HAL
open science

# Evaluating address register assignment and offset assignment algorithms

Johnny Huynh, Nelson José Amaral, Paul Berube, Sid Touati

## To cite this version:

Johnny Huynh, Nelson José Amaral, Paul Berube, Sid Touati. Evaluating address register assignment and offset assignment algorithms. ACM Transactions on Embedded Computing Systems (TECS), 2011, 10 (3), pp.37:1-37:22. 10.1145/1952522.1952530 . inria-00635915

**HAL Id: inria-00635915**

**https://inria.hal.science/inria-00635915v1**

Submitted on 26 Oct 2011

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Evaluating Address Register Assignment and Offset Assignment Algorithms

JOHNNY HUYNH, JOSÉ NELSON AMARAL, and PAUL BERUBE
University of Alberta
and
SID-AHMED-ALI TOUATI
Université de Versailles

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. This paper presents an evaluation of memory layouts that uses Gebotys' optimal address-code generation technique. The use of this evaluation method leads to new optimization problem: the Memory Layout Permutation (MLP) problem. We then use 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. The 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.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Address registers, DSPs, offset assignment, register allocation

## 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 to variables stored in memory through indirect addressing modes. These addressing modes often support post-incrementing and post-decrementing of the address stored in the AR. 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 called a memory layout. When two subsequent memory accesses indexed by the same AR are not adjacent in the memory layout, an extra address-computation instruction is required. 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.

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). Authors' addresses: J. Huynh, J. Amaral, and P. Berube, University of Alberta, Department of Computing Science, Edmonton, Alberta, Canada; email: {huynh,amaral,berube}@cs.ualberta.ca; S. Touati, University of Versailles Saint-Quentin-en-Yvelines, 45 avenue des Etats Unis, Versailles, France; email: touati@prism.uvsq.fr
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 1529-3785/20YY/0700-0001 $5.00

The total overhead of address-computation in DSP programs is influenced by three factors. First, the memory layout impacts the efficient use of post-increment and post-decrement addressing modes. Second, the instruction sequence determines the order of memory accesses and thus the traversal of the memory layout. Third, when more than one AR is available, each AR produces a concurrent traversal of the memory layout to access some portion of the required memory. Thus, the assignment of memory accesses to ARs determines the memory access sequence for each AR. In this work, we consider the case of a fixed instruction sequence and investigate the impact of memory layout and memory access assignment to ARs on address-computation overhead.

Previous work has investigated these three components of address-computation overhead individually. In all but one case, previous problem formulations have resulted in optimal solutions with high algorithmic complexity, necessitating heuristic solutions. The only exception is Gebotys' network-flow solution finds the optimal usage of ARs to access the data in polynomial time, given a fixed memory layout and instruction sequence [Gebotys 1997].

However, even with an optimal usage of ARs, explicit addressing instructions can make up for a third of the instructions in a block of code. Thus, there is still significant room for reducing code size, and consequently, execution time, by reducing the number of addressing instructions used by a DSP. In this paper, we demonstrate that the choice of memory layout can have a significant impact on the address-computation cost, even when ARs are used optimally. Consequently, the cost of the final addressing code depends on the quality of the algorithm used to generate the memory layout.

Several heuristic GOA algorithms generate a memory layout that attepmts to minimizes address computation overhead on the assumption that every access to a variable uses the same AR [Leupers and Marwedel 1996; Liao et al. 1996; Sugino et al. 1996; Zhuang et al. 2003]. Using this approach, variables accessed in an instruction sequence are partitioned, and each partition is assigned to an AR. This partitioning problem is the Address Register Assignment (ARA) problem. However, assigning all accesses to the same variable to the same AR is a source of sub-optimal address-computation overhead, and consequently even optimal ARA solutions may not minimize address computation overhead.

When variables have been partitioned and assigned to a single AR, finding an optimal memory layout for a partition is the Simple Offset Assignment (SOA) problem [Liao et al. 1996], which must be approximated heuristically. Given the SOA solutions for each partition, the network-flow technique can find the optimal AR usage for the concatenation of the SOA memory sub-layouts. Unfortunatly, the overhead of solutions found in this way is sensitive to the sub-layout concatenation order. This ordering problem is the Memory Layout Permutation (MLP) problem. The experiments reported in this paper show that different orderings of sub-layouts have a significant impact on the optimal address-computation overhead. Specifically, the minimum overhead of the worst layouts is twice that of the optimal layouts.

While there has been significant investigation of the ARA and SOA problems, and while all the proposed algorithms find memory layouts using similar techniques, there are very few comparisons of these algorithms [Leupers and Marwedel 1996; Zhuang et al. 2003]. Furthermore, a rigourous evaluation of the quality of these heuristic solutions is lacking, since address-computation overheads have only been measured using the cost models of the heuristic algorithms, and not by an optimal technique such as Gebotys' network-flow formulation. In contrast to our experimental results regarding the MLP problem, we find that 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;

—a precise formulation for the address register allocation problem that removes an important restriction that is present in most of the previous studies of offset assignments;

—the formulation of a new optimization problem, the memory-layout permutation problem (MLP). MLP must be solved in order to use a minimum-cost circulation (MCC) technique to evaluate the minimum address-computation overhead incurred in memory layouts generated 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 including a new formal definition of the problem and a motivating example. In spite of the extensive literature on GOA, to the best of our knowledge this is the first formalization of the GOA problem. 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

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. In the formulation of the address-register allocation and offset-assignment problems, these costs are parametrized by INIT$\in \mathbb{N}$ and JUMP$\in \mathbb{N}$, that corresponds to the number of processor cycles required to execute the address computation instructions $I_{INIT}$ and $I_{JUMP}$.

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

While the processor model is based on a specific family of processors, the problem formulation and the evaluation methodology presented in this paper are general and can be applied to any processor with post-increment/post-decrement addressing modes.

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 known 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.* [Liao et al. 1996] 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.* [Liao et al. 1996] 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 5.1).

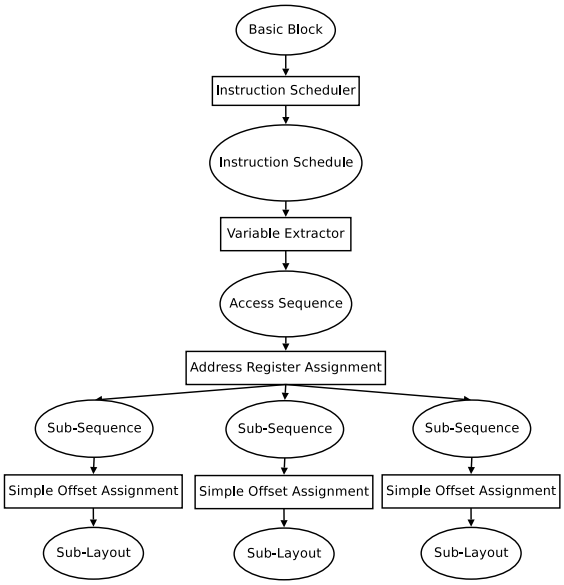


Fig. 1: The traditional approach to generate a memory layout for the access sequence of a basic block. The sub-sequences generated by address register assignment access disjoint sets of variables. The resulting set of sub-layouts can then be placed independently in memory to form the final memory layout.

