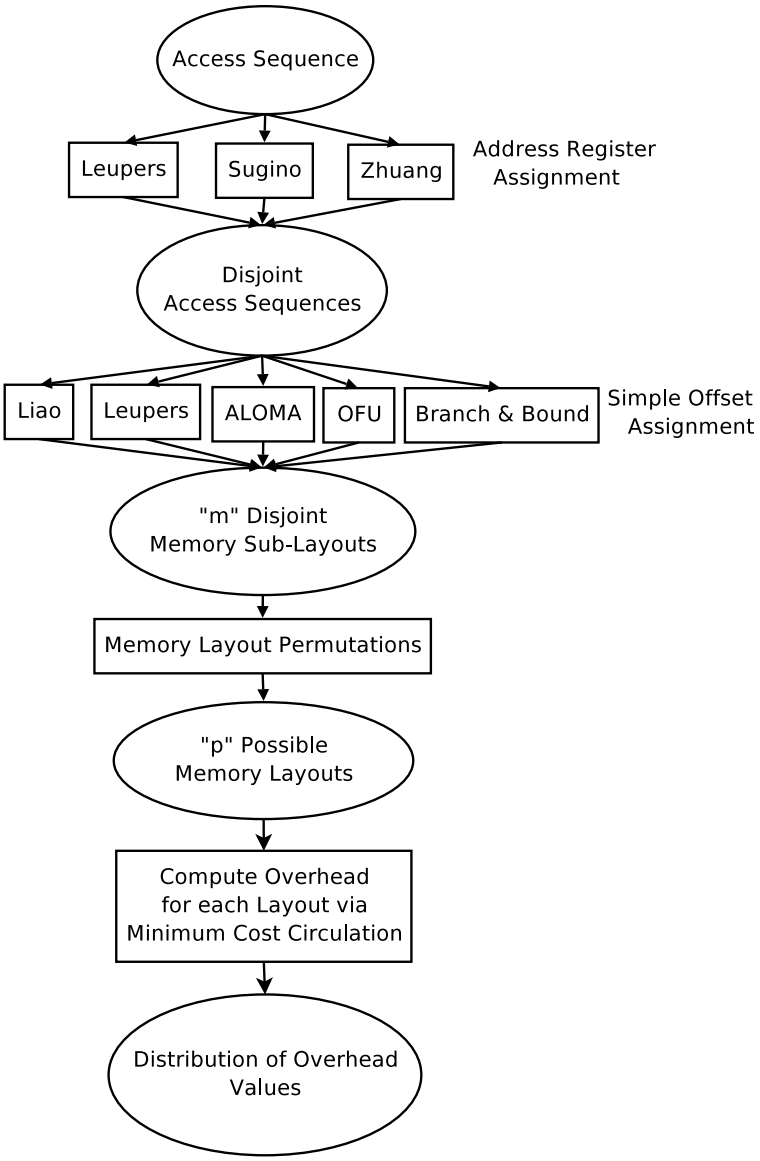


Figure 10: Experimental procedure for evaluating offset assignment algorithms. There are 15 unique paths in the chart, representing the 15 combinations of ARA and SOA algorithms. Let m be the number of sub-layouts produced by each combination. The total number of memory layouts evaluated is then $p = \frac{(m!)(2^m)}{2}$.

Kernel	Length	Variables	No. of Memory Layouts
iir_arr	21	8	20,160
iir_arr_swp	33	12	239,500,800
latnrm_arr_swp	30	10	1,824,400
latnrm_ptr	30	10	1,824,400
latnrm_ptr_swp	30	10	1,824,400

Table 1: Size of problem and solution space for kernels evaluated in the UTDSP Benchmark Suite

6.2 Test Environment

This experimental evaluation uses a processor model based on the TI C54X family of DSPs. This architecture has an overhead of 2 cycles to initialize address registers (INIT) and an overhead of 1 cycle to access non-adjacent memory locations (JUMP).

