An Optimization Framework for Embedded Processors with Auto-Modify Addressing Modes

A Dissertation
Presented to
The Academic Faculty

By

ChokSheak Lau

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science in Computer Science

Georgia Institute of Technology
December 2004

An Optimization Framework for Embedded Processors
with Auto-Modify Addressing Modes

Approved by:

Dr. Santosh Pande, Committee Chair
College of Computing, Georgia Tech

Dr. Hsien Hsin S. Lee, School of
Electrical and Computer Engineering,
Georgia Tech

Dr. Gang-Ryung Uh, Computer
Science, Boise State University

Date Approved: November 2004

He has made everything beautiful in its own time; also He has put eternity in their heart, yet so that man does not find out what God has done from the beginning to the end.

-- King Solomon, the son of King David, Ecclesiastes 3:11

This dissertation is dedicated to the one and only true God.

ACKNOWLEDGEMENTS

My greatest gratitude would be towards my professor, Dr. Santosh Pande, who has directed my studies and work in compiler research the past two years. Without him, I would not be where I am today. Special thanks to Xiaotong Zhuang, a fellow student, who contributed greatly to part of the work presented.

Also thanks to the One who commanded that life should be lived without any anxiety, bringing peace and rest to me who never had peace and rest. The same One who created heaven and earth, light, the Sun and the Moon, all the planets, and all of life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AG	Access Graph
AR	Address Register
ARA	Array Reference Allocation
AGU	Address Generation Unit
BB	Basic Block
CFG	Control Flow Graph
CG	Coalescence Graph
DAG	Directed Acyclic Graph
DSP	Digital Signal Processing
GCC	GNU Compiler Collection
GNU	GNU's Not UNIX
GOA	General Offset Assignment
IG	Interference Graph
MWPC	Maximum Weight Path Cover
LDAR	Load Address instruction, or any AR modification instruction
PC	Path Cover
SOA	Simple Offset Assignment

SUMMARY

Modern embedded processors with dedicated address generation unit support memory accesses using indirect addressing mode with auto-increment and auto-decrement. The auto-modify mode, if properly utilized, can save address arithmetic instructions, reduce static and dynamic footprint of the program and speed up the execution as well.

[Liao 1995; 1996] categorized this problem as the simple offset assignment (SOA) problem and the general offset assignment (GOA) problem which involve storage layout of variables and assignment of address registers respectively. He proposed heuristic solutions to these problems based on graph-theoretic algorithms. Later work proposed improvements in the performance of Liao's solution by undertaking other heuristics for offset assignment and also by undertaking program transformations which rearrange the sequence of accesses (called access sequence) to the memory locations.

Since techniques based on devising efficient graph covering algorithms have limited impact given the density of the underlying access graph, this work proposes a new direction to explore the solution space for this problem. The work proposes a framework to simplify the access graph using coalescence-based offset assignment, post-pre optimizations and using offset registers. Variables not interfering with other (not simultaneously live at any program point) can be coalesced into the same memory location. Coalescing allows simplifications of the access graph yielding better SOA solutions or can perhaps lead to such a small number of non-coalesceable memory locations that GOA solutions for them are optimal. Moreover, it can reduce the program footprint both statically and at runtime (for stack variables) in terms of data segment size.

Besides, variable coalescence is orthogonal to other heuristics proposed by early work. We have seamlessly incorporated our framework with an SOA solver. Our framework can work with any SOA solvers, making the scheme more flexible. Post-pre optimization considers how to do most effective code generation using both post-modify and pre-modify modes to solve the challenge of utilizing this mode within basic blocks as well as across basic block boundaries. Making use of both addressing modes further reduces effective SOA/GOA cost and our post-pre optimization phase is optimal in selecting post or pre mode after variable offsets have been determined.

Our experiments conducted on benchmark programs from MediaBench, MiBench and Spec2000Int showed improved code performance in terms of stack size, the number of address arithmetic instructions and execution cycles. We were able to obtain an average of 12.0% reduction in dynamic stack size in a compiler that reuses stack slots, so the actual savings could have been greater if the compiler were to not reuse stack slots. We base our comparisons against a base SOA algorithm, which is Liao's SOA with Leupers and Marwedel's tie-breaker. By using offset registers, we achieved a 36.5% reduction in address arithmetic instructions, compared to 2.77% for base SOA algorithm. For code size, we saved 2.27% compared to 0.32% for base SOA. For execution cycles, we saved 4.10% while base SOA saved 0.38%.

1. INTRODUCTION

The rapid evolution in embedded processors and DSP architectures has raised new challenges for compilers to generate efficient and small footprint code for the ever-increasing demands on user applications. Reducing the code size also reduces the amount of memory traffic for instruction fetching and data fetching, which can further speed up the program execution.

Memory is often a scarce resource in embedded systems because of their small size. Therefore, we want to optimize code with respect to both code size and stack size, because both of them consume memory.

Most modern embedded architectures have specialized address generation units (AGUs) to facilitate the memory address generation in different modes. The AGU normally provides auto-modify mode, i.e. simple Address Register (AR) operation (typically, plus or minus a small constant value) before or after the memory access operation, so that the address register operation is executed for free without dilating the clock cycle on the critical path. However, due to constraints on instruction size, traditional register-plus-offset addressing mode is either not supported (e.g. TMS320C25) or requires more instruction words (Motorola DSP56300). Therefore, transforming address arithmetic into auto-modify mode can help to generate compact and efficient code and speed up execution as well.

Most modern DSP processors have at least 8 address registers. For example, each of the Motorola DSP56300 processor [Motorola 2000] and the Sony pDSP processor has 8 address registers. StarCore's SC140 has 16 address registers [Motorola 2001]. Analog Devices' ADSP-21020 has 8 address registers (32 bit) for data memory and 8 address

registers for program memory (24 bit). Post-modify is supported for all these processors, and pre-modify is supported for some processors like DSP56300. The hardware support reflects the designers' expectation for heavy usage of these instructions; however the actual usage of them is still quite limited. In our experiments, we counted the number of instructions with auto-modify modes generated by GCC compiler retargeted for the Motorola DSP56300 processor. For most benchmark programs, less than 3% of the generated address instructions make use of the auto-modify mode before our optimizations. A recent study [Udayanarayanan 2001] also shows that on some embedded processors up to 55% of operations could potentially use address register operations to reduce cycle counts and code size. Therefore, significant opportunities exist for optimizing address register assignments.

Bartley [1992] and Liao et al [1995; 1996] first modeled this problem as offset assignment (also known as storage assignment). They identified the problem as two classes: simple offset assignment (SOA) and general offset assignment (GOA). They modeled the problem as an access graph and the objective is to find the maximum weight path cover (MWPC) on the graph. Liao proved that finding the MWPC is NP-complete; therefore heuristics are used to solve both SOA and GOA. Later, Leupers and Marwedel [Leupers 1996] extended Liao's work by proposing a Tie-break heuristic for SOA and a variable partitioning strategy for GOA to reduce the SOA and GOA costs. Atri, Ramanujam and Kandemir [Atri 2000] further improved the heuristics by an algorithm called Incremental-Solve-SOA, which requires much more running time in solving the same graph problem. Sudarsanam et. al. [1997] studied the offset problem in the presence of an auto-modify feature that varies from -l to +l with k address registers. [Rao 1998;

Rao 1999] extended beyond offset assignment with memory access sequence reordering (or program reordering) through algebraic transformations on the expression trees. [Kandemir 2003] proposed a more aggressive access sequence reordering scheme with both intra-statement and inter-statement transformations. Program reordering can better utilize the auto-modify mode by rearranging not only the variables' offsets but also the order of memory access instructions. An approach based on a genetic algorithm (GA) for SOA was presented in [Leupers 1998]. It uses a simulation of natural evolution process, which is relatively time-consuming. Finally, [Leupers 2003] did a comprehensive comparison among several existing algorithms (except program reordering) and proposed a combined algorithm based on Tie-break and incremental-Solve-SOA. He also found that the qualities of the solutions obtained are quite close among these algorithms.

Another type of problem is known as the Array Reference Allocation (ARA), which optimizes the access to array variables using auto-modify mode [Araujo 1996; Gebotys 1997; Leupers 1998; Ottoni 2001].

In this work, we propose an optimization framework for compiler-managed code generation based on the auto-modify mode on embedded processors. Previous approaches to offset assignment optimization concerns dealing with graph-theoretic algorithms and algebraic transforms to find a good memory layout, but do not provide any stack size savings. We want to optimize stack memory and also simplify the solution by making use of other techniques. Our framework consists of two parts. First, we enhance the effectiveness of offset assignment with a new technique called variable coalescence. We start with identifying webs, and then we coalesce them aggressively into fewer memory locations. Our study shows that the access graph of the atomic variables is sparse, and

coalescence can effectively reorganize them to generate simpler access sequences with high-weighted path covers. Besides, aggressive coalescence can significantly reduce the static and dynamic memory space requirements of a program for SOA and GOA based optimizations. Variable coalescence can be combined with most previous approaches to further boost the performance. Second, to further reduce the AR modification instructions (written as "LDARs" for short), we add a post-pre optimization phase to decide whether post- or pre-modify mode should be used for each access. Our post-pre optimization phase can optimally select post or pre mode after variable offsets have been determined. We also propose additional optimization methods to consider SOA as an inter-basic-block problem, and to scavenge the offset registers which can be used to save LDARs and execution cycles.

1.1 Address Generation in DSP Processors

Address generation hardware in DSP processors differs from that of standard processors [Leupers 1996]. Usually, several ARs are available, which can be updated in parallel to other machine operations, thereby introducing no code size or speed overhead. On the other hand, addressing may be quite restricted. In order to avoid long combinational delay, many DSPs do not permit indexing with an offset, but only post-modification, i.e. additions or subtractions involving ARs take place only at the end of a machine cycle. Besides high code quality, retargetability is another primary goal in DSP code generation, due to the growing diversity of DSPs in form of application-specific designs (ASIPs). Therefore, we consider a generic AGU architecture, which reflects a subset of AGU capabilities of many contemporary DSPs. Our AGU model (Figure 1) is parameterized by the number K of ARs and the number N of Offset Registers (ORs). In

case of multiple memory banks we assume separate AGUs for each bank. Typical AGU configurations are K = 4; N = 4 (ADSP-210x), or K = 8; N = 1 (TMS320C2x). The ARs provide effective memory addresses, while ORs store integer modify values for AR updates. AR and OR files are indexed by designated pointers, which select the current AR and OR for each machine cycle. AR and OR pointer updates usually does not contribute to code size [Leupers 1996]. Figure 1 shows a generic AGU model taken from Leupers and Marwedel's paper [Leupers 1996].

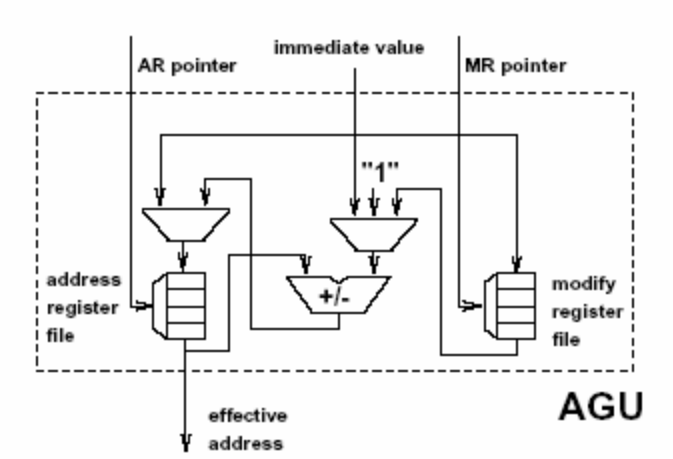


Figure 1. Generic AGU Model

The AGU model permits execution of the following primitive AGU operations in each machine cycle: 1) Immediate AR load: The current AR is loaded with an immediate value supplied by the instruction word. 2) Immediate AR modify: An immediate value is added to or subtracted from the current AR. 3) Auto-increment/decrement: The constant 1 is added to or subtracted from the current AR. 4) Immediate OR load: The current OR is loaded with an immediate value. 5) Auto-offset-modify: The contents of the current OR are added to or subtracted from the current AR.