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 can be accessed by multiple address registers at different points in the program. However, the union of all sub-layouts must still form a contiguous layout. For example, if the first access to variable $B$ is assigned to AR1, and the second access to $B$ to AR2, AR1 and AR2 must agree where $B$ is located in memory, and this location cannot be computed for each AR individually. Liao *et al.* [Liao et al. 1996] 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 5.2).

Figure 1 illustrates the traditional approach to produce a memory layout from a basic block's access sequence. 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 SOA problem. 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.

This section presents the mathematical formalism for a precise definition of the problems and concepts studied in this article in order to avoid . The intent of this formalism is to avoid confusion or miss-understanding, which are common when technical heuristics are described in textual forms.

## 2.2 Formal Definitions

2.2.1 *A More Precise Definition of the GOA problem.* The original SOA paper by Liao *et al.* presents a precise and formal definition of the SOA problem. However, that definition does not lead to a suitable definition of the GOA problem. The precise definition of GOA presented in this session leads to an SOA definition that is slightly distinct from the definition given by Liao *et al.*.

Let us assume a DSP processor with a set of address registers $\mathcal{R} = \{A_1, \cdots, A_k\}$. Let $V$ be the set of $n$ variables accessed in the program. Let $S$ be the sequence of variables accesses, which consists of a finite sequence of $l$ variable accesses $S = (s_1, \cdots, s_l)$. The variables belonging to $V$ should be placed in memory according to a memory layout. Mathematically, a memory layout is simply a topological sort of the

variables.[1]

*Definition* 2.1 *Memory Layout.* Let $V$ be a set of variables. The function $M$ is a *memory layout* of $V$ if it defines a topological sort of $V$:

$$\begin{aligned} M:\ & V \rightarrow [1,n] \\ & v \mapsto M(v) \end{aligned}$$

where $\forall v_1, v_2 \in V, v_1 \neq v_2 \Longrightarrow M(v_1) < M(v_2) \vee M(v_2) < M(v_1)$

In other words, $M$ is an integral injective function of $V$:

$$\forall v_1, v_2 \in V, v_1 \neq v_2 \Longrightarrow M(v_1) \neq M(v_2)$$

Once a memory layout is fixed, the final code for the sequence $S$ can be generated. This code includes, in addition to the variable accesses, some address computation instructions $I_{INIT}$ and $I_{JUMP}$. The code generated for $S$, given a memory layout $M$ and $k$ address registers, can be noted as $\widehat{S}^{M,k}$. The cost of a code $\widehat{S}^{M,k} = (\widehat{s}_1, \cdots, \widehat{s}_{l'})$ in terms of address computation overhead is equal to $c(\widehat{S}^{M,k}) = \sum_{i=1,l'} cost(\widehat{s}_i)$ where:

$$cost(\widehat{s}_i) = \begin{cases} INIT & \text{if the instruction } \widehat{s}_i \text{ is an initial address initialization;} \\ JUMP & \text{if the instruction } \widehat{s}_i \text{ is an explicit address jump;} \\ 0 & \text{otherwise.} \end{cases}$$

The formal definition of the GOA problem is presented below.

*Problem* 2.2 *GOA.* Let $V$ be a set of variables and $S$ be an access sequence. Assume a machine with $k$ address registers. Compute $M$, a memory layout for $V$, such that $c(\widehat{S}^{M,k})$ is minimal.

Until now, the literature does not present an algorithm that precisely solves the GOA problem. Instead, many sub-heuristics have been provided to solve some sub-problems, such as SOA (GOA with $k = 1$) and ARA, and some restricted GOA variants as we will see later.

As proved by Liao *et al.*, SOA is NP-complete, and consequently GOA is also NP-complete. However, some adjacent sub-problems have been proved to be polynomial ones, such as the following problem.

*Problem* 2.3 *Opti-Code-Gen.* Let $V$ be a set of variables and $S$ be an access sequence. Assume a machine with $k$ address registers. Let $M$ be a fixed memory layout for $V$. Compute a generated code $\widehat{S}^{M,k}$ such that $c(\widehat{S}^{M,k})$ is minimal.

Getobys solved the Opti-Code-Gen problem in polynomial time using the MCC minimum cost circulation (MCC) technique [Gebotys 1997].

Solving the GOA problem, and hence the Opti-Code-Gen problem, leads to a choice of an address register allocation for each variable access in $S$. That is, the following function should be computed.

*Definition* 2.4 *Address Register Allocation.* Let $V$ be a set of variables and let $\mathcal{R} = \{A_1, \cdots, A_k\}$ be a set of address registers. Let $S$ be a sequence of $l$ variables accesses $S = (s_1, \cdots, s_l)$. An address register allocation is a function, noted *alloc*, that assigns to each variable access the address register used to hold the computed memory address.

$$\begin{aligned} alloc:\ & S \rightarrow \mathcal{R} \\ & s_i \mapsto alloc(s_i) \end{aligned}$$

In this formal definition distinct accesses to the same variable inside the sequence $S$ can use distinct address registers. However, Liao *et al.* adopted a restricted address register allocation policy, which does not yield a precise GOA definition. In their formulation address registers are allocated to variables and not to their distinct accesses. This means that all accesses to the same variables are restricted to be done using the same address register. To contrast with our definition above, we re-state precisely the formulation of the address register allocation problem that appears in Liao *et al.*.

[1] In the experimental study presented is this paper the sequence of variable accesses $S$ is extracted from a basic block, however any topological sort of variable accesses can be used as an input to the problem. Thus the solution can be extended beyond the scope of a basic block.

*Definition* 2.5 *Restricted Address Register Allocation.* Let $V$ be a set of variables and let $\mathcal{R} = \{A_1, \cdots, A_k\}$ be a set of address registers. Let $S$ be a sequence of $l$ variable accesses $S = (s_1, \cdots, s_l)$. A *restricted address register allocation* is a function, noted $\overline{alloc}$, that assigns to each variable $v$ an address register used to hold the computed memory address of $v \in V$ in all accesses to this variable inside $S$.

$$\begin{aligned} \overline{alloc} : \; & V \rightarrow \mathcal{R} \\ & v \mapsto \overline{alloc}(s_i) \end{aligned}$$

The above definition means that

$$\overline{alloc}(v) = A_i \in \mathcal{R} \Longrightarrow \forall s_j \in S \text{ an access to } v, alloc(s_j) = A_i$$

This is precisely the limitation in Liao's model. Unfortunately most papers that appeared after Liao's use this definition of ARA. In order to not confuse the reader who is familiar with the usual ARA problem, we will still use the term ARA in the remaining of this article instead of *restricted* ARA.

2.2.2 *Sub-layouts.* The definition of GOA given by Liao *et al.* is based on the concept of restricted ARA. Therefore, many articles decompose the GOA problem into steps that solve the ARA and SOA sub-problems separately. Such decompositions produce multiple memory sub-layouts instead of a unified one. Following are formal definitions of these sub-layouts and their composition.

