Evaluation of Offset Assignment Heuristics

Johnny Huynh[1]*, José Nelson Amaral[1], Paul Berube[1], and Sid-Ahmed-Ali Touati[2]

[1] University of Alberta, Canada
[2] Université de Versailles, France

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 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. Many heuristic-based algorithms are proposed in the literature to approximate solutions to the partitioning and SOA problems. However, the address-computation overhead of the resulting memory layouts are not accurately evaluated. In this paper we use Gebotys' optimal address-code generation technique to evaluate memory layouts. Using this evaluation method introduces a new problem which we call the Memory Layout Permutation (MLP) problem. We then use the Gebotys' technique and an exhaustive solution to the MLP problem to evaluate heuristic-based offset-assignment algorithms. The memory layouts produced by each algorithm are compared against each other and against the optimal layouts. Our results show that even in small access sequences with 12 variables or less, current heuristics may produce memory layouts with address-computation overheads up to two times higher than the overhead of an optimal layout.

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 (ARs) to facilitate the access of variables stored in memory through indirect addressing modes. Post-increment and post-decrementing addressing modes are often supported, allowing the processor to update the AR in the same cycle a memory location is accessed. When two consecutive memory accesses, indexed by the same AR, are not adjacent in the memory, an extra address-computation instruction is required. Thus, the placement of data in memory affects how effectively the post-increment or post-decrement addressing

* This research is supported by fellowships and grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), the Informatics Circle of Research Excellence (iCORE), and the Canadian Foundation for Innovation (CFI).

modes can be used. This placement is called a memory layout; and the problem of finding a memory layout that minimizes address-computation overhead is called the General Offset Assignment (GOA) problem.

Given a memory layout and an instruction sequence, Gebotys' network-flow solution finds the optimal usage of ARs to access the data [1]. While this technique works well for a fixed memory layout, we discovered that the initial memory layout greatly affects the final code performance. Even in small test cases that access 12 variables, some memory layouts require twice as many cycles for address computations as other memory layouts.

Several heuristic algorithms have been proposed to generate a memory layout that minimizes address-computation overhead [2–5]. The algorithms simplify the GOA problem by assuming that every access to a variable uses the same AR. With this simplification, the GOA problem can be addressed as follows. First, variables accessed in an instruction sequence are partitioned, and each partition is assigned to an AR. We call this partitioning problem the Address Register Assignment (ARA) problem. Next, individual sub-layouts for each partition are generated by approximating a solution to the Simple Offset Assignment (SOA) problem [3]. Last, the sub-layouts can be ordered to form a single, contiguous memory layout. We call this ordering problem the Memory Layout Permutation (MLP) problem, and it arises because the overhead of solutions found by the network-flow technique is sensitive to the ordering of sub-layouts.

Although many algorithms have been proposed to address the GOA problem, only heuristics for the SOA problem have been comprehensively compared [6]. Furthermore, the address-computation overhead of the produced memory layouts has only been measured using the cost models of the heuristic algorithms, and not by an optimal technique such as Gebotys' network-flow formulation. The experiments reported in this paper show that different orderings of sub-layouts have a significant impact on the optimal address-computation overhead, producing layouts with overheads that span the entire solution space. Specifically, the worst layouts have a minimum overhead of twice as many cycles as the optimal layouts. Additionally, 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 poorly approximate the minimization of address-computation overhead;
– the formulation of a new optimization problem, the memory-layout permutation problem, that must 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. Section 2 presents the background to the offset assignment problem and discusses how the address-computation overhead of a memory layout can be computed. Current algorithms used to find memory layouts are presented in Section 3. The experimental evaluation of offset assignment algorithms is presented in Section 4. Finally, related work and conclusions are presented in Section 5 and 6.

2 Background

Many DSPs have a set of ARs used to access variables stored in memory. Post-incrementing and post-decrementing addressing modes allow an AR r to access a variable v and modify the content of r by one word in the same instruction. Thus, if the next access using r is to location v, or to the locations immediately adjacent to v in memory, r can be updated without any additional cost. However, if r accesses a location that is non-adjacent to v, an explicit address computation is necessary. The computational overhead required to initialize or update ARs is architecture-dependent.