1.2 The Offset Assignment Problem

Compiler optimizations for auto-modify addressing modes can be classified into two types: single-AR and multiple-AR, depending on the number of available address registers.

An AR modification instruction is written as “LDAR” for short. “LDAR” means “load address” and it means an instruction that sets an AR to a certain immediate address value. There are two other kinds of address arithmetic instruction, “ADAR” and “SBAR”. “ADAR” is an instruction that adds an immediate integer to an AR. “SBAR” is an instruction that subtracts an immediate integer to an AR. To simplify writing, we use “LDAR” to mean any one of the LDAR/ADAR/SBAR instructions.

Traditionally, it is studied as the Offset Assignment Problem. Offset assignment is to assign offsets (memory layout) to variables so that the number of address arithmetic instructions can be minimized by using auto-modify modes of register indirect addressing instructions. Accordingly, Simple Offset Assignment (SOA) assumes single-AR, while General Offset Assignment (GOA) tackles multiple-AR. For example, Figure 4(a) shows the memory layout for 6 variables (address grows upwards) and generated code corresponding to Figure 3(a)—we assume one address register AR0, so it is an SOA problem. Here, we assume that variables on the right-hand-side of the equation must be loaded one-by-one from left to right, then, after the evaluation, the result is stored into the left-side variable. For the time being, all variables are stored in memory (in case they are not, the access graph will show the order of only those accesses corresponding to memory accesses, i.e. load/stores). For the first instruction c=a+b, after accessing b, i.e. ADD *(AR0)-, we use auto-decrement to point AR0 to the memory location of variable c,

thus saving one AR modification instruction. In principle, all ADARs and SBARs can be replaced by LDAR in usage. Therefore, the problem of maximizing the use of auto-modify instructions is to find a good memory layout such that a maximum number of consecutively accessed variables are adjacently stored in memory. To apply the offset assignment optimization, we need to find out the access sequence first. An access sequence is defined as an ordered linear sequence of variable accesses [Liao 1995; Liao 1996]. For example, in Figure 2, we show the access sequence below for the code segment. From the access sequence, we can build an access graph based on the access sequence (Figure 2). An access graph is a weighted undirected graph, on which each node is a variable, while the edge weight is the number of transitions in the access sequence between the two end nodes (variables). In other words, the edge weight represents the number of times the two nodes are accessed consecutively in the access sequence.

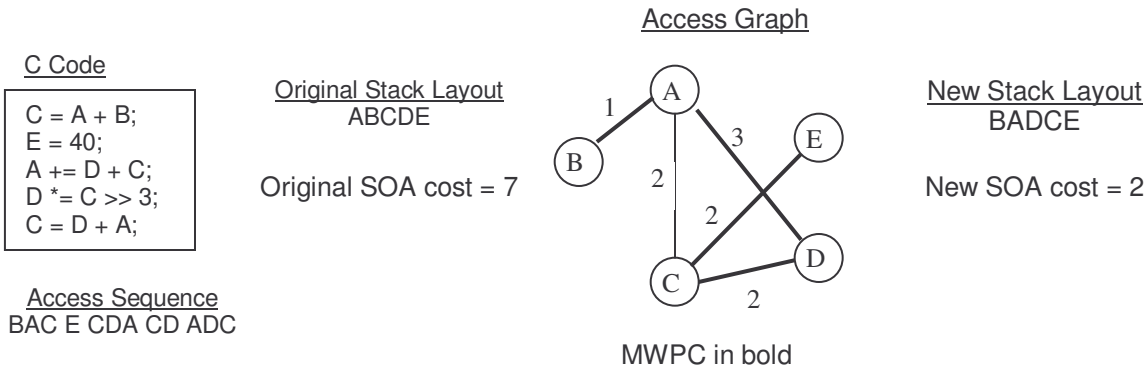


Figure 2. Example of SOA and Access Graph

After the access graph is constructed, our optimization objective is to find a maximum weight path cover (MWPC) [Liao 1995]. An MWPC is simply a path cover (PC) with maximum weight. Here, path cover is defined slightly differently from that in graph theory. The path cover here means an edge set such that 1) each node adjacent to

any edge in the edge set can only have either one or two neighbors in the edge set; 2) no cycle can be constructed solely with edges in the edge set. Intuitively, the subgraph with only edges in the edge set must consist of one or more linear path(s) so that the variables can be laid out linearly in memory. Weights covered on the PC is proportional to the number of times auto-modify modes can be used to access the next variable in memory, while the sum of the weights of all edges not on the PC is proportional to the number of times LDARs should be inserted, and this sum is called the SOA cost ([Liao 1995] gives details on the SOA cost. Intuitively, for uncovered edges, LDARs must be inserted and the edge weights now represents how many times these instructions are executed). The thick lines in the access path in Figure 2 shows one of the PC and also an MWPC solution. The weight for the MWPC is 7 and the SOA cost is 2. Earlier approaches [Liao 1995; Leupers 1996; Atri 2000] have shown that the MWPC problem is NP-complete and tried to find a good path cover with a weight close to the MWPC.

For example, in Figure 2, the original SOA cost of 7 means that in the access sequence, we can count 7 times for which we go from one variable to another for which the variables are not placed right next to each other in the stack layout. For the sequence BACECDACDADC, the 7 breaks in the access sequence are: AC, CE, EC, DA, AC, DA and AD. Using the MWPC solution, we can get a new SOA cost of only 2.

On the other hand, General Offset Assignment (GOA) is typically solved in two steps. During the first step, a heuristic algorithm assigns each variable to an address register, thus a variable assigned to an AR uses that AR only. Next, for all variables assigned to the same AR, the problem is solved as SOA. GOA cost is actually the sum of SOA costs associated with each address register. For GOA, the access sequence for

variables handled by one AR is derived from the all-variable access sequence but considering only the variables using that AR. For example, in Figure 3(a), if we have two address registers AR0 and AR1, and {a,b,c} is handled by AR0 and {d,e,f} is handled by AR1, then the access sequence for AR0 is abcacaaccb, the access sequence for AR1 is defddf.

In real programs with branches, the access sequence cannot be simply derived from static code during compilation time, because the compiler has little knowledge about the runtime execution trace. However, we can still construct the access graph in other ways. Notice that, on the access graph, the edge weight between two variables should indicate the frequency these two variables are accessed consecutively. In other words, as adopted in our experiments, we can use profile information to get the execution frequency for the path between two consecutive memory accesses to the variables. In case profile information is not available, we can roughly estimate the execution frequencies of the paths based on their loop depth [Muchnick 1997].

In addition to offset assignment, other approaches are possible to harness the auto-modify modes or to improve the effecti`veness of offset assignment. [Rao 1998; Rao 1999] proposed program reordering. Program reordering reschedules instructions according to the algebraic laws (like from a+b to b+a) so that a higher weight path cover solution can be obtained during offset assignment and more variable accesses can be covered with auto-modify mode.

In this work, we observe that the access graph is sparse in general, therefore coalescing nodes on the access graph might lead to a better MWPC solution based on offset assignment. After variable coalescence, the access graph can be much different,

inducing an improved solution even superior to the optimal MWPC that can be achieved without variable coalescence. Furthermore, variable coalescence can be combined with and improve all previously mentioned offset assignment approaches and it is applicable to both SOA and GOA. Secondly, our post-pre optimization comes after offset assignment and finds chances for both and post and pre addressing mode. In the next section, we will show a few examples to illustrate these optimizations.

1.3 Motivating Examples

1.3.1 Variable Coalescence

In Figure 3, we give an example to illustrate how variable coalescence works and how it can reduce the SOA and GOA cost.

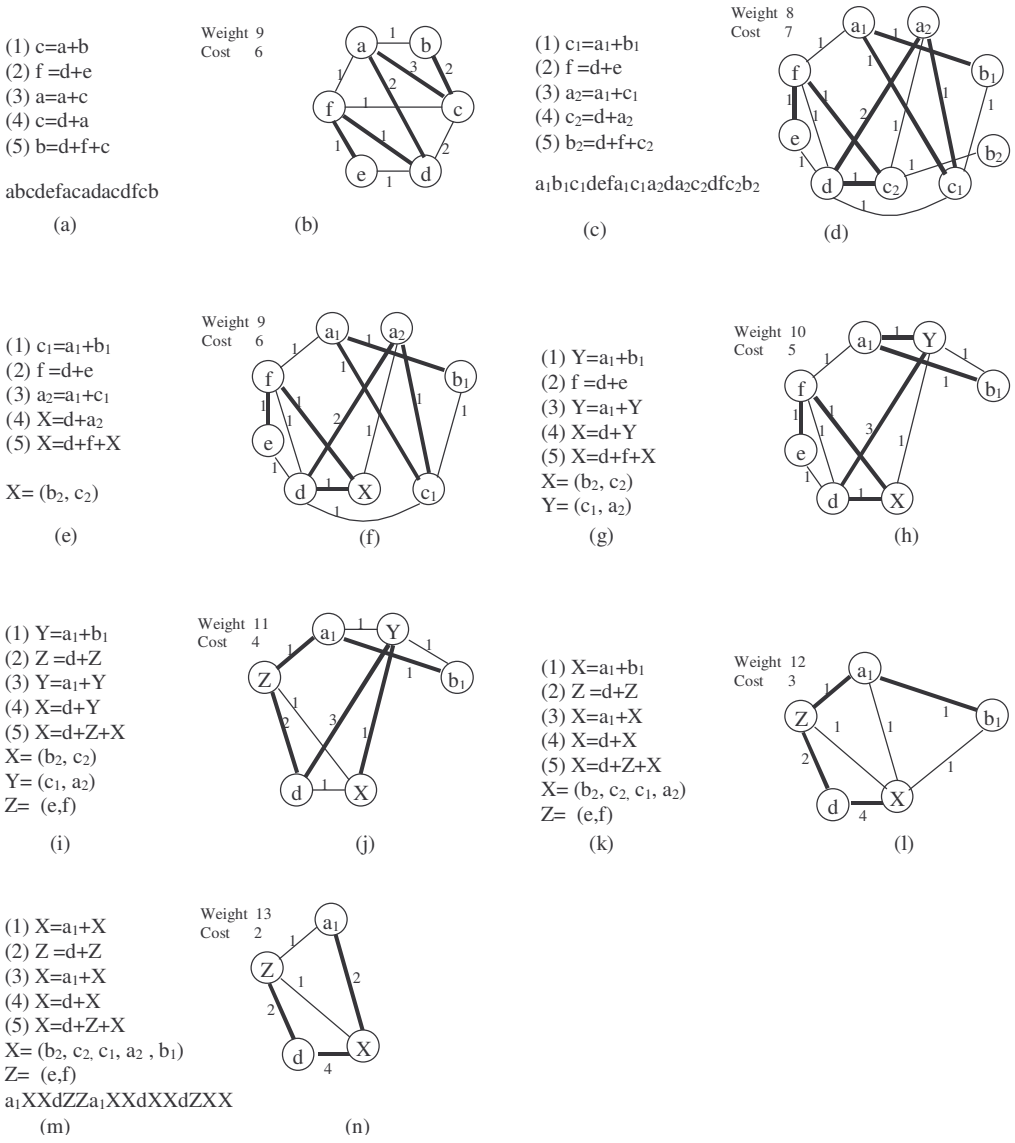
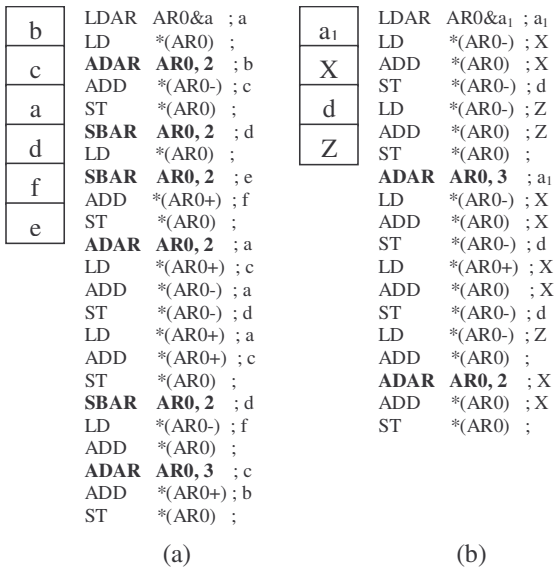


Figure 3. Motivating Example

The code segment in Figure 3(a) (taken from [Rao 1999] with minor changes) contains 5 instructions. We assume this code segment is the entire program itself. In real programs, we need to do liveness analysis and variable renaming/coalescing.