*Definition* 2.6 *Memory Sub-layout.* Let $V$ be a set of variables. $\ddot{M}$ is a sub-layout of $V$ associated to $V' \subseteq V$ if and only if it defines a topological sort for $V' \subseteq V$.

$$\begin{aligned} \ddot{M} : \; & V' \rightarrow [1, n] \\ & v \mapsto \ddot{M}(v) \end{aligned}$$

where $\forall v_1, v_2 \in V', v_1 \neq v_2 \Longrightarrow \ddot{M}(v_1) < \ddot{M}(v_2) \vee \ddot{M}(v_2) < \ddot{M}(v_1)$

*Definition* 2.7 *Composed Memory Layout.* Let $V$ be a set of variables. Let $\ddot{M}_1^{V_1}, \cdots, \ddot{M}_p^{V_p}$ be a set of memory sub-layouts for $V$ associated to the sub-sets of variables $V_1 \subseteq V, \cdots, V_p \subseteq V$ resp. Let $M$ be the function defined as follows:

$$\begin{aligned} M : \; & V_1 \cup \cdots \cup V_p \rightarrow [1, n] \\ & v \mapsto M(v) = \ddot{M}_i(v) \text{ if } v \in V_i, \forall i \in [1, p] \end{aligned}$$

We call $M$ a composed memory layout, noted $M = \ddot{M}_1 \circ \cdots \circ \ddot{M}_p$, if and only if

(1) The sets $V_i$ define a partition of the set $V$:

$$\begin{cases} V_1 \cup \cdots \cup V_p = V \\ \forall i, j, V_i \cap V_j = \phi \end{cases}$$

(2) $M$ defines a topological sort of $V$:

$$\forall v_1, v_2 \in V, v_1 \neq v_2 \Longrightarrow M(v_1) \neq M(v_2)$$

## 3. MOTIVATING EXAMPLE

Using the processor model described in Section 2, consider the following 6-variable access sequence: '`a d b e c f b e c f a d`'. According to the commonly used restricted address register allocation problem formulation, the variables are partionned into disjoint sets. Each set is accessed exclusively by a single AR. For example, variables {`a,b,c`} can be assigned to AR $A_1$, and variables {`d,e,f`} to AR $A_2$. According to the formal definitions of Section 2.1, this means that if access $s_i$ =`a`, then $alloc(s_i) = A_1$, and so on. The variables assigned to each AR are then independently arranged in memory to form two independent *sub-layouts*, as shown in Figure 2. ARs $A_1$ and $A_2$ independently access the variables, as shown in Figures 3(a) and 3(b). In this example, the AR assigned to each layout must perform one initialization and one jump. Thus, the address-computation overhead for each AR is 3 cycles, for a total of 6 cycles of overhead.

In the traditional approach to the GOA problem, the sub-layouts in Figure 2 are considered "optimal" for two reasons. First, no ordering exists for either variable subset, {`a,b,c`} or {`d,e,f`}, with less than 3 cycles of overhead. Second, no partitioning of the 6 variables exists that can produce sub-layouts with a total

| | | | | |
|---|---|---|---|---|
| | a | b | c | |

(a) Sub-layout assigned to $A_1$

| | | | | |
|---|---|---|---|---|
| | d | e | f | |

(b) Sub-layout assigned to $A_2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | a | b | c | d | e | f | |

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | d | e | f | b | c | a | |

(d) Layout using alternative sub-layouts

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | b | c | a | d | e | f | |

(e) Swapping the sub-layouts

Fig. 2: Several memory layouts.

| access | addressing code | over-head |
|---|---|---|
| | $A_1$ = &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) Addressing code to access the sub-layout in Figure 2(a)

| access | addressing code | over-head |
|---|---|---|
| | $A_2$ = &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) Addressing code to access the sub-layout in Figure 2(b)

Fig. 3: The layouts in Figures 2(a) and 2(b) are accessed by address registers $A_1$ and $A_2$, respectively. Overhead is incurred when an AR is initialized, or when an AR accesses two non-adjacent variables consecutively.

| access | addressing code | over-head |
|---|---|---|
| | $A_1$ = &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 | $A_1$ | |

(a) Accessing layout 2(c)

| access | addressing code | over-head |
|---|---|---|
| | $A_1$ = &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 layout 2(d)

| access | addressing code | over-head |
|---|---|---|
| | $A_1$ = &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 | $A_1$ | |

(c) Accessing layout 2(e)

Fig. 4: Addressing code to access the layouts in Figures 2(c), 2(d), and 2(e). Allowing variables `d` and `a` to be accessed by both address registers (a) may reduce the overhead to 5 cycles; (b) may keep it at 6 cycles; or (c) may result in the minimum overhead value – 4 cycles.

overhead that is less than 6 cycles. However, these sub-layouts do not minimize the address-computation overhead of the input access sequence. The problem is the restriction that each set of variables be accessed by a single AR.

The memory layouts in Figure 2(a) and 2(b) can be placed contiguously in memory to form the single memory layout shown in Figure 2(c). Figure 4(a) shows that when the general formulation to the address registration allocation problem (see definition 2.4) is adopted, $A_1$ can be used for the last access of variable `d` (originally assigned to $A_2$) without requiring additional overhead. Similarly, $A_2$ is used for the last access of variable `a` (originally assigned to $A_1$). This solution has an address-computation overhead of 5 cycles instead of 6.

Now, consider an alternative sub-layout to 2(a), with the variables ordered as {`b,c,a`}. Variables {`d,e,f`} are kept in the same order as shown in Figure 2(d). Similar to Figure 3, if variables {`d,e,f`} are assigned

to $A_1$ and variables {`b,c,a`} are assigned to $A_2$, the total overhead is 6 cycles. If the two sub-layouts are placed contiguously in memory, as shown in Figure 4(b), the address-computation overhead is still 6 cycles.

However, if variables {`b,c,a`} are placed before variables {`d,e,f`}, producing the memory layout shown in Figure 2(e), and each variable can be accessed by more than one AR, then the address computation overhead is reduced to 4 cycles, as shown in Figure 4(c). Despite the similarities with memory layouts 2(c) and 2(d), the layout in Figure 2(e) is the only one that allows for the minimum amount of address-computation overhead.

The objective of offset assignment is to minimize address-computation overhead. The above example highlights the fact that the traditional approach to solving GOA by using ARA and SOA does not minimize overhead because of the restriction that each variable must be accessed by a single AR. However, the network-flow technique can reduce the overhead of a memory layout by allowing multiple ARs to access a variable. Considering the network-flow technique for GOA raises the following questions:

—Do different ARA and SOA heuristics affect the combined-layout overhead?

—How to arrange sub-layouts to minimize address-computation costs?

—How to determine whether a given memory layout is optimal?

## 4. COMPUTING ADDRESS-COMPUTATION OVERHEAD

The example in Section 3 indicates that variables must be accessed by multiple ARs to minimize addressing-code overhead. Thus, 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 minimum overhead. In order to evaluate the overhead of memory layouts, optimal addressing code is required.