All experiments in this paper model the Texas Instruments TMS320C54X family of processors. These DSPs have eight 16-bit ARs. Most instructions are one word in length and have one cycle of overhead. Initializing an AR requires a two-word instruction and two cycles of overhead. Similarly, auto-incrementing (or auto-decrementing) an AR by more than one word requires one extra word to encode the instruction and one extra cycle to execute. Thus, inefficiently using address registers results in an increase in both code size and run-time overheads.

Similar to many other DSP architectures, the address registers in the C54X 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 typically 32-bit values.
2. Address registers can only be manipulated by the address-generation unit, which is limited to addition and subtraction of 16-bit values.

Thus, it may be infeasible to use ARs as general purpose registers, and the offset assignment problem must be solved to effectively place variables in memory.

2.1 The Offset Assignment Problem

Given a set of variables stored contiguously in memory, a *memory layout* is an ordering of these variables in memory. A basic block in a program accesses n variables. The order of variable accesses by the instructions in the 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 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 AR is available to access all the variables in the memory. Liao *et al.* [3] convert the access sequence to an undirected *access graph*. 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.* [3] reduce the SOA problem for an access graph to the NP-Complete maximum-weight path cover problem, and propose a heuristic to solve SOA in polynomial time (see Section 3.1).

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 ARs. This assignment creates multiple access *sub-sequences* — one for each AR. 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.* [3] simplify the GOA problem by assigning *variables*, instead of *variable accesses*, to address registers. This simplification produces sub-sequences 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 3.2).

Figure 1 illustrates the traditional approach to produce a memory layout from a basic block. First, the instruction scheduler emits the sequence of memory accesses. Then, the ARA problem is solved to produce sub-sequences. Offsets are assigned in each sub-sequences by solving several instances of the SOAproblem. 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 5.

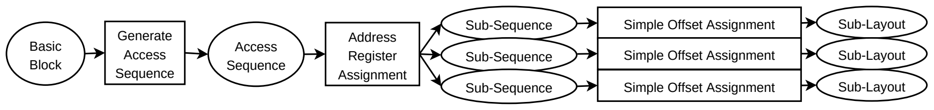


Fig. 1: The traditional approach to generate a memory layout for the access sequence of a basic block. The sub-sequences generated by address register assignment access disjoint sets of variables. The resulting set of sub-layouts can then be placed independently in memory to form the final memory layout.

2.2 Computing Address-Computation Overhead

The traditional approach to finding a memory layout assumes that each variable is accessed exclusively by one AR. Thus, the resulting addressing-code and address-computation overhead are based on assigning *variables* to ARs. However, an *optimal* addressing code for a memory layout, M, is an assignment of *accesses* to ARs such that the access sequence, S, can be accessed with the minimum overhead. In order to accurately evaluate the overhead of memory layouts, optimal addressing code is required.

Gebotys proposes an algorithm to find optimal addressing code [1]. The assignment of *accesses* to ARs can be found by transforming M and S 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 memory layout. The MCC for these network-flow graphs can be solved in polynomial time. In this paper, the MCC technique is used to evaluate the quality of all memory layouts.

3 Offset Assignment Algorithms

Existing heuristic-based algorithms solve the GOA problem as described in Section 2.1. Given an access sequence S, and k ARs, a memory layout is found by:

1. ARA assigns each variable $v \in S$ to a single AR $A_i, 1 \leq i \leq k$.
2. SOA finds a sub-layout m_i for the variables assigned to each AR A_i.
3. The *memory-layout permutation* (MLP) problem combines all sub-layouts $m_1 \dots m_k$ into a contiguous memory layout. MLP is absent from the literature because previous solutions to GOA assigned *variables*, rather than *accesses*, to ARs. The MLP problem only appears when using traditional algorithms in conjunction with the MCC technique.

We use MCC to evaluate the performance of several ARA and SOA algorithms.

3.1 Simple Offset Assignment

The SOA problem was introduced by Bartley and solved as a *maximum-weight Hamiltonian-path* problem [7]. Given an access sequence S, a weighted access graph G, can be constructed. A path in G represents an ordering of variables in memory. Liao *et al.* refine the SOA problem formulation to a *maximum-weight path cover* problem [3], which is NP-complete. Thus, all subsequently proposed algorithms approximate a solution to the SOA problem by finding a path cover on G.