Given an access sequence with n variables, we compute the optimal memory layout by evaluating the MCC of all possible $\frac{n!}{2}$ memory layouts (see Section 5.3). Comparative experiments are restricted to sequences with up to 12 variables. For $n = 12$ this exhaustive search took over 30 hours and 240MB of disk space on a 14-node dual Opteron 248 cluster.

Access sequences are obtained from several kernels in the UTDSP benchmark suite. Each kernel is compiled with `gcc` version 3.3.2, using `-O2` optimization. The compiler is modified to output the access sequence from the inner-most loop, prior to register allocation. Only five kernels produce access sequences with $n \leq 12$. The access sequences evaluated are presented in Table 1.

6.3 The Efficiency of Offset Assignment Heuristics

Table 2 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 2 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 overhead obtained using the heuristic-based algorithms presented in Sections 5.2 and 5.1 indicate that, in general, they are not very effective. 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 average overhead of all layouts in the solution space. Moreover, the layouts formed by combining sub-layouts produced these heuristic-based algorithms have address-computation overheads that range from the best (minimum) to the worst

(maximum).

6.4 The Efficiency of of ARA Heuristics

This section studies impact of individual ARA algorithms on the address-computation overhead. Each of the three ARA algorithms — Leupers, Sugino, and Zhuang — can be combined with five SOA algorithms (see Figure 10) to generate an algorithm that produces a memory layout. All of the layouts produced by an ARA algorithm are combined into a set and the overhead distributions of these sets of layouts are presented in Table 3. For instance, for the `iir_arr_swp` access sequence, the combination of Leupers' ARA algorithm with the five SOA algorithms produces 204 memory layouts that have an overhead of 8 cycles.

The total number of memory layouts in each column varies because each ARA algorithm can use a different number of address registers which in turn, yields a different number of memory-layout permutations (see Section 5.3). The results in Table 3 indicate that ARA algorithms that produce fewer layouts, such as Sugino's, also tend to produce better layouts. This result indicate that it is better better to user fewer address registers even when more registers are available in the processor. For instance, in the `iir_arr_swp` access sequence, Leupers and Marwedel's ARA algorithm yields 9600 memory layouts. But only 2 layouts have an overhead of 7 cycles. On the other hand, the ARA algorithm proposed by Sugino *et al.* generates 2688 memory layouts with 61 layouts having an overhead of 7 cycles. Similar distributions occur for the other access sequences.

Locally optimal sub-layouts do not lead to globally optimal memory layouts. When more address registers are used by an ARA algorithm, there are less variables assigned 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 address register. Two variables can be trivially accessed without incurring JUMP overhead and can be considered 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.

6.5 The Efficiency of SOA Heuristics

The distributions in Table 4 Are complementary to the distributions in Table 3. Here the focus is on the layouts produced by each of the five SOA algorithms when it is combined with each of the three ARAalgorithms. For instance, for the `iir_aar_swp` sequence the combinations of Sugino's SOA algorithm with the three ARAalgorithms together produce 1,187 layouts have an overhead of 9 cycles.

A SOA algorithm is used to estimate the increase in overhead when assigning variables to address registers. The number of estimations produced affect the number sub-layouts produced by the ARA algorithms. This variation appears in the table as a different total number of layouts in the various column. Low variability between the columns indicates that the SOA algorithms can usually find an optimal or near-optimal SOA solution for the sub-layouts created by the ARA algorithm for that sequence.

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 2: 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.

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 3: 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 naive Order First Use (OFU) algorithm consistently produced more memory layouts. The OFU algorithm generates poor sub-layouts and leads the ARA algorithms to create more variable partitions. However, larger number of sub-layouts do not predicate the success of the SOA algorithm.

The results in Table 4 confirm that combining optimal sub-layouts does not result in optimal layouts. For instance, in the `latnrm_ptr_swp` access sequence, the OFU algorithm generates sub-layouts that can be combined to form optimal memory layouts, while the Branch and Bound (B&B) algorithm, which finds optimal sub-layouts, does not form any optimal memory layouts.