The coalescence algorithm actually first separates variables into atomic units called webs (explained in Chapter 3.2) [Muchnick 1997] through variable renaming. A web is a du/ud chain closure of a variable and allows independent allocation of values in memory.



*Note: variables on the right of semicolon is what AR0 points to after the instruction.

Figure 4. Assembly Code (a) Before, and (b) After Coalescence

Figure 3(c) shows how we separate each of variables a, b and c into two webs. Intuitively, in instruction (3), defining variable a starts a new web. We thus rename the variable a, then use that new name in later references. Similarly, b and c are renamed in instructions (4) and (5). In this code segment, c1, which is live from instructions (1) to (3), constitutes a closed web, c1 can be arbitrarily renamed regardless of other parts of the program. Figure 3(c) and Figure 3(d) show the access sequence and access graph after

variable separation. The weight of the MWPC is 1 unit smaller than the one before variable separation. In Figure 3(e) and Figure 3(f), we coalesce b2 with c2, i.e. we combine these two variables into one variable, putting them into the same memory location. Because the last use of c2 ends before the definition of b2, they can be safely coalesced as one variable X. Their edges are coalesced accordingly as shown in Figure 3(f). After coalescing, the cost is reduced by one (notice when we coalesce two variables, the weight of the edge between them is saved, since we do not need to modify the address register when consecutively accessing the same memory location). From Figure 3(g) to Figure 3(n), we coalesce 4 other nodes. The final MWPC weight is 13 (including edges between nodes that were coalesced together) with an improvement of 44%. Also, the data segment size is reduced from 6 variables to 4 variables (a 33% reduction). The final variable layout and modified code are listed in Figure 4(b). After saving 4 ADAR/SBAR instructions, we achieve a 17% code size reduction and 17% speedup (assuming all instructions require the same number of cycles).

We now discuss the effect of coalescing on GOA. Suppose 2 address registers AR0 and AR1 are available, for the code in Figure 3(m), we can simply assign two variables to each of them, e.g. {X, a1} to AR0, {Z, d} to AR1. The access sequence for {X, a1} as derived from the whole access sequence in Figure 3(m) is a1XXa1XXXXXX, thus the access graph has only one edge with weight 3, which is on the MWPC. Similarly, for {Z, d}, the solution is also optimal (SOA cost of 0). We will show in Chapter 3.7 that coalescence can often generate an optimal solution for GOA.

Figure 3(b) already shows the optimal solution of MWPC for the case of no coalescence, and therefore no heuristic can reduce the cost below 6 without variable

coalescence. For GOA, since variable coalescence already obtained the optimal solution, no other algorithm can do any better.

This example shows that by separating and coalescing the variables, we get better performance (fewer execution cycles) and code size. Using coalescing can often produce a solution with a lower SOA cost than the best MWPC that could possibly be obtained without coalescing. Also, coalescing gives a stack size savings which the other algorithms cannot give.3

1.3.2 Post-pre Optimization

This example illustrates post-pre optimization. As mentioned previously, both post- and pre-modify are supported for some embedded processors. However, current research on offset assignment does not consider pre-modify modes altogether. In Figure 5(a), assume that after offset assignment, the four variables are laid out sequentially as d,c,b,a (address grows upwards). Meanwhile, the four load instructions are distributed in 3 basic blocks. Based on the variable offsets, we can generate the final code as in Figure 5(b), where auto-address mode is only used once. 3 LDAR instructions have to be used to set the address register AR0. However, in Figure 5(c), we give another solution with the post-pre optimization. Although the two successors of variable c, i.e. a and b, have different offsets, with both post- and pre-modifies, we can avoid any LDARs. After accessing c, AR0 is post-incremented to point to variable b. On the path to BB2, AR0 is pre-incremented before accessing variable a. In the meantime, the SBAR instruction in BB3 can be avoided as well. AR0 is post-decremented and then pre-incremented before accessing variable d, so SBAR AR0, 2 can be removed. Notice that, post-pre

optimization is done after variable offsets have been assigned by the offset assignment algorithms.

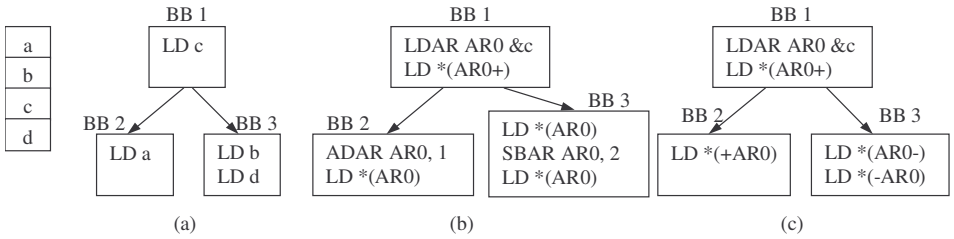


Figure 5. Example for Post-pre Optimization (a) Original Code and Offsets (b) Without Post-pre Optimization (c) With Post-pre Optimization

1.3.3 Inter-basic-block Offset Assignment

In a typical CFG, a basic block can have multiple predecessors and/or successor basic blocks. Therefore, the access sequence does not terminate along basic block boundaries. The problem of considering the continuation of access sequence even across basic block boundaries is called inter-basic-block offset assignment. In the case when a basic block P has a unique successor S, where S has a unique predecessor P, we can merge two access sequences into one longer access sequence. However, the problem becomes more complicated when an access sequence can take one of several different paths. In such a case, we try to continue the access sequence along the path which could possibly save an LDAR.

Figure 6 shows the case when the variables' stack layout is (a, b, c) and variable a can be followed by either b or c in a CFG split point. In the original access graph, we have only two edges that we can pick from, either (a, b) or (a, c). Since edge (a, b) has a greater weight, we pick it. However, note that the weight of edge (a, b) overlaps with the weight of edge (a, c). This means that we can either count edge (a, b), or count edge (a, c),

but not both at the same time. Why is this so? This is because of the CFG split. The topmost basic block, BB1, has two successors BB2 and BB3. The last memory access in BB1 is variable a. The first memory access in BB2 is b, while the first memory access in BB3 is c. Therefore, we have two different access sequences at the bottom of BB1. It could be either a-b, or a-c, but not both.

Since we pick (a, b), edge (a, c) loses a weight of one. This is because we chose to realize the address of variable b at the bottom of BB1, which is the access sequence from BB1 continuing on to BB2. b has a different address from c, so if we realize the address of b, we cannot realize the address of c at the bottom of BB1. Therefore, BB3 needs an LDAR at the top of the basic block. If we had picked (a, c) instead, the weight of edge (a, b) would not be 3 anymore, but would become 2. Vice-versa, if we choose to go from BB1 to BB3, then that will break the access sequence from BB1 to BB2.

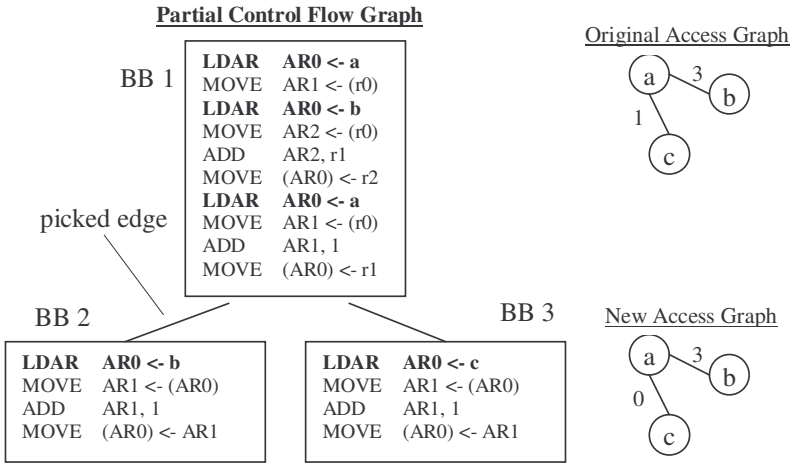


Figure 6. Example of Access Graph being Modified

Thus, the access graph changed as we picked edges during offset assignment. In this way, the access graph dynamically evolves during the process of selecting access graph edges and by so doing, we take into account access sequence across basic block

boundaries. In the case when we do not consider any inter-basic-block access sequences, we need one LDAR for each of variables b and c because they appear as the first load/store instruction in the basic block, which is the worst case possible. This worst case corresponds to the case of intra-basic-block offset assignment. In general, any intra-basic-block offset assignment scheme always needs at least one LDAR at the top of each basic block in which at least one offset-assigned variable exists.

1.4 Offset Registers Optimization

Offset registers are the special set of registers in typical DSP processors that allow an offset to be applied to an address register without incurring any additional execution cycles. Therefore, when we want to access a particular memory location with a known offset from the current address register value, we can either modify the address register directly, or modify an offset register and use it with a base address register. The offset register is particularly useful when referring to stack memory because by modifying the offset register, we do not have to modify the stack pointer register. Also, unaliased stack variables always have a fixed, pre-known offset from the stack pointer, and hence we can reuse the same offset value to point to the same stack variable.

Since DSP instructions only support a modification by one offset in auto-modify modes, offset registers can be used to reach those variables which are placed more than one offset location away in memory. This is especially useful when the offset register is pre-assigned a small fixed value, typically 2 or 3, and this makes it possible for us to save LDARs even when the access sequences are separated by fixed offsets that are not reached by using auto-increment or auto-decrement.

2. OVERALL FRAMEWORK

2.1 Outline

Figure 7 shows the overall framework for the optimizations with auto-modify mode. Figure 7(a) shows the optimization flowchart for single-AR and Figure 7(b) shows the one for multiple-AR. In both cases, two optimization objectives are considered during coalescence and offset assignment, leading to two kinds of algorithms. We propose two kinds of heuristics to minimize the SOA or GOA cost, which corresponds to address modification code (LDAR/ADAR/SBAR). Algorithm "OpCost" targets the incremental minimization of SOA or GOA cost, while algorithm "OpSize" aims to minimize the nodes on the access graph (or the runtime memory space these variables take) through aggressive coalescence. As a starting point, we need to build the access graph (AGs) and interference graphs (IGs). These two graphs are necessary to guide the coalescence and offset assignment process.