We implement and evaluate five solutions to the SOA problem:

1. Liao *et al.* propose an algorithm that builds the path cover, one edge at a time, by using a greedy heuristic to select edges in G [3].
2. Leupers extends the algorithm by Liao *et al.* by proposing a tie-break function to decide between edges of equal weights [2] [6].
3. Sugino *et al.* propose an algorithm that uses a greedy heuristic to remove one edge at a time from G until a valid path cover is formed [4].
4. Liao and Leupers present a naive algorithm which builds a memory layout based on the declaration order of variables in the access sequence. The algorithm is also known as Order First Use (OFU).

5. Liao also presents a branch-and-bound algorithm that finds the maximum-weight path cover. The algorithm has exponential time-complexity, but for small graphs, our implementation runs in a reasonable amount of time.

3.2 Address Register Assignment

In the GOA problem, $k > 1$ ARs are used to access variables in memory. Liao *et al.* decompose the GOA problem into multiple instances of SOA by assigning each variable to an AR 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 ARs, assign each $v \in V$ to an AR $A_i, 1 \leq i \leq k$, such that $\sum_{i=1}^{k} C(A_i)$ is minimum.

Solving this problem does not produce a memory layout — it is only an assignment of variables to ARs. Thus, this problem should not be considered the *real* GOA problem. We call this problem the Address-Register Assignment (ARA) problem. We conjecture that ARA is NP-hard because SOA is NP-complete and is an instance of ARA.

This paper examines several heuristic-based algorithms for ARA. In each case, an approximation of $C(A_i)$ is required to estimate the overhead of assigning a variable to an AR. Any of the SOA algorithms in Section 3.1 can be used as a sub-routine to approximate $C(A_i)$ for the following ARA algorithms:

1. Leupers and David propose a greedy algorithm that assigns variables to ARs by selecting one edge at a time from the access graph [2].
2. Sugino *et al.* use a heuristic-based algorithm that iteratively partitions the variables and selects the partitioning with the lowest estimated overhead [4].
3. Zhuang *et al.*'s variable coalescing algorithm for offset-assignment problems includes assigning variables to ARs [5]. The assignment portion of the algorithm greedily assigns one variable to an AR until all variables are assigned.

3.3 Memory Layout Permutations

As illustrated in Figure 2, the traditional approach to offset assignment produces a set of disjoint sub-layouts. ARA produces a set of disjoint access sub-sequences that are solved as independent SOA problems. Each SOA instance is solved to produce a memory layout called an ARA sub-layout. However, each ARA sub-layout is formed by the combination of disjoint paths from the SOA path cover. Each disjoint path in the path cover is called an SOA sub-layout and defines an ordering of variables in memory. Unless otherwise stated, the term *sub-layout* refers to an SOA sub-layout. In the traditional approach to offset assignment, each sub-layout is independent, and can be placed in memory arbitrarily.

However, sub-layouts are only considered independent because of the traditional assumption that variables are accessed exclusively by one AR. Since the

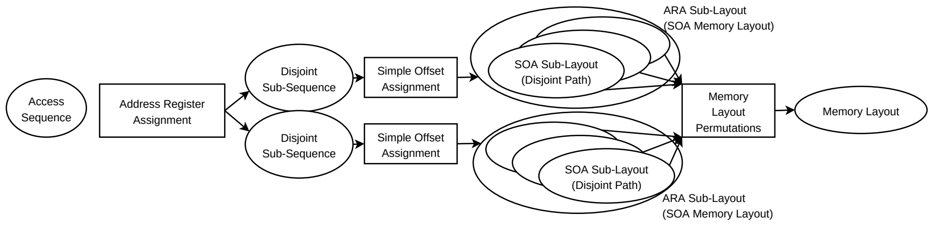

Access Sequence
Address Register Assignment
Disjoint Sub-Sequence
Disjoint Sub-Sequence
Simple Offset Assignment
Simple Offset Assignment
SOA Sub-Layout (Disjoint Path)
SOA Sub-Layout (Disjoint Path)
ARA Sub-Layout (SOA Memory Layout)
ARA Sub-Layout (SOA Memory Layout)
Memory Layout Permutations
Memory Layout

Fig. 2: 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.