Gebotys proposes an algorithm to find optimal addressing code [Gebotys 1997]. 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 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. In this paper, MCC is used to evaluate the quality of all memory layouts.

## 5. OFFSET ASSIGNMENT ALGORITHMS

Existing heuristic-based algorithms solve GOA as illustrated in Section 3. Given an access sequence $S$, and $k$ ARs, a memory layout is found through the execution of the following steps:

(1) ARA assigns each variable $v \in V$ to a single AR $A_i, 1 \leq i \leq k$.[2]

(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 $m_1 \ldots m_k$ into a contiguous memory layout. MLP is absent from the literature because previous solutions to GOA assigned variables, rather than accesses, to ARs. MLP 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.

### 5.1 Simple Offset Assignment

Bartley introduced the SOA problem in 1992 and solved it as a *maximum-weight Hamiltonian-path* problem [Bartley 1992]. Given an access sequence $S$, construct a weighted access graph $G$. 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 [Liao et al. 1996], which is NP-complete. Thus, all subsequently proposed algorithms approximate a solution to the SOA problem by finding a path cover on $G$.

We examine five heuristic solutions to the SOA problem:

[2]Herein lies a fundamental difference between existing algorithms and the formulation presented in Definition 2.4. While existing algorithms assign each variable $v \in V$ to a single AR, the address register allocation formulation in Section 2.1 assigns each variable access $s \in S$ to an AR.

(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 $G$ [Liao et al. 1996].

(2) Leupers extends the algorithm by Liao *et al.* by proposing a tie-break function to decide between edges of equal weights [Leupers and Marwedel 1996] [Leupers 2003].

(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 [Sugino et al. 1996].

(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) [Liao et al. 1996; Leupers 2003].

(5) Liao also presents a branch-and-bound algorithm that finds the maximum-weight path cover [Liao 1996]. The algorithm has exponential time-complexity, but for small graphs, our implementation runs in a reasonable amount of time.

### 5.2 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. Additionally, as shown in Section 3, assigning variables to ARs may not minimize address-computation overhead.

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 5.1 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 [Leupers and Marwedel 1996].

(2) Sugino *et al.* use a heuristic-based algorithm that iteratively partitions the variables and selects the partitioning with the lowest estimated overhead [Sugino et al. 1996].

(3) Zhuang *et al.*'s variable coalescing algorithm for offset-assignment problems assign variables to ARs [Zhuang et al. 2003].

### 5.3 Memory Layout Permutations

Figure 5 illustrates the process of generating a memory layout. ARA produces a set of disjoint access sub-sequences that are solved as independent SOA problems. Solving each SOA instance produces a memory layout called an ARA sub-layout. 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, Section 3 demonstrates that if a variable can be accessed by multiple ARs, address-computation overhead may be reduced by placing the sub-layouts contiguously in memory. Since the MCC technique allows variables to be accessed by multiple ARs, the sub-layouts can no longer be placed independently in memory. Let $\ddot{M}_i$ be a sub-layout (for instance resulted from a sub-solution of GOA). So we can define a reverse memory layout as follows.

*Definition* 5.1 *Reverse Memory Layout.* Let $M$ be a memory layout for a set of variables $V$. $\widetilde{M}$ is a reverse memory layout of $M$ if it defines a topological sort of $V$ in reverse order compared to $M$. For

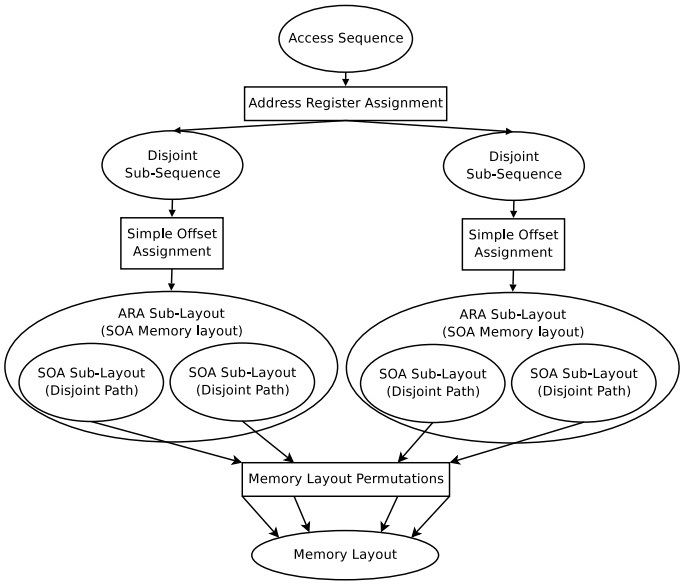


Fig. 5: Performing address register assignment followed by simple offset assignment generates memory sub-layouts that must be placed in memory. The problem of finding a placement that minimizes overhead is called the memory-layout permutation problem.

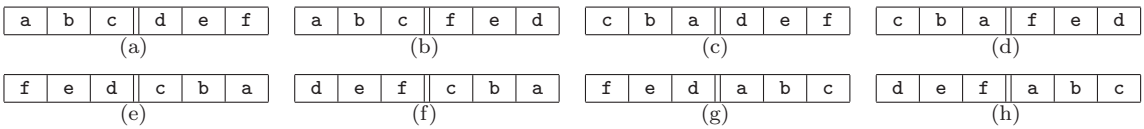


Fig. 6: Permutations of two sub-layouts

instance, the following function can be used to define the reverse memory layout of $M$.

$$\begin{aligned}\widetilde{M}:\ & V \rightarrow [1,n]\\ & v \mapsto \widetilde{M}(v) = n - M(v) + 1\end{aligned}$$

$\ddot{M}_i$ and $\widetilde{\ddot{M}_i}$ produce the same local cost. Thus, a precise definition for the memory sub-layout composition problem must consider both $\ddot{M}_i$ and $\widetilde{\ddot{M}_i}$. Let $(\ddot{M}_1|\widetilde{\ddot{M}_1})$ stand for an instance of either $\ddot{M}_i$ or its reverse $\widetilde{\ddot{M}_i}$. Now the *memory-layout permutation* (MLP) problem can be defined as follows:

*Problem* 5.2 *MLP*. Let $V$ be a set of variables, $S$ an access sequence and $k$ be the number of address registers. Let there be $p$ disjoint memory sub-layouts $\ddot{M}_1, \cdots, \ddot{M}_p$. Compute $M$ a composed memory layout such that

$$M = (\ddot{M}_1|\widetilde{\ddot{M}_1}) \circ \cdots \circ (\ddot{M}_p|\widetilde{\ddot{M}_p})$$

such that $c(\widehat{S}^{M,k})$ is minimal.