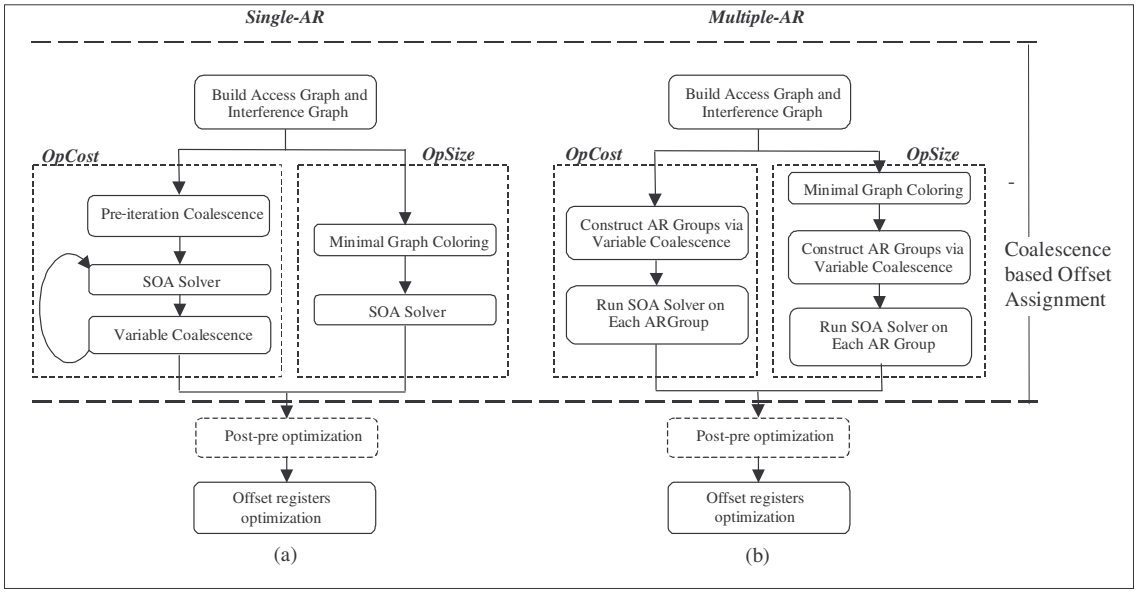


Figure 7. Optimization Framework

In Figure 7(a), both OpCost and OpSize invoke an SOA solver, which could be any one of the previous offset assignment algorithms without variable coalescence. The SOA solver only assigns offsets for given variables and attempts to minimize the SOA cost. For the OpCost algorithm, a heuristic approach is chosen to iterate over MWPC searching and variable coalescence after the pre-iteration coalescence is done (explained in Chapter 3.6). In each iteration, the heuristic algorithm finds 2 nodes to coalesce if possible. Then, the two nodes are coalesced and the access graph and interference graph are changed accordingly. The solution with the least cost ever achieved is saved and used as the final solution. On the other hand, OpSize simply coalesces the nodes maximally through a graph coloring algorithm, then runs the SOA solver to obtain a solution.

In Figure 7(b), with multiple ARs, the algorithm classifies variables into several AR groups, so each group can be assigned to one AR and solved with a single-AR algorithm. The OpCost algorithm constructs AR groups together with variable coalescence, then runs the SOA solver afterwards on each AR group. In contrast, the OpSize algorithm aggressively coalesces the nodes by minimally coloring the IG, since the minimal number of nodes can lead to optimal solutions in many cases. In case the optimal solution cannot be given out after graph coloring, we apply the coalescence algorithm as in OpCost.

After variable coalescence and offset assignment, we optionally perform post-pre optimization if both post- and pre-modify modes are supported. This is a cheap operation and the algorithm does not take much more time to run than SOA itself. Finally we use the offset registers as far as we can to cover the rest of the remaining breaks in access sequences so that we can further save LDARs.

Clearly, our framework incorporates more optimizations than solely assigning offsets for variables. Coalescence-based offset assignment is the phase in which we perform variable coalescence together with offset assignment. We will discuss this in detail in Chapter 3. We will make use of an SOA solver from early "offset assignment only" approaches. Post-pre optimization will be discussed in Chapter 4.

2.2 Assumptions

Most of the basic assumptions are followed from previous work [Bartley 1992; Liao 1995; Liao 1996; Rao 1998; Rao 1999; Leupers 1996]. We list some specific ones as follows:

1) This work only considers auto-modify addressing with stride 1, which means the address register can only be increased or decreased by 1 in each instruction that has the auto-modification. Auto-modify with stride 1 only is the most widely supported auto-modify mode on state-of-the-art embedded architectures.

2) Not all address register operations can be converted into auto-modify mode addressing. For instance, some address registers can point to multiple variables depending on the direction of the control flow or due to multiple aliasing; thus, we cannot bind it to one single variable since it would be unsafe to optimize it as auto-increment or auto-decrement for a given layout. Thus, in a multiple alias case, one has to use explicit address register modification (like LDAR, ADAR, SBAR in Figure 4) operations.

3) In addition, array index based optimizations have been an active area of research and there are techniques to analyze array-indexed memory accesses, esp. in loops [Ottoni 2001; Araujo 1996; Gebotys 1997; Leupers 1998; Zhang 2003]. However, such research work is entirely different from offset assignment optimizations for scalar

variables in terms of the problem formulation and approaches. Currently, we consider it beyond the scope of this article.

3. COALESCENCE-BASED OFFSET ASSIGNMENT

3.1 Use of Alias Analysis

The framework starts with performing a simple alias analysis [Aho 1986] to determine the variables that might be referenced via pointers. For a given variable P, where P is a pointer, we can determine what P points to if P is locally assigned across all reaching paths in the CFG of the function before P is first used. Consider the CFG in Figure 8. P is a pointer to an integer, and A and B are two local variables. The CFG then splits into two possible paths, with one path into BB 2 setting P to the address of A, and the other path into BB 3 setting P to the address of B. Therefore, in BB 4. we know that P can point to either A or B, and no other locations. We also know that A and B do not have the same stack memory address. Therefore, P is multiply-aliased.

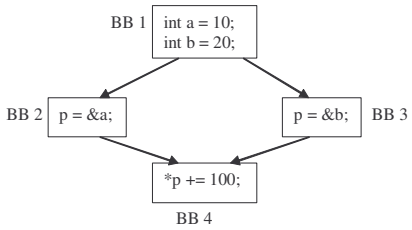


Figure 8. Illustration of Possible Multiple Aliasing

Another case is when a pointer P is first used before its definition in the function, or P is assigned a value which came through as a function argument value or from some external unknown value. In these cases, we cannot determine an alias for P. But we do not treat any unknown aliases as possibly pointing to any local stack variable. Rather, we know that an unknown alias value can never point to a local stack variable, and hence, we ignore such aliases in our optimizations.

We consider using an address pointed to by a pointer only when we can determine a unique target alias for it. Otherwise, we simply cannot optimize for it because we will have to use an load the AR with a value that cannot be known at compile time, and is only known at runtime. In Figure 8, P may point to either A or B. We cannot save any LDARs here because we have to load the AR with the address of either A or B at runtime, and speculating that P will point to A only or B only does not help to save any LDARs. We still have to use at least one LDAR to cover for the memory access in BB 4.

3.2 Variable Renaming, Webs and Variable Separation

In order to separate memory references, which can be independently considered for allocation, we rename variables and construct webs (as in Figure 3(c) and Figure 3(d)). A web [Muchnick 1997] or live range is defined as the maximal union of du-chains. Each web builds a separate variable after renaming, i.e. one must bind all the definitions and uses within a web to a single memory location. In this manner, we are able to achieve effective value separation at different program points. Value separation is extremely important as the compiler normally generates lots of temporaries that are reused repeatedly. Decoupling these variables that are disjoint in terms of values through re-naming gives us more freedom to coalesce them in a proper way to maximize the profit of offset assignment optimizations.

Our results show that over 80% local variables in the backend that can make use of the auto-modify instructions are recycled temporaries and the data segment size for them can increase after web identification. However, coalescing phase which follows greatly reduces the data segment size and brings about an overall size reduction when compared to the original data segment size.

To avoid interfering with a good register allocator and other optimizations before register allocation, our optimizing pass comes after register allocation, when all virtual registers that will be on the stack are identified. Also, for user-defined variables and temporaries, webs are built to achieve value separation.

3.3 Interference Graph and Coalescence Graph

After values separation, our coalescence algorithm needs to determine which variables are coalesceable.

An interference graph (IG) is built to represent the overlapping of the live ranges between different variables. The IG is defined as a graph where each node is a live range and an edge between a pair of nodes means that at a certain program point, the two nodes are simultaneously live, so they cannot be coalesced. It is perhaps most-used in register allocation.

A coalescence graph (CG) is a graph in which two nodes can be coalesced if and only if there is an edge between them. The CG is simply the complementary graph of the IG, which means, any two nodes connected by an edge on the IG will not be connected by an edge on the CG, and same vice-versa. In actual implementation, we only use the IG.

In our 10 benchmark programs, the IGs after value separation are sparse. Intra-procedurally, the average degree for each node is 8.17 on the IG and 210 for the CG. The strong connectivity on the CG means live ranges have plenty of chances to be coalesced with one another. The high average degree on the CG and the low average degree for the IG are probably due to the large amount of temporaries generated by the compiler. These temporaries are initially generated as virtual registers and then spilled. Most of the temporaries are defined once and used only a few times within the same basic block.

3.4 Profitability of Variable Coalescence

The high connectivity of nodes on CG grants us ample freedom to make good coalescing decisions to simplify the access graph (AG) considerably. Simplifying access sequence through judicious choice of coalescing is a non-trivial problem. Coalescence must be performed so that the resulting MWPC solution is improved. A key observation is that increasing edge weights through coalescence does not always lead to a better MWPC solution. In other words, coalescence may worsen the solution for offset assignment if not properly conducted. Coalescence seems to impact graph topology more than the edge weights as far as MWPC is concerned. This is due to the fact that in a final MWPC solution, there can be at most two incident edges on each node and thus, attempting to increase edge weights does not seem to impact the MWPC as much as reduction in node degrees which is a function of graph topology more than edge weights.

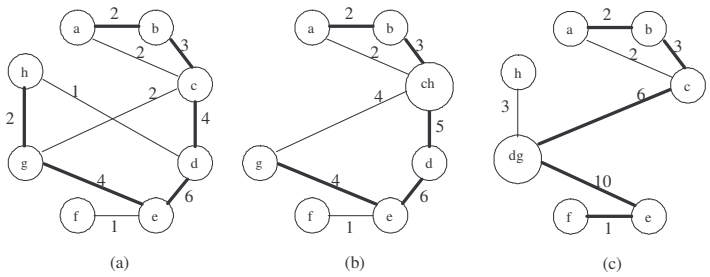


Figure 9. Profitability of Variable Coalescence

Figure 9(a) shows the original access graph and the current status of MWPC, i.e., a-b-c-d-e-g-h and f with total weight 21. If the coalescence graph permits the coalescing of nodes c and h, we can coalesce the two nodes and get an MWPC (a-b-ch-d-e-g and f) in Figure 9(b), the weight is 20. After coalescence, the MWPC is worse. The reason is because node c already has 4 neighbors. Adding more neighbors from h is not going to be

profitable. In contrast, in Figure 9(c), we coalesce node d and g. The MWPC is a-b-c-dg-e-f and h with a total weight of 22. This example shows that coalescence cannot be done arbitrarily without consideration of the topology of the IG and the AG.