MCC technique allows variables to be accessed by multiple ARs, it is possible to reduce address-computation overhead by placing sub-layouts contiguously 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. 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 placed contiguously in memory.

The MLP solution space is extremely large: m sub-layouts can form $m!$ permutations. 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, an ordering of layouts $M_1, \ldots, M_m$ is equivalent to its reciprocal layout, $M_m^r, \ldots, M_1^r$, since all variables have the same relative offset to each other. Thus, the MLP solution space is $\frac{(m!)(2^m)}{2}$ memory layouts. Figure 3 shows how 2 sub-layouts can form 8 possible layouts, half of which are reciprocals of another.

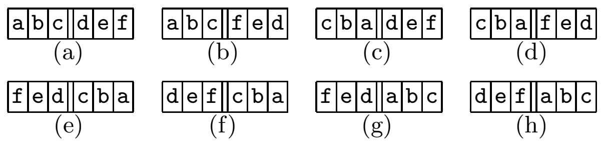


Fig. 3: Permutations of two sub-layouts

When reciprocals are considered, an offset assignment problem with n variables has a solution space of $\frac{n!}{2}$ 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 solves the MLP problem, the same algorithm solves the offset assignment problem.

4 Evaluating Offset Assignment Algorithms

An extensive empirical evaluation of the available heuristic offset-assignment algorithms supports the following conclusions:

- Contrary to the conjectures of other authors [1], the selection of memory layout has a significant impact on address-computation overhead. Less than 0.1% of all memory layouts for the examined access sequences result in minimum overhead. Using optimal address-code generation alone (using the MCC technique) is not sufficient to minimize overhead.
- The 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.

4.1 Experimental Methodology

Figure 4 outlines the experimental methodology. For each access sequence, heuristic solutions to the offset assignment problem are found by using all combinations of three ARA and five SOA algorithms. Each combination produces a set of memory sub-layouts (see Figure 1). If m sub-layouts are produced, then there are $p = \frac{(m!)(2^m)}{2}$ possible memory layouts. The address-computation overhead of each memory layout is computed using the MCC method. 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.

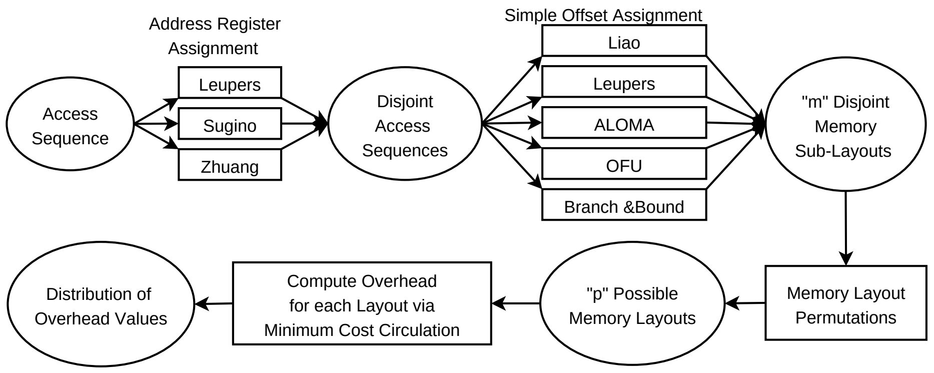

Access Sequence
Address Register Assignment
Leupers
Sugino
Zhuang
Disjoint Access Sequences
Simple Offset Assignment
Liao
Leupers
ALOMA
OFU
Branch &Bound
"m" Disjoint Memory Sub-Layouts
Memory Layout Permutations
"p" Possible Memory Layouts
Compute Overhead for each Layout via Minimum Cost Circulation
Distribution of Overhead Values

Fig. 4: Procedure for evaluating offset assignment algorithms. There are 15 paths in the chart, for the 15 combinations of ARA and SOA algorithms.