The MLP solution space is extremely large: $p$ sub-layouts can form $p!$ permutations. For each permutation, each sub-layout can be placed in memory as either $\ddot{M}_i$ or $\widetilde{\ddot{M}_i}$. Thus, $p$ sub-layouts originate $(p!)(2^p)$ layouts. However, an ordering of layouts $\ddot{M}_1, \ldots, \ddot{M}_p$ is equivalent to its reciprocal layout, $\widetilde{\ddot{M}_p}, \ldots, \widetilde{\ddot{M}_1}$, since all variables have the same relative offset to each other. Thus, the MLP solution space is $\frac{(p!)(2^p)}{2}$ memory layouts. Figure 6 shows how 2 sub-layouts can form 8 possible layouts, half of which are reciprocals of another.

When reciprocals are considered, a single 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 (producing $n$ sub-layouts) then the MLP

problem is reduced to the offset assignment problem. In other words, MLP finds an ordering of sub-layouts to minimize overhead while GOA finds an ordering of variables. If the sub-layouts used in MLP are 1-variable long, then the problems are the same. This implies that if an algorithm solves the MLP problem, the same algorithm solves the offset assignment problem.

## 6. EVALUATING OFFSET ASSIGNMENT ALGORITHMS

Thus far, we have shown that existing heuristic offset-assignment algorithms are not optimal. Furthermore, we have shown that the existing approach to solving the GOA problem is incomplete, as the MLP problem has never been addressed. Despite these problems with existing algorithms, it is useful to compare how well these algorithms perform when used in conjunction with optimal address-code generation.

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

—Contrary to the conjectures of other authors [Gebotys 1997], 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.

### 6.1 Experimental Methodology

Figure 7 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 5). Since an algorithm for the MLP problem does not exist, we evaluate all possible concatenations of the sub-layouts. 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. These distributions are compared against each other, and against the entire solution space, to reveal the significance of the MLP problem, and how much it can be affected by ARA and SOA algorithms.

### 6.2 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 [Lee and Stoodley 1992], the MiBench suite [Guthaus et al. 2001], and the DSPStone suite [Zinojnovic et al. 1994]. 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 4.

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 5.3). Due to the exponential growth of the solution space, experiments are restricted to sequences with up to 12 variables. For $n = 12$, this exhaustive search took over 30 hours on a 14-node dual Opteron 248 cluster. The benchmark kernels that produced access sequences with $n \leq 12$ and are shown in Table I.

### 6.3 The Efficiency of Offset Assignment Heuristics

Tables II-IV show 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

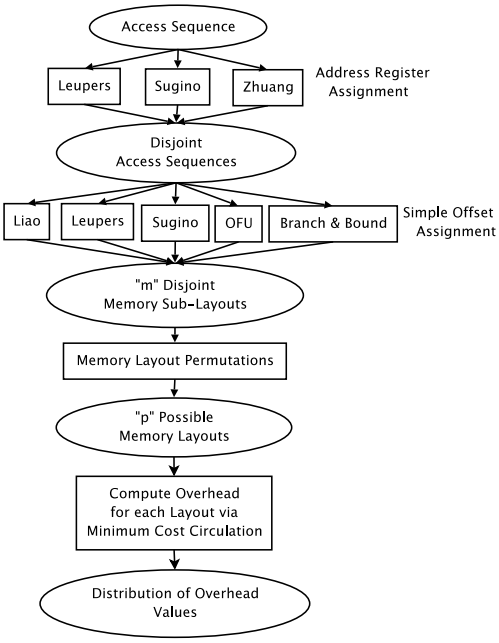


Fig. 7: Procedure for evaluating offset assignment algorithms. There are 15 paths in the chart, for the 15 combinations of ARA and SOA algorithms.

| Kernel | Suite | Length | Variables | Number of Memory Layouts |
|---|---|---|---|---|
| iir_arr | UTDSP | 21 | 8 | 20,160 |
| iir_arr_swp | UTDSP | 33 | 12 | 239,500,800 |
| latnrm_arr_swp | UTDSP | 30 | 10 | 1,824,400 |
| latnrm_ptr | UTDSP | 30 | 10 | 1,824,400 |
| latnrm_ptr_swp | UTDSP | 30 | 10 | 1,824,400 |
| bf_cbc.c_1 | MiBench Security | 20 | 10 | 1,814,400 |
| sha.c_7 | MiBench Security | 18 | 10 | 1,814,400 |
| zdeflate.c_4 | MiBench Security | 25 | 10 | 1,814,400 |
| jccolor.c_0 | MiBench Consumer | 18 | 10 | 1,814,400 |
| jdmerge.c_1 | MiBench Consumer | 13 | 10 | 1,814,400 |
| lame.c_39 | MiBench Consumer | 19 | 10 | 1,814,400 |
| timing.c_4 | MiBench Telecom | 18 | 10 | 1,814,400 |
| convert.c_0 | DSPstone | 18 | 9 | 181,440 |
| speed_control.c_9 | DSPstone | 19 | 8 | 20,160 |
| tone_detector.c_0 | DSPstone | 20 | 8 | 20,160 |

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

space for each GOA problem. The *Algorithmic* columns of Tables II-IV show the *combined* distribution and average address-computation overhead for memory layouts produced by *all 15 combinations* of the ARA and SOA algorithms. These tables show how frequently the concatentation of sub-layouts, produced by some combination of an SOA and ARA algorithm, produces a memory layout with a particular overhead. For instance, in Table III even though the exhaustive search indicates that there is a single layout for the access sequence bf_cbc.c_1 with an overhead of two cycles, this layout is generated fifteen times by the combination of sub-layouts. The motivation for examining the distribution of cycle overhead over all possible layouts for each sequence is to characterize the search space and thus provide solid basis to compare the different

algorithms.

Even though it is known that heuristic-based algorithms are sub-optimal, it is interesting to note that no combination of an SOA and ARA algorithm can produce sub-layouts that generate an optimal memory layout for some sequences. However, it is surprising to observe that the heuristic algorithms can produce sub-layouts that generate memory layouts that vary significantly in overhead, even in the presense of an optimal address-code generation algorithm. The average overhead of all layouts in each GOA problem ranges from 18% to 176% 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 distribution of the overheads obtained using the heuristic-based algorithms presented in Sections 5.2 and 5.1 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.

### 6.4 The Efficiency of ARA Heuristics

Each of the three ARA algorithms — Leupers, Sugino, and Zhuang — can be combined with five SOA algorithms (Figure 7) to produces a memory layout. All of the layouts produced by an ARA algorithm are combined into a set. The distribution of overhead values for the possible layouts produced by each ARA algorithm are shown in Figure 8. For instance, Figure 8(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 5.3). Figure 8 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 8(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.

### 6.5 The Efficiency of SOA Heuristics

The distributions in Figure 9 are complementary to those in Figure 8, but focused on the layouts produced by each of the five SOA algorithms. For instance, Figure 9(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 SOA algorithm with one of the three ARA 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 9. 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,

| Access Sequence | overhead (cycles) | Exhaustive Number of Layouts | Exhaustive % of Layouts | Algorithmic Layout Frequency | 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 II: 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.

Sugino *et al.*'s SOA algorithm admits the most number of optimal layouts.

Figure 9 also further supports previous suggestions that combining optimal sub-layouts does not result in optimal layouts. For instance, in Figure 9(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.