3.5 Problem Formulation

The objective of offset assignment based on variable coalescence is to find both the coalescence scheme and the MWPC on the coalesced graph. We start with a few definitions and lemmas for variable coalescence.

3.5.1 Definitions

Coalesced Node (C-Node): A C-node is a set of live ranges (webs) in the AG or the IG that are coalesced. Nodes within the same C-node cannot interfere with each other on the IG. Before any coalescing is done, each live range is a C-node by itself.

Coalesced Edge (C-Edge): The C-edge is an edge set defined for a pair of C-nodes. A C-edge <c1,c2> between two C-nodes c_1 and c_2 on graph G is a set defined as:

$$\{<n_1,n_2> \mid n_1 \in c_1,\ n_2 \in c_2,\ <n_1,n_2> \text{ is an edge on } G\}$$

C-edges apply to either the AG or the IG. A C-edge exists only when this set is not empty.

C-AG (Coalesced Access Graph): The C-AG is the access graph after node coalescence, which is composed of all C-nodes and C-edges.

C-IG (Coalesced Interference Graph): The C-IG is the interference graph after node coalescence, which is composed of all C-nodes and C-edges. A C-edge between two C-nodes means the two C-nodes has interfering live ranges, therefore cannot be coalesced.

Coalesced Path Cover (C-PC): On a C-AG, a C-PC consists of a sequence of C-nodes c_1, c_2,…c_k, where $<c_i,c_{i+1}>$ is a C-edge between C-node c_i and c_{i+1}. The C-PC covers all

C-nodes exactly once, contains no cycles, and no C-node has a degree larger than two in the C-PC.

Weight of a C-Edge: The weight of a C-edge is the sum of all edge weights in the C-edge. C-edges with weight zero are C-edges that do not exist.

Weight of a C-Node: The weight of a C-node is the sum of all edge weights between any two nodes contained in this C-node.

Weight of a C-PC: The weight of a C-PC is the sum of weights of all the C-nodes and C-edges along the path.

C-MWPC (Coalesced Maximum Weight Path Cover): The C-MWPC is the C-PC with the maximum weight for all possible C-PCs on the C-AG. This maximum weight does not necessarily produce a unique path cover.

The algorithm starts with the original, uncoalesced AG, where each node is labeled as a C-node and by using the IG, the algorithm updates the C-nodes in both graphs through coalescing leading to the C-AG and the C-IG which keeps on changing dynamically as we coalesce more and more C-nodes. We first show that finding the best MWPC for a coalesced graph (called C-MWPC) is a hard problem. Next we attempt two heuristic solutions.

LEMMA 1: *The C-MWPC problem is NP-complete.*

Proof: C-MWPC can be easily reduced to the MWPC problem assuming a coalescence graph without any edge or a fully connected interference graph. Therefore, each C-node is an un-coalesced live range after value separation and C-PC is equivalent to PC. A fully

connected interference graph is possible, when all live ranges interfere with each other. Thus, the C-MWPC problem is NP-complete because the MWPC problem is NP-complete [Liao 1995; 1996]. □

LEMMA 2: *The solution to the C-MWPC problem is no worse than the solution to MWPC.*

Proof: Any solution to the MWPC is also a solution to the C-MWPC. But some solutions to the C-MWPC may not apply to the MWPC (if any coalescing were made). □

3.6 Coalescence-based Offset Assignment for Single-AR

Since the C-MWPC problem is NP-complete, heuristic algorithms must be applied to seek solutions in a reasonable amount of time. As mentioned in Chapter 2.1, two types of heuristics can be introduced to achieve different objectives: either to reduce the cost on the access graph (using OpCost) or to get a smaller memory footprint (using OpSize).

3.6.1 OpCost, a Heuristic Algorithm to Minimize Cost

Our first heuristic algorithm, OpCost, is separated into 2 parts. First, a set of pre-iteration coalescence rules are applied to capture cases that are definitely profitable. Then, in an iterative loop, coalescing is done incrementally. In each iteration, two C-nodes are selected for coalescing and the base SOA solver (we use Liao's SOA algorithm [Liao 1995; 1996] with the tie-break rule [Leupers 1996]) is run repeatedly, until no more coalescing is possible. Finally, the minimal SOA cost is returned together with a node to C-node mapping and the memory layout assignment. We call this base SOA solver "BaseSOA" for short.

Pre-Iteration Coalescence Rules

The pre-iteration rules are applied before we do iterative coalescing. Applying these rules will not worsen the SOA cost in all cases. All these rules are with respect to the access graph (AG). Note that we can coalesce a pair of C-nodes only if the C-nodes do not have an interference edge between them.

RULE 1: Coalesce all degree-0 C-nodes with any other C-node. Doing so will not affect the SOA cost.

RULE 2: Coalesce all degree-1 C-nodes with its neighbor. If its C-edge is already on the C-PC, the SOA cost is not affected, otherwise we reduce the SOA cost by the weight of this C-edge.

RULE 3: Coalesce all degree-2 C-nodes with the neighbor having a higher weight C-edge connected to it.

Rule 3 is explained in Figure 10. For C-nodes A, P, and Q, suppose the C-edge <A,P> is heavier than the C-edge <A,Q>. According to Rule 3, we should coalesce A with P. Assume there is a C-PC solution without coalescing A with P. Figure 10(a) to Figure 10(d) show 4 cases of that C-PC for C-edge <A,P> and <A,Q>. In Figure 10(a), none of the 2 C-edges is a part of C-PC, so the coalescence will reduce the cost of the SOA solution by Weight(<A,P>). In Figure 10(b), <A,P> is already on the C-PC and the cost remains unchanged. Similarly, when only <A,Q> is on the C-PC (Figure 6.c), we improve the SOA solution by Weight(<A,P>). And, if both of them are on the C-PC (Figure 10(d)), the cost is unchanged. Therefore, in each case, coalescing A with P can only improve (or cause no change to) the total weight of the C-PC before A and P are coalesced but will never worsen the solution.

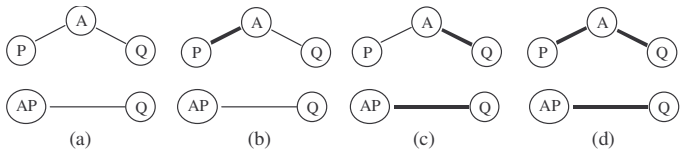


Figure 10. Profitability of Rule 3 Coalescence

Saving Due To Coalescence

After applying pre-iteration rules, we start to iterate. In each step of the iteration, we pick two C-nodes with maximum calculated saving and coalesce them. The basic idea is to use the current C-PC offset assignment to estimate savings if the 2 C-nodes were coalesced. For example, Figure 11(a) shows a C-AG with 8 nodes. The thick line is the current C-PC of the C-AG. If we coalesce d with g, C-edge <h,d> will now be on the C-PC, and C-edges <c,d> and <d,e> will be eliminated. C-edge <g,d> is also saved after d is merged with g. So, the total saving is W(h,d)+W(g,d)-W(d,e)-W(d,c) = 1, where W(<i,j>) is the weight of a C-edge <i,j>. In other words, the SOA cost is reduced by 1 if we coalesce d with g. In Figure 11, we illustrate 3 different cases to coalesce J with I. Figure 11(a) is a general case.

We save:

- The weight of the C-edge between I and J.
- The weight of all C-edges from I's neighbors (on the path cover) to J, i.e. C-edges <C,J> and <P,J> if they exist.

We lose:

- The weight of all C-edges from J's neighbors (on the C-PC) to J, i.e. C-edges <D,J> and <Q,J> if they exist.

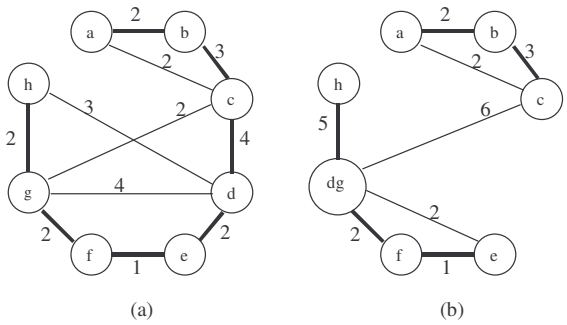


Figure 11. Cases to Calculate the Savings

Figure 12(b) is a special case where if I and J are already neighbors on the C-PC, then the weights of both C-edges <I,Q> and <J,P> are saved. In Figure 12(c), I and J have a common neighbor C. Then, the weight of the C-edge <C,J> is not a loss.

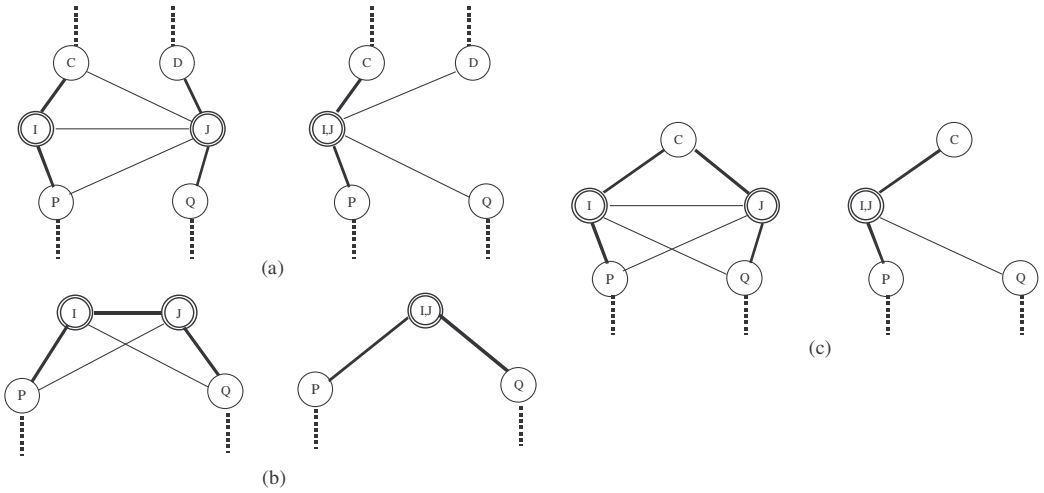


Figure 12. Coalescence Cases Based on Previous C-PC

Tie-Break for the Same Savings

If two or more pairs of C-nodes have the same coalescence savings, we apply a tie-break rule. This tie-break rule is similar to the one used in [Leupers 1996] to select

edges with the same weight during the construction of path covers. In our case, for each coalescence candidate $\{c_1, c_2\}$, the tie-break weight T is calculated as:

$T = \Sigma$ *weight (all C-edges joined to* c_1 *and/or* c_2*)*

A smaller T has higher priority, as explained in [Leupers 1996]. T reflects the graph density, and we want a smaller graph density because that would more likely bring about a better MWPC solution. C-edge $<c_1,c_2>$ (if it exists) is only counted once. In our benchmarks, this rule breaks all ties and improves the results slightly.

The Coalescence Algorithm

The whole coalescence algorithm is shown in Figure 13. Coalesce_OA_Single_AR takes a C-AG and a C-IG as input (here, the original AG and IG are passed to this function), and returns the minimal SOA cost and a node to C-node mapping. From the node mapping, we can easily generate the final C-AG, C-IG and C-PC solution.