4.2 Test Environment

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

Access sequences are obtained from kernels in the UTDSP benchmark suite. Each kernel is compiled using `-O2` optimization by `gcc` version 3.3.2. Unfortunately, the `gcc` compiler does not generate code for the C54X family of DSPs. Instead, we modified `gcc` to output access sequences from inner-most loops prior to register allocation. Then the overhead of access sequences and memory layouts are evaluated statically, using the MCC technique described in Section 2.2.

Given an access sequence with n variables, we compute the optimal memory layout by evaluating the MCC of all possible $\frac{n!}{2}$ layouts (see Section 3.3). Due to the exponential growth of the solution space, experiments are restricted to sequences with up to 12 variables. The five access sequences used in this study are from five kernels in the UTDSP benchmark suite which produced access sequences with $n \leq 12$.

4.3 The Efficiency of Offset Assignment Heuristics

Table 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 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 Section 3.2 and 3.1 indicate 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 average overhead of all layouts in the solution space. The importance of selecting a suitable way to combine sub-layouts cannot be overstated. For instance, a surprising finding is that the heuristically generated sub-layouts for a given instance of the problem can be combined in one way to generate the best possible overhead for that instance, and the same sub-layouts can be combined in another way to generate the worst possible overhead.

4.4 The Efficiency of ARA Heuristics

Each of the three ARA algorithms — Leupers, Sugino, and Zhuang — can be combined with five SOA algorithms (Figure 4) to produces a memory layout.

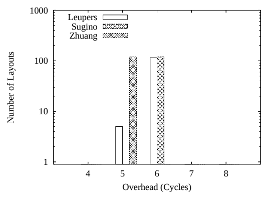
Access Sequence	overhead (cycles)	Exhaustive Number of Layouts	Exhaustive % of Layouts	Algorithmic Number of Layouts	Algorithmic % of Layouts
iir_arr	4	5	0.02%	0	0.00%
	5	281	1.39%	125	34.72%
	6	5707	28.31%	235	65.28%
	7	10526	52.21%	0	0.00%
	8	3641	18.06%	0	0.00%
Average overhead		6.87		5.65	
iir_arr_swp	6	144	0.00%	0	0.00%
	7	19557	0.01%	72	0.33%
	8	1514917	0.63%	2240	10.23%
	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	
latnrm_arr_swp	6	323	0.02%	117	0.60%
	7	10785	0.59%	303	1.55%
	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	
latnrm_ptr	6	1449	0.08%	28	0.21%
	7	29682	1.64%	481	3.68%
	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	
latnrm_ptr_swp	6	323	0.02%	5	0.04%
	7	7706	0.42%	138	1.04%
	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 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.

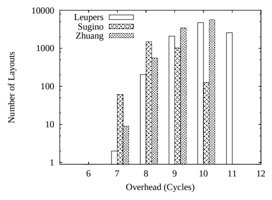
All of the layouts produced by an ARA algorithm are combined into a set. The distribution of overhead values for the possible layouts produced by each ARA algorithm are shown in Figure 5. For instance, Figure 5(a) shows that Leupers' ARA algorithm can admit over 100 layouts with 6 cycles of overhead 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 3.3). Figure 5 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(b), Leupers and Marwedel's ARA algorithm yields a total of 9600 possible layouts, two 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.

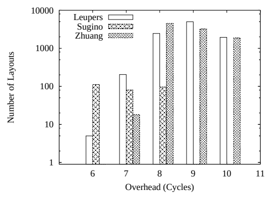
The results also suggest that locally optimal sub-layouts do not lead to globally optimal memory layouts. An ARA algorithm using more ARs assigns fewer variables to each register. In the case of Leupers and Marwedel's algorithm, and occasionally Zhuang's algorithm, as few as two variables may be assigned 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.



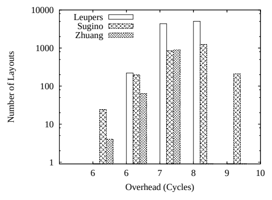
(a) iir_arr



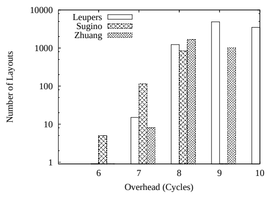
(b) iir_arr_swp



(c) latnrm_arr_swp



(d) latnrm_ptr



(e) latnrm_ptr_swp

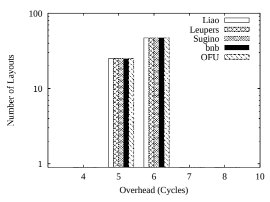
Fig. 5: Distribution of overhead values produced by each ARA algorithm on different test cases. The number of layouts shown for each algorithm is the union of 5 sets of layouts, each produced with one of the 5 different SOA algorithms, but using the same ARA algorithm.