## 7. RELATED WORK

Although Leupers has presented a comprehensive experimental evaluation of algorithms for the SOA problem [Leupers 2003], the experimental study presented in this paper is the first comparative evaluation of algorithms for the GOA problem. It has three distinguishing features:

—GOA is evaluated as three problems: address register assignment, simple offset assignment, and memory-

| Access Sequence | overhead (cycles) | Exhaustive Number of Layouts | Exhaustive % of Layouts | Algorithmic Layout Frequency | Algorithmic % of Layouts |
|---|---|---|---|---|---|
| bf_cbc.c_1 | 2 | 1 | 0.00% | 15 | 0.16% |
| | 3 | 23 | 0.00% | 60 | 0.62% |
| | 4 | 1737 | 0.10% | 610 | 6.35% |
| | 5 | 30441 | 1.68% | 3085 | 32.10% |
| | 6 | 220314 | 12.14% | 5840 | 60.77% |
| | 7 | 668266 | 36.83% | 0 | 0.00% |
| | 8 | 670058 | 36.93% | 0 | 0.00% |
| | 9 | 206280 | 11.37% | 0 | 0.00% |
| | 10 | 17280 | 0.95% | 0 | 0.00% |
| Average overhead | | 7.47 | | 5.53 | |
| sha.c_7 | 4 | 892 | 0.05% | 56 | 5.74% |
| | 5 | 18834 | 1.04% | 300 | 30.74% |
| | 6 | 163925 | 9.03% | 620 | 63.52% |
| | 7 | 622917 | 34.33% | 0 | 0.00% |
| | 8 | 722800 | 39.84% | 0 | 0.00% |
| | 9 | 265592 | 14.64% | 0 | 0.00% |
| | 10 | 19440 | 1.07% | 0 | 0.00% |
| Average overhead | | 7.61 | | 5.58 | |
| zdeflate.c_4 | 4 | 284 | 0.02% | 64 | 6.93% |
| | 5 | 9302 | 0.51% | 440 | 47.62% |
| | 6 | 122413 | 6.75% | 482 | 52.16% |
| | 7 | 580856 | 32.01% | 2 | 0.22% |
| | 8 | 797795 | 43.97% | 0 | 0.00% |
| | 9 | 280278 | 15.45% | 0 | 0.00% |
| | 10 | 23472 | 1.29% | 0 | 0.00% |
| Average overhead | | 7.71 | | 5.80 | |
| tone_detector.c_0 | 4 | 68 | 0.34% | 36 | 23.38% |
| | 5 | 932 | 4.62% | 95 | 61.69% |
| | 6 | 6870 | 34.08% | 23 | 14.94% |
| | 7 | 10731 | 53.23% | 0 | 0.00% |
| | 8 | 1559 | 7.73% | 0 | 0.00% |
| Average overhead | | 6.63 | | 4.92 | |
| jdmerge.c_1 | 4 | 648 | 0.04% | 106 | 10.61% |
| | 5 | 18308 | 1.01% | 413 | 41.34% |
| | 6 | 171960 | 9.48% | 480 | 48.05% |
| | 7 | 629580 | 34.70% | 0 | 0.00% |
| | 8 | 713080 | 39.30% | 0 | 0.00% |
| | 9 | 261384 | 14.41% | 0 | 0.00% |
| | 10 | 19440 | 1.07% | 0 | 0.00% |
| Average overhead | | 7.60 | | 5.37 | |

Table III: Number of layouts with a specific address-computation overhead, for the entire solution space for the MiBench suite (part 1)

| Access Sequence | overhead (cycles) | Exhaustive Number of Layouts | Exhaustive % of Layouts | Algorithmic Layout Frequency | Algorithmic % of Layouts |
|---|---|---|---|---|---|
| timing.c_4 | 4 | 188 | 0.01% | 116 | 1.20% |
| | 5 | 6222 | 0.34% | 1085 | 11.27% |
| | 6 | 99796 | 5.50% | 4510 | 46.83% |
| | 7 | 539947 | 29.76% | 3920 | 40.70% |
| | 8 | 828763 | 45.68% | 0 | 0.00% |
| | 9 | 309748 | 17.07% | 0 | 0.00% |
| | 10 | 29736 | 1.64% | 0 | 0.00% |
| Average overhead | | 7.79 | | 6.27 | |
| convert.c_0 | 3 | 1 | 0.00% | 8 | 0.82% |
| | 4 | 289 | 0.16% | 22 | 2.27% |
| | 5 | 5261 | 2.90% | 240 | 24.74% |
| | 6 | 37349 | 20.58% | 700 | 72.16% |
| | 7 | 85422 | 47.08% | 0 | 0.00% |
| | 8 | 47538 | 26.20% | 0 | 0.00% |
| | 9 | 5580 | 3.08% | 0 | 0.00% |
| Average overhead | | 7.05 | | 5.68 | |
| speed_control.c_9 | 4 | 218 | 1.08% | 36 | 26.47% |
| | 5 | 2547 | 12.63% | 100 | 73.53% |
| | 6 | 7880 | 39.09% | 0 | 0.00% |
| | 7 | 8435 | 41.84% | 0 | 0.00% |
| | 8 | 1080 | 5.36% | 0 | 0.00% |
| Average overhead | | 6.38 | | 4.74 | |
| jccolor.c_0 | 4 | 176 | 0.01% | 72 | 0.75% |
| | 5 | 6140 | 0.34% | 686 | 7.13% |
| | 6 | 95145 | 5.24% | 3550 | 36.87% |
| | 7 | 517317 | 28.51% | 5320 | 55.26% |
| | 8 | 824619 | 45.45% | 0 | 0.00% |
| | 9 | 342095 | 18.85% | 0 | 0.00% |
| | 10 | 28908 | 1.59% | 0 | 0.00% |
| Average overhead | | 7.82 | | 6.47 | |
| lame.c_39 | 4 | 324 | 0.02% | 16 | 1.64% |
| | 5 | 8086 | 0.45% | 60 | 6.15% |
| | 6 | 109891 | 6.06% | 600 | 61.48% |
| | 7 | 542770 | 29.91% | 300 | 30.74% |
| | 8 | 797567 | 43.96% | 0 | 0.00% |
| | 9 | 329278 | 18.15% | 0 | 0.00% |
| | 10 | 26484 | 1.46% | 0 | 0.00% |
| Average overhead | | 7.78 | | 6.21 | |

Table IV: Number of layouts with a specific address-computation overhead, for the entire solution space for the MiBench suite (part 2) and the DSPStone suite

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.