Coalesce_OA_Single_AR contains two while loops. The first while loop tries to coalesce C-node pairs that are neighbors on the C-AG, until the largest calculated saving is zero, or when no more C-nodes pairs can be coalesced. The second while loop then exploits all remaining coalesceable C-node pairs, until no coalesceable C-node pairs can be found. Our coalescence framework works aggressively to reduce the number of C-nodes. Function Soa_Cost runs BaseSOA to find the SOA cost for the current C-AG. Notice that, the second loop coalesces even when the calculated saving is not positive. This is because our savings calculation is only a heuristic formula. After re-running the SOA solver, we may get a different C-PC, which may have an even lower SOA cost.

```
Input: C-AG, C-IG
Output:
  a. The minimal soa cost.
  b. A node map from original node to its C-node.

1.  Coalesce_OA_Single_AR(C-AG, C-IG) {
2.    Apply_Pre_Iteration_Rules();
3.    min_soa_cost = Soa_Cost (C-AG);
4.    min_node_map = a one to one map

5.    do{
6.      find two C-nodes satify: a.Do not interfere
                                 b.Connected on C-AG
                                 c.With max_saving
7.      if(max_saving>0){
8.        coalesce C-nodes, update C-AG,C-IG
9.        if(Soa_Cost(C-AG)< min_soa_cost)
            record as min_soa_cost, min_node_map.
10.     }
11.   } while(max_saving>0)

12.   while(there are C-nodes we can coalesce){
13.     find two C-nodes satisfy: a.Do not interfere
                                  b.With max_saving
14.     coalesce C-nodes, update C-AG,C-IG,
15.     if(Soa_Cost(C-AG)< min_soa_cost)
16.       record as min_soa_cost, min_node_map.
17.   }
18.   return min_soa_cost, min_node_map;
19. }
```

Figure 13. Coalescence-based Offset Assignment for Single-AR

The reason we have two separate while loops is that usually, a lower node degree density gives a lower SOA cost; thus, coalescing neighboring C-node pairs will less likely increase the node degree density. In this manner, we try to drive coalescence via a limited graph topology property i.e. the node degree; more complicated solutions are possible but may not yield much benefit due to the complexity of the problem.

SOA Cost Fluctuation During Algorithm Execution

To illustrate how SOA cost fluctuates during the two while loops, we show the SOA cost vs. iteration steps in Figure 14. Data in the figure are collected from one of the procedures called “findcost” in benchmark Twolf. In our experience, the SOA cost progression is very random and fluctuates greatly. This figure only gives its trend roughly.

It takes 90 coalescences for procedure ‘findcost’ to finish the two while loops. The thick vertical line at iteration 31 marks the end of the first whole loop and the start of the second while loop. ‘findcost’ has a starting SOA cost of 144, and a minimum SOA cost of 115 achieved at iteration 44. Therefore, the minimum SOA cost is achieved during the early part of the second while loop, which is commonly observed in most procedures.

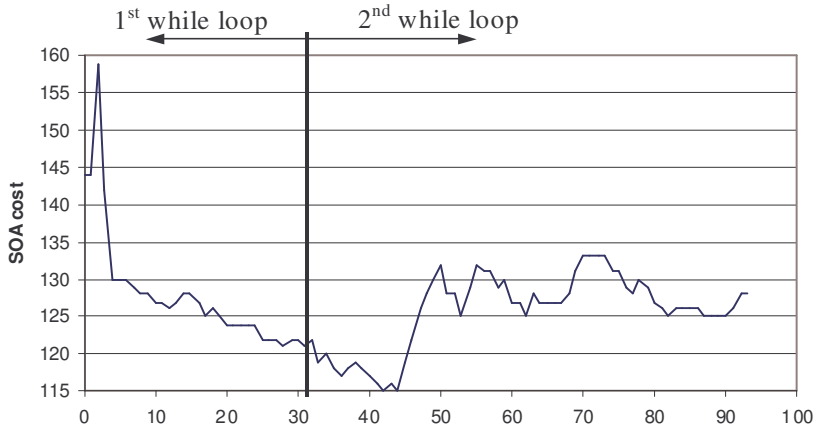


Figure 14. SOA Cost Fluctuation Along with Iterations for Twolf Procedure ‘findcost’

Note that the final SOA cost achieved might not be the lowest SOA cost ever achieved by the algorithm. If we want to place emphasis on optimizing for code size rather than stack size, we can remember the information for the case when the lowest SOA cost was ever achieved, and then revert to that solution at the end of all coalescing.

3.6.2 OpSize, a Heuristic Algorithm to Minimize Size

The second heuristic algorithm attempts to minimize the number of C-nodes so the program will have a small memory footprint at runtime. The heuristic consists of two distinct phases. The first phase is minimal coloring of the IG. Nodes with the same color

are coalesced on both the AG and the IG. The following lemma says minimal coloring of the IG is equivalent to achieving minimal number of C-nodes after coalescence.

LEMMA 3: *The minimal number of C-nodes after node coalescence is equal to the minimal number of colors required to color the IG. Furthermore, a coloring scheme of the IG is equivalent to a legal C-node formation.*

Proof: A coloring scheme of the IG can be directly applied to a C-node formation by assigning nodes with the same color in the IG to the same C-node. The number of C-nodes is the number of colors for the IG. Similarly, a C-node formation can be directed to a coloring scheme by coloring the nodes in the same C-node with the same color and nodes in different C-nodes with different colors. Since nodes in the same C-node do not interfere with each other, i.e. no edge exists between them on the IG. Therefore, the two problems are equivalent and minimal coloring is the same as minimal number of C-nodes we can get. □

We use a simple coloring algorithm similar to the one used for the Chaitin style register allocation [Chaitin 1981; Chaitin 1982]. When removing nodes from the IG and pushing them onto the coloring stack, we always remove the one with lowest degree first. Since coloring is performed on the IG, nodes with the same color are guaranteed to be coalesceable. After the coloring phases, an SOA solver (no coalescence) is applied on the resulting C-AG and C-IG to assign offsets for coalesced nodes.

Aggressive coalescing might possibly lead to higher SOA costs. However, our experiments show the OpSize heuristic still performs better than the baseline SOA solver

without variable coalescence. Compared with OpCost, OpSize is less effective in lowering the SOA cost but achieves greater stack size reduction.

3.7 Coalescence-based Offset Assignment for Multiple-AR

The Multiple-AR model allows more than one AR to utilize the auto-modify mode. With the trend in embedded processor design to increase the number of ARs, multiple-AR model is playing a more and more important role in optimizing compilers to generate efficient code. In Motorola DSP56300, the AR is the general purpose register, and one of the 8 ARs is used as stack pointer. The other 7 ARs can be allocated for other purposes to hold variables. If one could solve the problem of address register assignment with fewer registers, the remaining address registers can be used for other purposes.

Generally, previous work on offset assignment for Multiple-AR (or GOA) [Liao 1995; Leupers 1996] all attempts to separate variables into several group, so that each group can be served with one AR. Here, we define *AR Group* as a group of variables that are allocated to one AR. With variable coalescence, our algorithm not only needs to partition variables into AR Groups, but also should coalesce them properly.

As Single-AR, Multiple-AR can be optimized towards two objectives. Both OpCost and OpSize require a heuristic algorithm to coalesce and partition variables into AR Groups, however as shown in 4.b, OpSize has an additional phase to minimally color the IG. We will discuss these phases in the following sections.

3.7.1 Coalescence Algorithm for Multiple-AR

Figure 15 shows the algorithm called Coalesce_OA_Multiple_AR. This algorithm is invoked by both OpCost and OpSize. The only difference is, for OpSize, a graph

coloring algorithm first coalesces nodes on the graphs aggressively, and then Coalesce_OA_Multiple_AR works afterwards if an optimal solution cannot be obtained immediately.

```
Input: AG, IG, K—number of ARs
Output:
  a. The minimal GOA cost.
  b. A mapping from node to its C-node.
  c. A mapping from C-node to AR number.

V: node set, contains all nodes initially
G1,G2,…Gk: AR Groups, i.e. a set of C-nodes

1.  Coalesce_OA_Multiple_AR(AG, IG, K) {
2.    G1=G2=..=Gk=Φ;

3.    //add each node to an AR Group
4.    while(V is not empty){
5.      mini_set=Φ; min_cost=MAX_INT;

6.      //build mini_set
7.      foreach node v in V{
8.        cost=minimal add-on cost to put in one of
9.             the Gi by running Coalese_OA_Single_AR on Gi.
10.       if(cost == min_cost){
11.         add (v,i) to mini_set;
12.       }else if(cost<min_cost){
13.          mini_set={(v,i)}; min_cost=cost;
14.       }
15.     }

16.     //tiebreak
17.     foreach pair (v,i) in mini_set{
18.       w1(v)=sum(weight<u,v> on AG) u ℳG1⋓G2..⋓Gk-Gi
19.       w2(v)=number of v's neighbors on the IG
20.     }
21.     keep only pairs with maximal w1 in mini_set(tie break on w1)
22.     if(|mini_set|>1)
23.       keep only pairs with smallest w2 in mini_set(tie break on w2)
24.     if(|mini_set|>1)
25.       still have tie, pick one randomly.
26.
27.     for selected pair(v,i) add v to Gi
28.     remove v from AG and IG
29.   }
30.   run Coalese_OA_Single_AR on all Gi
31.   return 1)the GOA cost as the sum of all SOA costs
32.          2)mapping from node→C-node, C-node→AR number
33. }
```

Figure 15. Coalescence Algorithm for Multiple-AR

Initially, the algorithms stores all nodes in set V and all AR Groups G_1,G_2…G_k are empty. In the while loop from line 4 to line 29, during each iteration, one node in V is assigned to an AR Group. The while loop has two main parts. The first part builds up the mini_set. It attempts to put each node to each AR Group and calculate the extra cost that

will be incurred by calling Coalesce_OA_Single_AR (Figure 13) on that AR Group. We should find a (v, i) pair so that assigning node v to AR Group Gi incurs minimal add-on cost, however it may happen that several pairs have the same minimal add-on cost. If so, there will be multiple entries in mini_set and the second part picks one entry through a 3-step tie-break scheme.

The tie-break scheme we use shares some features with the tie-break GOA algorithm in [Leupers 1996]. We calculate two values for tie-break. Value w_1 is calculated for each entry in mini_set. If v is selected for Gi, we sum all the edges on the AG from v to a node that is in $G_1 \cap G_2 .. \cap G_k - G_i$. Since, the edge from v to any node in AR Groups other than G_i are eliminated as we illustrated in the motivation example, we prefer a larger w_1. If this still cannot break all ties, we try another value w_2. w_2 is calculated for each node v as the number of neighbors that are still on the IG. Larger w_2 means more interference with the nodes that have not been added to one of the ARs. We prefer a smaller w_2, which means more nodes on the IG later can be coalesced with v. If both tie-breaks fail, we just randomly pick one from the remaining entries in mini_set. Our experiments show this rarely happens. Finally, the algorithm calls Coalesce_OA_Single_AR (Figure 13) for each AR Group. It returns a node to C-node mapping and a C-node to AR Group number mapping.

3.7.2 OpSize Algorithm for Multiple-AR

Since aggressive variable coalescence can greatly reduce the number of C-nodes on the graph, with multiple ARs, in many cases, we can actually get the optimal solution. The following lemmas specify when the optimal solution can be achieved.