4.5 The Efficiency of SOA Heuristics

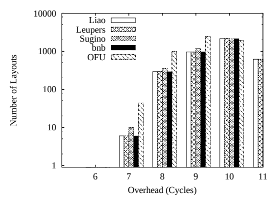
The distributions in Figure 6 are complementary to those in Figure 5, but focused on the layouts produced by each of the five SOA algorithms. For instance, Figure 6(b) shows that the SOA algorithm designed by Sugino *et al.* can admit over 1000 layouts with 9 cycles of overhead. Each of these layouts 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 6. 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. Because the SOA sub-problems are small the algorithms 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.

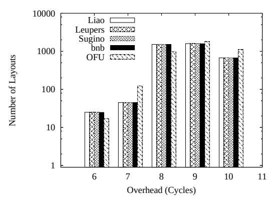
Figure 6 also further supports previous suggestions that combining optimal sub-layouts does not result in optimal layouts. For instance, in Figure 6(e), the OFU algorithm generates sub-layouts that combined to form optimal memory layouts, while the Branch-and-Bound algorithm, which finds optimal sub-layouts, does not admit any optimal memory layouts.



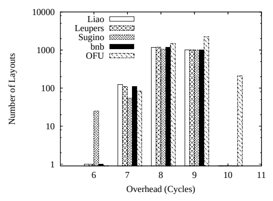
(a) iir_arr



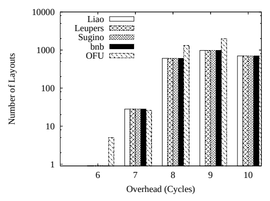
(b) iir_arr_swp



(c) latnrm_arr_swp



(d) latnrm_ptr



(e) latnrm_ptr_swp

Fig. 6: 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.

5 Related Work

In 2003, Leupers presented a comprehensive experimental evaluation of algorithms for the SOA [6]. This is the first comparative evaluation of algorithms for the GOA problem. It has three distinguishing features:

- The GOA problem is evaluated as three problems: address register assignment, simple offset assignment, and memory-layout permutation.
- 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 technique.

Some algorithms for generating a memory layout were not included in our study. Atri *et al.* and Wess and Zeitlhofer, propose algorithms that attempt to iteratively improve a memory layout [8, 9]. Leupers and David, and Wess and Gotschlich, propose simulation-based algorithms to generate memory layouts. These algorithms were omitted because they are computationally expensive. The algorithms must compute the overhead of many memory layouts before a final memory layout is produced.

This study only focused on evaluating offset assignment for *scalar* variables in straight line code. Reducing address-computation overhead in loops is more difficult because the problem of finding optimal addressing code for a loop is NP-complete [1]. Many researchers have proposed alternative methods to reduce overhead for array accesses in loops. Leupers and David, Cheng and Lin, and Chen and Kandemir, all propose algorithms to reduce address-computation overhead in loops through improved address register allocation and data and instruction re-ordering [10–12].

Although our study investigates algorithms that directly generate a memory layout, overhead can also be reduced by manipulating the access sequence. Rao and Pande, Lim *et al.*, and Kandemir *et al.* each propose an algorithm that reorders the access sequence so that an offset assignment algorithm can produce a lower-overhead layout [13–15]. Choi and Kim propose a unified algorithm to simultaneously find an instruction schedule and a low-overhead offset assignment [16]. The access sequence can also be manipulated by reducing the number of unique variables accessed. Ottoni *et al.*, and Zhuang *et al.*, propose algorithms to coalesce variables [17, 5]. Although some of the scheduling and coalescing algorithms simultaneously find memory layouts, it is still possible to perform an additional offset assignment pass to further reduce overhead.