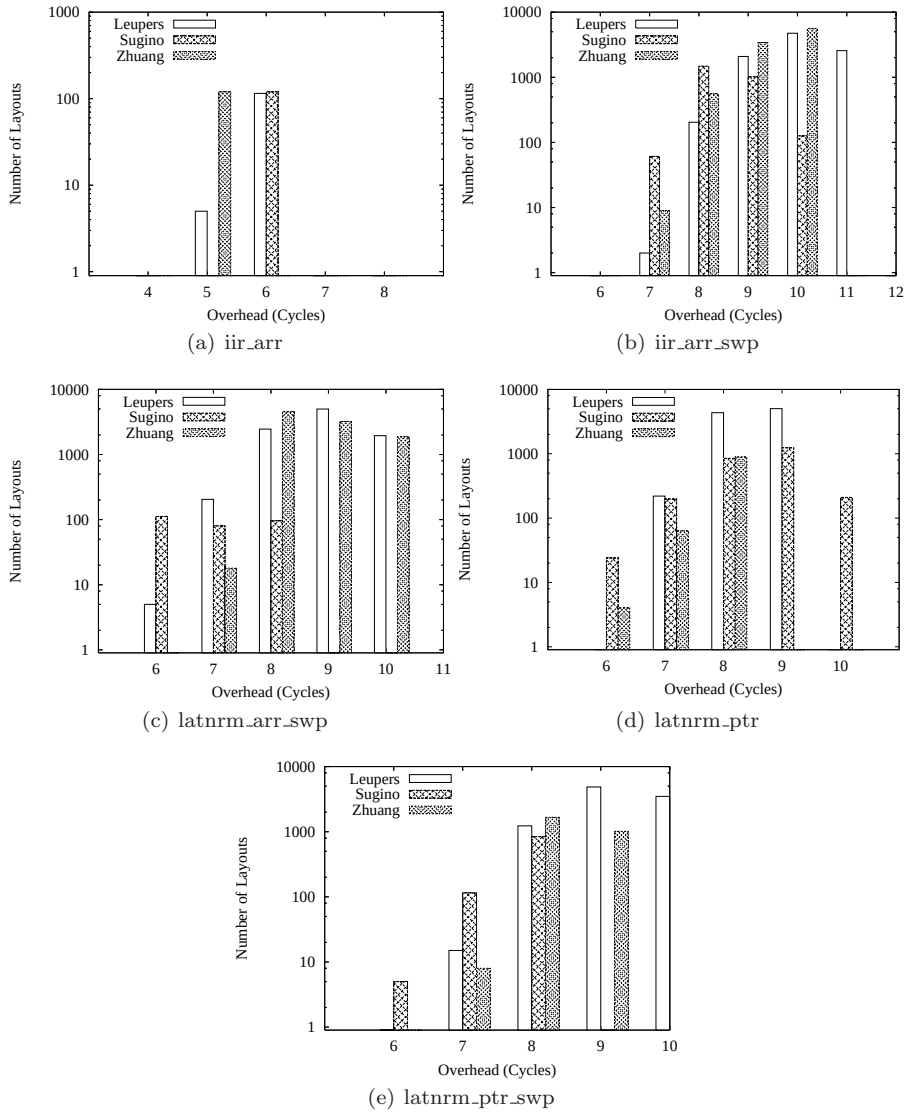


Fig. 8: Distribution of overhead values produced by each ARA algorithm on different test cases. The number of layouts shown for each algorithm is the union of 5 sets of layouts, each produced with one of the 5 different SOA algorithms, but using the same ARA algorithm. The full range of overhead values (obtained by exhaustive search) is plotted, regardless if an algorithm produced a layout with the specified overhead.

The above mentioned heuristic-based algorithms, and the MCC technique, can produce a memory layout and optimal addressing code in polynomial time, and thus, is a feasible approach to minimizing address computation overhead in a production compiler. Several algorithms were not included in our study because they may not be practical to use in a production compiler. Ozturk *et al.* propose an integer linear programming (ILP) formulation that generates optimal addressing code (similar to the MCC technique) as well as an optimal memory layout [Ozturk et al. 2006]. However, solving multiple ILP problems during compilation is not practical. Additionally, the approach does not model address register initialization costs, which can result in more registers being used than necessary; thereby, increasing address-register pressure. Atri *et al.* propose an SOA algorithm that iteratively improves a given memory layout [Atri et al. 2001]. Similarly, Wess and Zeitlhofer propose to approximate a solution to GOA by iteratively modifying offset assignments and address register assignments [Wess and Zeitlhofer 2004]. These algorithms are omitted because their performance depends on the initial memory layout. Leupers and David use genetic algorithms to find GOA

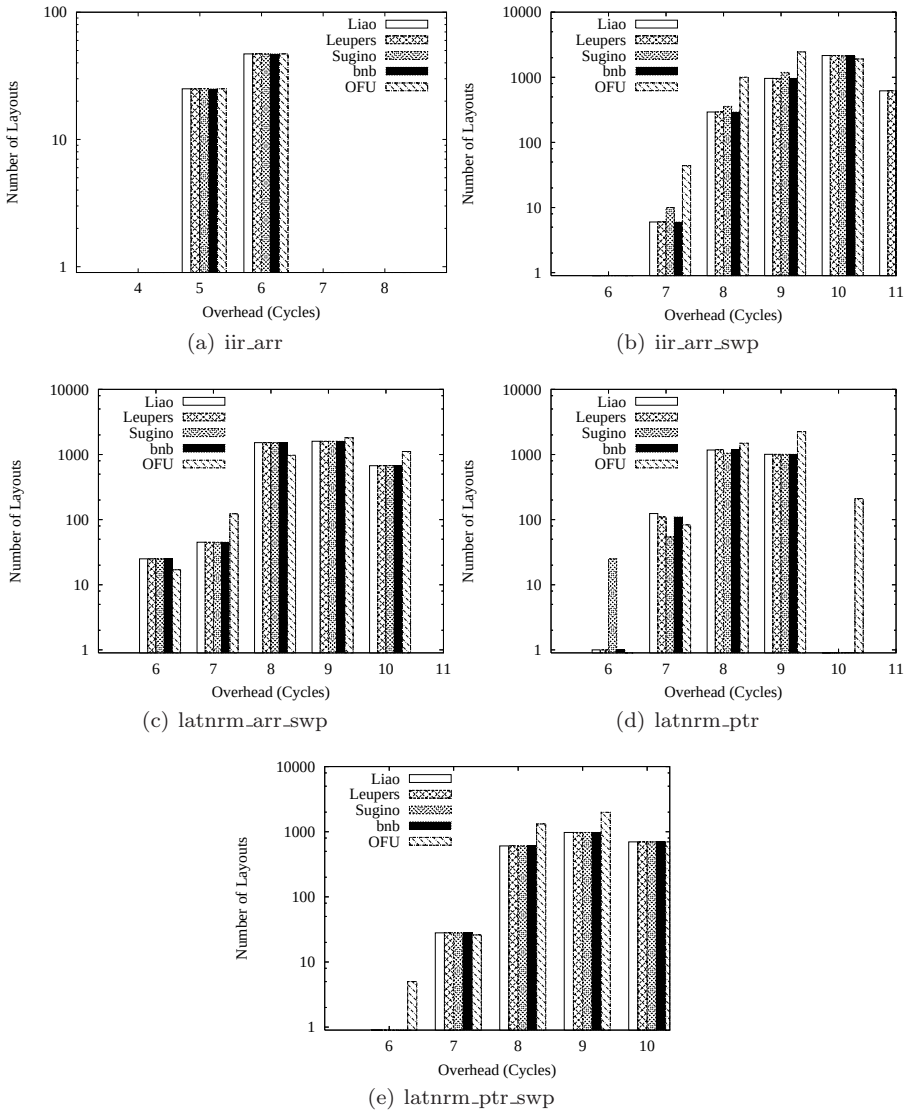


Fig. 9: Distribution of overhead values produced by each SOA algorithm on different test cases. The number of layouts shown for each algorithm is the union of 3 sets of layouts, each produced with one of the 3 different ARA algorithms, but using the same SOA algorithm. The full range of overhead values (obtained by exhaustive search) is plotted, regardless if an algorithm produced a layout with the specified overhead.