LEMMA 4: *If there are only two C-nodes on the C-AG, then the SOA cost is optimal.*

Proof: Since there is only one C-edge on the C-AG, so this C-edge must be on the C-MWPC. Hence, the SOA cost is 0. □

LEMMA 5: *If there are K address registers available for use and the number of C-nodes is no more than 2K, we can get the optimal solution, i.e. GOA cost=0 by assigning no more than two C-nodes to each address register.*

Proof: Following the Lemma, the SOA problem for each address register is optimal—zero SOA cost. The GOA cost is equal to the sum of the SOA cost for all address registers, so the GOA cost is also 0. Therefore, the solution is optimal. □

As we know, the IG constrains the nodes from being coalesced (AG affects the cost but can be disregarded when minimizing the C-node number). From Lemma 3 and Lemma 5, we have the following corollary.

COROLLARY 1: *If we can color an IG with 2K colors, then there is an optimal solution, i.e. GOA cost=0 with K address registers.*

Notice that, Corollary 1 is only a sufficient condition. Even when the color number is greater than 2K, we may still get an optimal solution by first aggressively coalescing the nodes followed by the coalescence algorithm (the Coalesce_OA_Multiple_AR algorithm in Figure 15) on the resulting C-AG and C-IG.

Like Single-AR, we use a simple coloring algorithm similar to the one used for Chaitin style register allocation.

To quantify the number of times we can get optimal solutions with certain number of address registers, we did experiments on 10 benchmark programs. All data pertains to local variables. We count the number of procedures that can be optimally solved in cases of 1) after IG coloring, and 2) after both coloring and Coalesce_OA_Multiple_AR. This count gives the final number of optimal solutions. As mentioned earlier, Corollary 1 only gives a sufficient condition, i.e. even if an AG has more than two nodes, its SOA cost can still be zero, or the GOA cost can still be zero if the IG is not 2K-colorable. So, the final number of optimal solutions could be larger than the one obtained from IG coloring.

Table 1. Percentage of Optimal Solutions for Multiple-AR

#AR	Epic	Gsm	G721	Mpeg2d	Mpeg2e	Bzip2	Gzip	Mcf	Twolf	Vpr	Average
2 (color)	84.9	85.56	76.92	82.68	63	52.38	85.15	80	62.94	65.83	73.94
2 (final)	86.8	90	96.15	90.55	77.23	87.18	90.1	93.33	79.19	82.01	87.25
3 (color)	90.57	93.33	96.15	91.34	81	87.2	90.1	93.34	76.1	85.25	88.44
3 (final)	94.34	97.78	100	94.49	88.12	92.31	96.04	100	89.85	94.24	94.72

Table 1 shows the percentage of optimal solutions for different number of address registers. Rows 2 and 4 are the percentage of optimal solutions given by the number of colors. For instance, for Epic, with 2 ARs, 84.9% procedures can generate optimal solutions after coloring. In other words, 84.9% procedures' IG can be colored by 4 colors. But with 3 ARs, 90.57% of the procedures are 6-colorable. Row 3 and 5 are the final number of optimal solutions. The percentage of optimal procedures is increased.

On average, 87.25% of the procedures can finally get optimal solutions with 2 ARs, while 94.72% procedures can finally get optimal solutions with 3 ARs. This means our solution is very close to the optimum.

4. POST-PRE OPTIMIZATION

Post-pre optimization determines whether post- or pre-modify mode should be used for each memory access instruction (in this chapter, we implicitly restrict "memory access instructions" to those accessing the variables on the access graph) so as to minimize the number of LDARs. This optimization comes after offsets are assigned to variables. According to the offsets, we find out the offset difference between adjacent memory accesses. Given that attempting all possibilities of post-pre mode and AR modification insertion can make the problem intractable, our algorithm greatly reduces the complexity via two techniques. Firstly, we split basic blocks at certain points without losing the optimality of the problem. Basic Block Splitting leads to smaller optimization units that can be independently optimized, therefore the problem complexity is significantly lowered. Secondly, we undertake a branch and bound algorithm to narrow down the search space.

4.1 Offset Distance

For each memory access instruction, we can mark the offset of each variable being accessed. "Offset Distance" is the offset difference between two adjacent memory access instructions. In Figure 16, we show the variable offsets, selected code segment with only memory accesses, offsets and offset distances. It is easy to observe, if the offset distance is 1, either the first memory access instruction can post-modify the AR or the second memory access instruction can pre-modify the AR before its memory access.

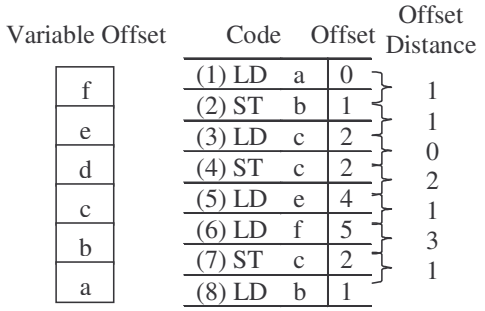


Figure 16. Example for Offset Distance

The addressing mode decision of one memory access instruction can affect its neighbors in certain circumstances. For example, if the 1st instruction LD a in Figure 16 does not perform post-modify, i.e. post-increment, the 2nd instruction ST b must do pre-increment to avoid an extra LDAR. However, sometimes the decision on one memory access instruction does not depend on its neighbor(s). For instance, the 3rd and 4th instructions access the same variable c, therefore no post-modify is needed for the 3rd instruction and no pre-modify is needed for the 4th instruction. On the other hand, the 3rd instruction might use pre-modify depending on the other neighbor, but this is independent of the addressing mode of the 4th instruction. Similarly, the 4th instruction might use post-modify, but it is irrelevant to the addressing mode of the 3rd instruction. As another example, the offset distance between the 6th instruction and the 7th instruction is 3, which means an LDAR is not avoidable to modify the AR between these two instructions. After the LDAR is inserted, the addressing mode of instruction 6 becomes independent of that of instruction 7 due to the same reason as for instructions 3 and 4. In short, we can summarize the addressing mode relationship between two neighboring instructions as in Figure 17. Up till now, we have only considered addressing modes for instructions inside

one basic block. It becomes more complicated to establish the relationship of addressing modes at the boundary of basic blocks, like the example in Figure 5, when one basic block has multiple predecessors and successors, we will discuss such constraints later.

Offset Distance	1st Instr.	2nd Instr
0	no+	no
1	Post	no
	no	Pre
2	Post	Pre
>2*	no	no

+This means neither post nor pre mode is required.
*An AR modification instruction is required.

Figure 17. Addressing Modes between Two Adjacent Memory Access Instructions

4.2 Basic Block Splitting and Canonical Form

Following the identification of offset distance, in this section, we will talk about how to split basic blocks and transform the CFG to Canonical Form as defined below.

4.2.1 Definition of Canonical Form and Canonical CFG

If a CFG has offset distance equal to 0, 1 or 2 inside all the basic blocks, it is in canonical form. The CFG is called Canonical CFG.

Canonical form facilitates the formulation of post-pre optimization. Based on the table in Figure 17, we can easily transform a CFG to its canonical form through basic block splitting. Each part of the canonical CFG after basic block splitting is called a "sub-CFG". Furthermore, basic block splitting can greatly reduce the problem complexity because we consider each sub-CFG as a single unit of optimization. However, we must guarantee that basic block splitting transforms the CFG without affecting the optimal

solution for post-pre optimization. Our basic block splitting technique involves two steps; the following lemma says we can split between two memory accesses with offset distance 0 or greater than 2. After this step, all basic blocks only have offset distance 1 or 2. In the second step, we will further get rid of offset distance 2.

LEMMA 6: *Inside one basic block, if two consecutive memory access instructions have offset distance >2 (in this case, one AR modification is unavoidable), the basic block can be split between these two instructions. After splitting, the split point becomes the boundary of the two new basic blocks. Such splitting does not affect the optimal solution to the post-pre optimization. When the two consecutive memory access instructions are within the same basic block and the offset distance is 0, then we can split as well.*

Proof: Notice that, after splitting, the 1st instruction becomes the last memory access instruction in its basic block and the basic block has no successor, therefore no post-modify is necessary. Likewise, the 2nd instruction becomes the first memory access instruction in that basic block and the basic block has no predecessor, therefore no pre-modify is needed for it. In the first case, i.e. the offset distance is greater than 2, one and only one LDAR must be inserted in the optimal solution, and no post-modify is needed for the 1st instruction, since the LDAR is sufficient to set the AR to point to the next offset. Also, no pre-modify is necessary for the 2nd instruction. This is also enforced on the CFG after splitting. In the second case, assume the offset distance is 0 between the two instructions, the optimal solution should not require post-modify for the 1st instruction, nor should the pre-modify for the 2nd instruction be needed. But the restraining condition is that these two memory access instructions must be on the same

basic block. This is because when we have a CFG split or join, the AR offset after execution of the first instruction might be needed for another instruction on a different CFG path. By restraining the two instructions to the same basic block, such a case will not occur, and the 1st instruction will always be followed by the 2nd instruction in execution order. Thus, after CFG splitting, the optimal solution for the new CFG should be one less than the original optimal. □

Consider an example given in Figure 18 which illustrates two different CFGs, each having an instance of offset 0. Figure 18(a) is called a "CFG join" because the control flow for two basic blocks go into the same basic block, thus "joining" paths. On the contrary, Figure 18(b) is called a "CFG split". In Figure 18(a), instruction 1 goes to instruction 3 with offset 0. Hence, instruction 1 does not need an LDAR to point to instruction 3. However, we cannot split the CFG between these two points because instruction 3 may be reached from instruction 2, and if we use a pre-increment mode for instruction 3 and no auto-modification for instruction 1, we produce an incorrect solution. The same principle holds for Figure 18(b). We cannot split the CFG because we have to consider the basic block boundaries carefully.

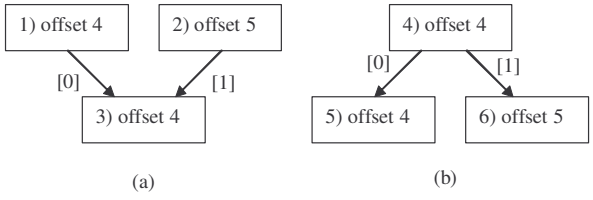


Figure 18. Requirement for Offset Distance to Split only within a Basic Block

Based on Lemma 6, after step 1, all basic blocks only have offset distances 0, 1 or 2. In the second step, we simply split basic blocks between two memory accesses with

offset distance 2. In contrast to the first step, after splitting at offset distance 2, the two new basic blocks become predecessor and successor. Figure 19 shows an example with three basic blocks. After step 1, in Figure 19(b), BB1 and BB3 are split, and after step 2--Figure 19(c), BB2 is split into two basic blocks at the point with offset distance 2. The two new basic blocks in Figure 19(c) are still connected and become predecessor and successor. Notice that the offset distance between two basic blocks are not considered during basic block splitting, but will be considered when we start to solve the canonical CFG.

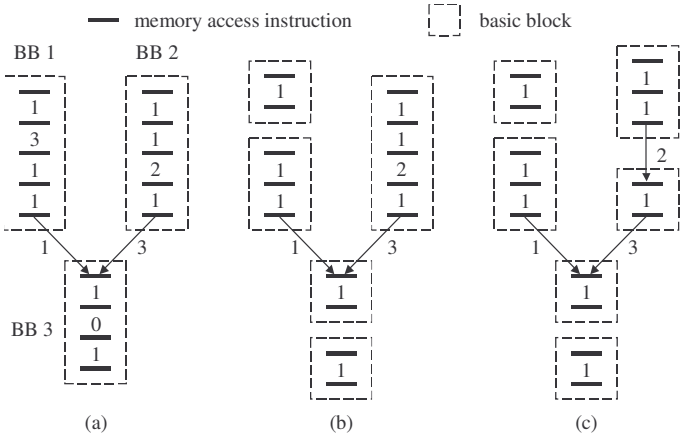


Figure 19. Example for Canonical Form Transformation (a) Original Code (b) After Step 1 (c) After Step 2

After splitting, the resulting CFG is likely to be disconnected and contain many small, connected components that can be separately optimized, reducing the problem complexity. We can split basic blocks, then solve it optimally. To get the solution for the original CFG, basic blocks are reconnected at the split points and one LDAR is added at each point with offset distance greater than 2. For the example in Figure 19(c), the CFG is in canonical form and it now splits into 3 connected components. After solving the

canonical CFG optimally, we need to add the cost by one LDAR, since the splitting in BB1 is at offset distance 3, therefore we should make up for that LDAR.