6 Conclusion

The minimum cost circulation technique produces the optimal addressing code for a *fixed* memory layout and access sequence by allowing variables to be accessed by multiple address registers. This paper shows that the memory layout has a significant impact on the address-computation overhead, even when using optimal address-code generation. Furthermore, current offset assignment algorithms produce sub-layouts that can span the full range of values in the solution space. In order for current algorithms to generate only low-overhead layouts, a new combinatorial problem, the memory-layout permutation problem, must be solved.

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. We observe 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 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. Thus, for GOA problems with 12 or fewer variables, 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 paper 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. We call this new problem the memory-layout permutation (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 we can find an algorithm to address the MLP problem, the algorithm can be used to solve GOA.

Our study suggests two new directions for improving GOA solutions. One direction is to propose a solution to the MLP problem. An alternative direction is to replace the individual solutions of the ARA, SOA, and MLP problems with a combined method that generates memory layouts that minimizes overhead, as computed by the minimum-cost circulation technique.

References

1. Gebotys, C.: DSP address optimization using a minimum cost circulation technique. In: ICCAD '97: Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design, Washington, DC, USA, IEEE Computer Society (1997) 100–103

2. Leupers, R., Marwedel, P.: Algorithms for address assignment in DSP code generation. In: Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design. (1996) 109–112
3. Liao, S., Devadas, S., Keutzer, K., Tjiang, S., Wang, A.: Storage assignment to decrease code size. ACM Transactions on Programming Languages and Systems **18**(3) (1996) 235–253
4. Sugino, N., Iimuro, S., Nishihara, A., Jujii, N.: DSP code optimization utilizing memory addressing operation. IEICE Trans Fundamentals (8) (1996) 1217–1223
5. Zhuang, X., Lau, C., Pande, S.: Storage assignment optimizations through variable coalescence for embedded processors. In: LCTES '03: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tools for Embedded Systems, New York, NY, USA, ACM Press (2003) 220–231
6. Leupers, R.: Offset assignment showdown: Evaluation of dsp address code optimization algorithms. In: CC '03: Proceedings of the 12th International Conference on Compiler Construction. (2003) 290–302
7. Bartley, D.H.: Optimizing stack frame accesses for processors with restricted addressing modes. Software – Practice & Experience **22**(2) (1992) 101–110
8. Atri, S., Ramanujam, J., Kandemir, M.: Improving offset assignment for embedded processors. Lecture Notes in Computer Science **2017** (2001) 158–172
9. Wess, B., Zeitlhofer, T.: On the phase coupling problem between data memory layout generation and address pointer assignment. In: SCOPES. (2004) 152–166
10. Leupers, R., Basu, A., Marwedel, P.: Optimized array index computation in DSP programs. In: Asia and South Pacific Design Automation Conference. (1998) 87–92
11. Cheng, W.K., Lin, Y.L.: Addressing optimization for loop execution targeting dsp with auto-increment/decrement architecture. In: ISSS '98: Proceedings of the 11th International Symposium on System Synthesis, Washington, DC, USA, IEEE Computer Society (1998) 15–20
12. Chen, G., Kandemir, M.: Optimizing address code generation for array-intensive dsp applications. In: CGO '05: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, IEEE Computer Society (2005) 141–152
13. Rao, A., Pande, S.: 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, New York, NY, USA, ACM Press (1999) 128–138
14. Lim, S., Kim, J., Choi, K.: Scheduling-based code size reduction in processors with indirect addressing mode. In: CODES '01: Proceedings of the 9th International Symposium on Hardware/Software Codesign, New York, NY, USA, ACM Press (2001) 165–169
15. Kandemir, M.T., Irwin, M.J., Chen, G., Ramanujam, J.: Address register assignment for reducing code size. In: CC '03: Proceedings of the 12th International Conference on Compiler Construction. (2003) 273–289
16. Choi, Y., Kim, T.: Address assignment combined with scheduling in DSP code generation. In: DAC '02: Proceedings of the 39th Conference on Design Automation, New York, NY, USA, ACM Press (2002) 225–230
17. Ottoni, D., Ottoni, G., Araujo, G., Leupers, R.: Improving offset assignment through simultaneous variable coalescing. In: 7th International Workshop on Software and Compilers for Embedded Systems. (2003) 285–297