memory layouts [Leupers and David 1998], while Wess and Gotschlich generate memory layouts using simulated annealing [Wess and Gotschlich 1998]. The running time of these iterative techniques can be very high because they require many simulation steps to find a low-overhead memory layout. Another drawback is that finding a fast, but accurate, *fitness* function is difficult.

This study only focused on evaluating offset assignment for *scalar* variables in straight line code; however, there have also been studies for *array* accesses in loops. Leupers and David propose a heuristic-based and branch-and-bound algorithm to improve AR usage for array accesses in loop bodies [Leupers et al. 1998]. Cheng and Lin propose a graph-based address register allocation and data re-ordering algorithm to reduce overhead for loop execution [Cheng and Lin 1998]. Chen and Kandemir present a scheme to transform arrays and reschedule array accesses to reduce overhead using minimum cost traversals of reference graphs [Chen and Kandemir 2005].

Although our study investigates algorithms that directly generate a memory layout, overhead can also be reduced by manipulating the access sequence through instruction scheduling and variable extraction (see Figure 1). Rao and Pande apply algebraic transformations (such as commutativity) on expression trees to produce a *least-cost access sequence* [Rao and Pande 1999]; Lim *et al.* manipulate the entire instruction schedule [Lim et al. 2001]. Kandemir *et al.* propose an algorithm to change the access sequence *after* a full or partial memory layout is formed for each basic block [Kandemir et al. 2003]. Choi and Kim propose a unified algorithm to simultaneously find an instruction schedule and a low-overhead offset assignment [Choi and Kim 2003]. After an instruction schedule is found, the access sequence of variables must be extracted. Ottoni *et al.* coalesce variables and simultaneously find an SOA memory layout [Ottoni et al. 2006]. Similarly, Zhuang *et al.* propose algorithms that coalesces variables for both SOA and GOA [Zhuang et al. 2003]. 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.

## 8. 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


Atri, S., Ramanujam, J., and Kandemir, M. 2001. Improving Offset Assignment for Embedded Processors. *Lecture Notes in Computer Science 2017*, 158–172.

Bartley, D. H. 1992. Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software – Practice & Experience 22,* 2 (February), 101–110.

Chen, G. and Kandemir, M. 2005. Optimizing Address Code Generation for Array-Intensive DSP Applications. In *CGO '05:*

*Proceedings of the international symposium on code generation and optimization*. IEEE Computer Society, Washington, DC, USA, 141–152.

Cheng, W.-K. and Lin, Y.-L. 1998. Addressing optimization for loop execution targeting DSP with auto-increment/decrement architecture. In *ISSS '98: Proceedings of the 11th international symposium on system synthesis*. IEEE Computer Society, Washington, DC, USA, 15–20.

Choi, Y. and Kim, T. 2003. Address Assignment in DSP Code Generation — An Integrated Approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 22,* 8 (August), 976–984.

Gebotys, C. 1997. DSP Address Optimization using a Minimum Cost Circulation Technique. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society, Washington, DC, USA, 100–103.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*. Austin, TX, 3–14.

Kandemir, M. T., Irwin, M. J., Chen, G., and Ramanujam, J. 2003. Address Register Assignment for Reducing Code Size. In *CC '03: Proceedings of the 12th international conference on Compiler Construction*. 273–289.

Lee, C. and Stoodley, M. 1992. UTDSP BenchMark Suite. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.

Leupers, R. 2003. Offset Assignment Showdown: Evaluation of DSP Address Code Optimization Algorithms. In *CC '03: Proceedings of the 12th international conference on Compiler Construction*. 290–302.

Leupers, R., Basu, A., and Marwedel, P. 1998. Optimized Array Index Computation in DSP Programs. In *Asia and South Pacific Design Automation Conference*. 87–92.

Leupers, R. and David, F. 1998. A Uniform Optimization Technique for Offset Assignment Problems. In *ISSS '98: Proceedings of the 11th international symposium on System synthesis*. IEEE Computer Society, Washington, DC, USA, 3–8.

Leupers, R. and Marwedel, P. 1996. Algorithms for Address Assignment in DSP Code Generation. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*. 109–112.

Liao, S. 1996. Code Generation and Optimization for Embedded Digital Signal Processors. Ph.D. thesis, Massachusetts Institute of Technology.

Liao, S., Devadas, S., Keutzer, K., Tjiang, S., and Wang, A. 1996. Storage Assignment to Decrease Code Size. *ACM Transactions on Programming Languages and Systems 18,* 3, 235–253.

Lim, S., Kim, J., and Choi, K. 2001. Scheduling-Based Code Size Reduction in Processors with Indirect Addressing Mode. In *CODES '01: Proceedings of the 9th international symposium on Hardware/software codesign*. ACM Press, New York, NY, USA, 165–169.

Ottoni, D., Ottoni, G., Araujo, G., and Leupers, R. 2006. Improving Offset Assignment through Simultaneous Variable Coalescing. *ACM Transactions on Embedded Computing Systems (TECS) 5,* 4 (November), 864–883.

Ozturk, O., Kandemir, M., and Tosun, S. 2006. An ilp based approach to address code generation for digital signal processors. In *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*. ACM, New York, NY, USA, 37–42.

Rao, A. and Pande, S. 1999. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSP. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 128–138.

Sugino, N., Iimuro, S., Nishihara, A., and Jujii, N. 1996. DSP Code Optimization Utilizing Memory Addressing Operation. *IEICE Trans Fundamentals* 8 (Aug), 1217–1223.

Wess, B. and Zeitlhofer, T. 2004. On the Phase Coupling Problem Between Data Memory Layout Generation and Address Pointer Assignment. In *SCOPES*. 152–166.

Wess, B. R. and Gotschlich, M. 1998. Minimization of Data Address Computation Overhead in DSP Programs. In *Proc. ICASSP98*. 3093–3096.

Zhuang, X., Lau, C., and Pande, S. 2003. 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*. ACM Press, New York, NY, USA, 220–231.

Zinojnovic, V., Velarde, J. M., Schlager, C., and Meyr, H. 1994. DSPStone": A DSP-Oriented Benchmarking Methodology. In *5th International Conference on Signal Processing Applications and Technology*. Dallas, TX.

Received Month Year; revised Month Year; accepted Month Year