There is no SOA algorithm that consistently produces sub-layouts that can form the most number of optimal or near-optimal layouts. In two access sequences, OFU is best, while in two other sequences, the SOA algorithm proposed by Sugino *et al.* is best.

7 Related Work

In 2003, Leupers presented a comprehensive experimental evaluation of algorithms for the *simple* offset assignment (SOA) problem [9]; however, there has not been an evaluation of algorithms for the *general* offset assignment (GOA) problem. Our comparison of offset assignment problems has three distinguishing features:

- The GOA problem is evaluated as three separate problems: address register assignment (ARA), simple offset assignment (SOA), and memory-layout permutation (MLP).
- All known *heuristic-based* algorithms that generate a *single* approximate solution to the ARA or SOA problems are compared against each other, and against the optimal solutions.
- The minimum address-computation overhead of each memory layout generated is computed using a minimum cost circulation (MCC) technique.

Some algorithms for generating a memory layout were not included in our study. 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 [20]. One reason these algorithms are exclude from the experiments is because their performance is dependent on the initial memory layout produced. Leupers and David propose to find a memory layout for the GOA problem using a genetic algorithm [11], while Wess and Gotschlich propose to generate memory layouts using simulated annealing [11, 21]. One drawback of using a genetic algorithm or simulated annealing is that finding a fast, but accurate, *fitness* function is difficult. Another drawback is that the algorithms require many simulation steps to find a memory layout with a minimum overhead, which may require too much time to be practical in a production compiler.

The original offset assignment problem presented by Bartley focused only on *scalar* variables [2]; thus, this study did not examine algorithms that optimize *array* accesses

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 4: 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.

in loops. Leupers *et al.* propose an algorithm to find a good usage of address registers for array accesses in a loop body [10]. Cheng and Lin also propose an address register allocation algorithm, but also discuss how array data can be reordered to further reduce address-computation overhead [4]. Chen and Kandemir present a scheme to transform arrays and reschedule array accesses to reduce overhead [3].

Although our study only focuses on algorithms that directly generate a memory layout, address-computation overhead can also be reduced by manipulating the access sequence through instruction scheduling and variable extraction (see Figure 1). Rao and Pande propose to apply algebraic transformations (such as commutativity) on expression trees to produce a *least-cost access sequence* [17]; Lim *et al.* propose to manipulate the entire instruction schedule [15]. Kandemir *et al.* propose an algorithm to change the access sequence of variables *after* a full or partial memory layout is formed for each basic block [7]. These three studies focus only on reducing overhead for the SOA problem; however, the optimizations can be applied independently of the offset assignment optimizations. Choi and Kim propose a unified algorithm to find simultaneously find an instruction schedule and offset assignment with low overhead [5]. However, the unified approach still uses an offset assignment algorithm as a sub-routine. Thus, finding an improved offset assignment algorithm does not interfere with scheduling optimizations.

After an instruction schedule is found, the access sequence of variables must be extracted. Ottoni *et al.* propose to simultaneously coalesce variables and find a memory layout for SOA problems [16]. Similarly, Zhuang *et al.* propose algorithms that coalesces variables for both the SOA and GOA problems [22]. Although the coalescing algorithms in both works simultaneously find memory layouts, it is still possible to perform an additional offset assignment pass to further reduce address-computation overhead.

Regardless of the optimizations performed, the ultimate objective of generating a memory layout is to reduce code size and address-computation overhead. Given a memory layout, we computed the *minimum* overhead using a minimum-cost circulation (MCC) technique [6]; however, an alternative approach to find the minimum overhead is to use a minimum-weight perfect matching (MWPM) technique [19]. The running-time complexity of the two approaches can be stated in terms of the length of the access sequence l, and the number of address registers k. The complexity of finding a minimum-weight perfect matching is $O((l+k)^3)$, which is theoretically higher than the $O(l^4\ log\ l)$ time complexity to find a minimum-cost circulation [19]. In practice, k is small and bounded, so the MWPC can be found in less time. However, solutions to both problems can be quickly implemented using a linear program, which makes the complexity of both problems the same. Thus, for our implementation, there was no benefit to using the MWPC approach.

8 Conclusion and Future Work

The minimum cost circulation technique (MCC) produces the optimal addressing code for *fixed* memory layout and access sequence by allowing variables to be accessed by multiple address registers. However, the initial memory layout still has a significant

impact on the address-computation overhead. Current offset assignment algorithms can be used to generate the initial memory layouts. However, by solving offset assignment problems as an address register assignment problem, the algorithms introduce a new combinatorial problem we call the memory-layout permutation problem.

We see that layouts generated by different ARA algorithms have different distributions of overhead values. Distributions with fewer memory layouts (due to less address registers used by the ARA algorithm) consistently produce more layouts with low overheads. Thus, the average overhead of memory layouts produced by Sugino's ARA algorithm was usually the lowest. When an ARA algorithm uses more address registers, it is easier to find optimal sub-layouts. However, locally optimal sub-layouts do not necessarily produce globally optimal memory layouts. We observe problem instances where using the naive OFU algorithm produces naive sub-layouts that can be combined to form optimal layouts. Conversely, using the branch and bound algorithm produces optimal sub-layouts can only be combined to form non-optimal layouts.

Our experiments show that different ARA algorithms have a significant impact on the number and quality of memory layouts produced, while the heuristic-based SOA algorithms have very little impact. However, the minimal differences between the SOA algorithms can be attributed to the small problem sizes. The SOA algorithms are only given SOA instances with 6 variables or less, and the same MWPC is usually found between the different algorithms. Thus, for GOA problems with 12 variables or less, we find that using an ARA algorithm that generates fewer sub-layouts (such as the algorithm proposed by Sugino *et al.*) combined with any SOA algorithm produces sub-layouts that can be most easily combined to form a memory layout with low or minimum overhead.

As shown in this paper, regardless of the ARA and SOA algorithm used, placing the resulting sub-layouts contiguously in memory is necessary in order to minimize address-computation overhead in a basic block. This is what we call the memory-layout permutation (MLP) problem. The placement of sub-layouts has a significant impact on the final memory layout's 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 we can find an algorithm to address the MLP problem, the same algorithm can be used to address the GOA problem itself!

Our experimental study shows evidence that we should explore new directions for the GOA problem. One direction is to explore better ways to solve the MLP problem that we highlighted in this paper. The alternative direction is to avoid addressing the ARA, SOA, and MLP problems separately and find a combined method to generate a memory layout that minimizes the overhead as computed by the MCC technique.

References

[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] Yoonseo Choi and Taewhan Kim. Address assignment combined with scheduling in DSP code generation. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 225–230, New York, NY, USA, 2002. ACM Press.

[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] Mahmut T. Kandemir, Mary Jane Irwin, Guilin Chen, and J. Ramanujam. Address register assignment for reducing code size. In *CC*, pages 273–289, 2003.

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

[9] Rainer Leupers. Offset assignment showdown: Evaluation of dsp address code optimization algorithms. In *CC*, pages 290–302, 2003.

[10] 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.

[11] 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.

[12] 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.

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

[14] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tjiang, and Albert Wang. Storage assignment to decrease code size. *ACM Trans. Program. Lang. Syst.*, 18(3):235–253, 1996.

[15] Sungtaek Lim, Jihong Kim, and Kiyoung Choi. Scheduling-based code size reduction in processors with indirect addressing mode. In *CODES '01: Proceedings of*

the ninth international symposium on Hardware/software codesign, pages 165–169, New York, NY, USA, 2001. ACM Press.

[16] Desiree Ottoni, Guilherme Ottoni, Guido Araujo, and Rainer Leupers. Improving offset assignment through simultaneous variable coalescing. In *S.COPES*, pages 285–297, 2003.

[17] 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.

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

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

[20] 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.

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

[22] 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.