4.2.2 Solving the Canonical CFG with Branch And Bound

To find an optimal solution to the canonical CFG, we take a branch and bound algorithm, which prunes the solution space significantly and identifies the optimal within a short compilation time. This is done by considering each connected component of the canonical CFG as a standalone block. Recall that a component is determined by the fact that we might need LDARs at all its boundaries. Within the component itself, the process of determining which auto-modify mode to use, if needed, is very straightforward and is only a matter of traversing down the basic block. Therefore, our concern is to determine whether we are able to save any LDARs along any boundaries of the connected components.

For a connected component on the canonical CFG with M basic blocks, the search space is 2^{2M}, i.e. we can specify 2M 0-1 integer variables such that each variable indicates whether a particular AR modification instruction should be inserted. These variables are defined as follows.

B_i : Can be 0 or 1, indicates if an AR modification instruction should be inserted at the beginning of basic block i.

E_i : Can be 0 or 1, indicates if an AR modification instruction should be inserted at the end of basic block i.

The algorithm flow graph is shown in Figure 20. The search space SP is initialized to contain all of the 22M 2M-bit vectors. Every time, one element *spe* is selected from SP and checked if it gives a feasible solution. The details about how to check the feasibility will be discussed later. If *spe* is not a feasible solution, another element is picked from SP and checked. Otherwise, the feasible solution can be used to prune the solution space, i.e. all unchecked vectors with cost no less than *spe* can be removed from SP. Here the cost of a vector in SP is the number of bit 1's in the vector, because each bit 1 means an AR modification instruction is inserted at a particular location. Finally, the solution with minimal cost is output.

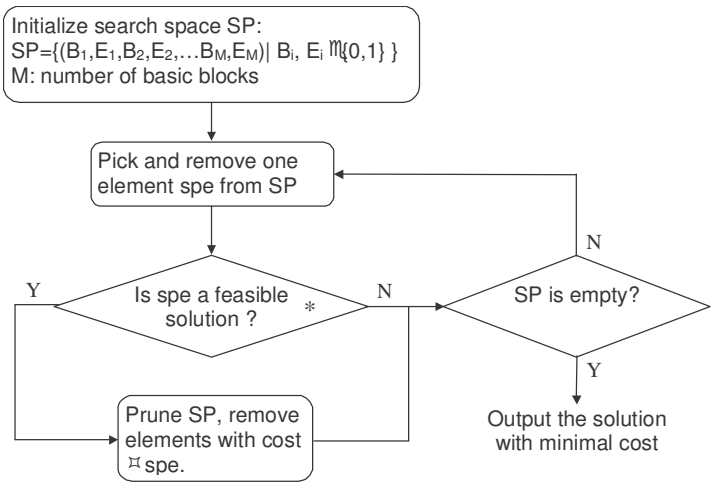


Figure 20. Flow Graph for Solving the Canonical CFG

4.2.3 Checking the Feasibility

To check the feasibility of a solution vector *spe*, we need to verify if all memory access instructions are satisfied, which means the AR should contain the required address value before reaching a memory access instruction. It either points to the variable being

accessed or can be pre-incremented or pre-decremented to point to that variable. First we give three definitions.

EO_i : An integer value, which is the ending offset of basic block i. This is the value in the AR when execution leaves the end of a basic block.

BVO_i : An integer value, which is the variable offset of the first memory access instruction in basic block i.

EVO_i : An integer value, which is the variable offset of the last memory access instruction in basic block i.

Notice that, a solution vector specifies all B_i and E_i ($i \in \{1,\ldots M\}$) values. Also, BVO_i and EVO_i ($i \in \{1,\ldots M\}$) are constants for a canonical CFG. The feasibility checking involves finding out if the EO values can be obtained with respect to the following restrictions.

RESTRICTION 1. If $B_i=0$, for basic block i's predecessors p_1, $p_2 \ldots p_k$, we have $EO_{p1}=EO_{p2}= EO_{pk}=BO_i$, where $BO_i \in \{ BVO_i-1, BVO_i , BVO_i+1\}$.

RESTRICTION 2. If $E_i=0$, $EO_i \in \{ EVO_i-1, EVO_i, EVO_i+1\}$.

RESTRICTION 3. If $B_i=E_i=0$, $|EO_i - EVO_i|+|BO_i-BVO_i|\leq 1$ (here BO_i is defined in Restriction 1 when $B_i=0$).

Restriction 1 is true, since all predecessors should come to basic block i with the same value in the AR if the AR is not changed at the beginning of basic block i, i.e. $B_i=0$.

Also, the position pointed to by the AR should be at most 1 slot away from the first memory access instruction's offset, i.e. BVO_i so pre-modify can handle it. In this case, we define BO_i as the value in AR. Restriction 2 is simply correct, when no AR modification is performed at the end of basic block i, the value in AR when leaving the basic block should be one of $\{EVO_i\text{-}1, EVO_i, EVO_i+1\}$. Finally, Restriction 3 says either the first memory access instruction does pre-modify or the last memory access instruction does post-modify or none of them, but not both. This restriction is illustrated in Figure 21(a). If the first memory access needs pre-modify, the last memory access must be inhibited from post-modify to avoid an extra AR modification instruction. Similarly, if the last memory access does post-modify, then the first one cannot use a pre-modify addressing mode.

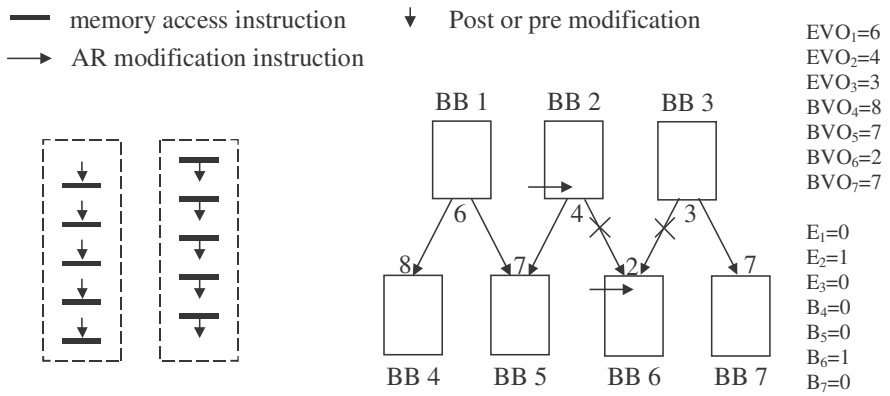


Figure 21. Illustrations for Feasibility Checking

As an example, Figure 21(b) shows one of the connected components on a canonical CFG with 7 basic blocks. We list all needed BVO and EVO values on the right (also marked on the CFG). We need to check, as specified by the *spe* vector, if the insertion scheme, i.e. to insert at the end of BB2 and the start of BB6 leads to a feasible solution. In our algorithm, we first group EO values that are equal based on Restriction 1.

Notice that we can build a transitive closure through the predecessor/successor relationship. In the example, by Restriction 1, $EO_1=BO_4$ and $EO_1=EO_2=BO_5$. Transitively, $EO_1=EO_2=BO_4=BO_5$. Meanwhile, $B_6=1$, therefore Restriction 1 cannot be applied to BO_6, the two edges coming into BB6 can be removed. In other words, the AR modification instruction at the beginning of BB6 blocks both EO_2 and EO_3. Upon this point, $\{EO_1, EO_2, BO_4, BO_5\}$ form a group and $\{EO_3, BO_7\}$ form another group, which means variables in the same group are equal. Next, we check the value range for each group. With Restriction 1, $7\leq BO_4\leq 9$, $6\leq BO_5\leq 8$. With Restriction 2, $5\leq EO_1\leq 7$. Thus, this group can take value 7, which is the intersection of the three ranges. Similarly, the second group has two value ranges, i.e. $6\leq BO_7\leq 8$, $2\leq EO_3\leq 4$. However, these two ranges have no overlapping. Eventually, our feasibility checking concludes that this insertion scheme is infeasible.

5. FURTHER OPTIMIZATIONS

5.1 Inter-Basic-Block Offset Assignment

We can consider the basic block as the basic unit of offset assignment because a basic block gives a static access sequence that does not depend on execution. In existing literature, offset assignment is usually considered only on the basic block level. Very little, if any, was mentioned about a realistic offset assignment for an entire control flow graph (CFG).

It is useful to consider the entire CFG when determining offset assignments and addressing modes. This is because the stack variables must have the same stack location throughout the entire CFG, among each of the different basic blocks. When we consider offset assignment only within basic blocks, we cannot capture the actual effect of offset assignment in the CFG. Also, the basic block boundaries denote a joining of access sequences, and they should be considered as well. The consequence of not considering offset assignments across basic blocks is that every basic block that contains a stack memory access will need at least one LDAR, for the first instruction that accesses the stack. So we need to consider offset assignment at the CFG level.

5.1.1 Algorithm for Inter-Basic-Block Offset Assignment

The input to the algorithm is the CFG, and we need to identify those instructions that access stack locations. Let N be the number of stack locations accessed in the CFG. Then we have N choose 2 (N * (N-1) / 2) pairs altogether. We need to know how many pairs there are because we need to allocate an array for storing the SOA cost of not

putting each pair of slots adjacent in memory. Figure 22 gives the pseudocode for this algorithm.

```
1   function InterBlockOffsetAssign (cfg)
2     N = number of stack locations in cfg;
3     if (N <= 2) return;
4     G = null graph;
5
6     Remove all basic blocks without any loads/stores to form
7       the opaque CFG;
8
9     do {
10      for (a, b) = each pair of stack locations
11        {count(a,b) = 0;}
12
13      for bb = each bb in opaque cfg {
14        F = first stack location accessed in bb;
15        if (F is not in G or F has less than 2 neighbors in G)
16          for pred = each predecessor of bb {
17            L = last stack location accessed in pred;
18            if (L is not in G or has less than 2 neighbors in G)
19              count(F,L) = count(F,L) + 1;
20          }
21
22        for (a, b) = each pair of consecutive stack locations in bb
23          if (both a and b are each not in G
24              or have less than 2 neighbors in G)
25            count(a,b) = count(a,b) + 1;
26      }// for each bb
27
28      (a, b) = pair with highest count(a,b);
29      if (count(a,b) == 0) break;
30
31      if (a is not in G) {add a to G; N--;}
32      if (b is not in G) {add b to G; N--;}
33      increment weight of edge (a, b) in G;
34    } while (N > 0);
35
36    form offset assignment from G;
37  end function
```

Figure 22. Algorithm for Inter-Basic-Block Offset Assignment

We first build the opaque CFG, which is the CFG with only basic blocks that that contain at least one stack access instruction. In offset assignment, there is nothing we need to do with basic blocks that do not have any stack accesses. We call such basic blocks “transparent basic blocks”. Then we enter an iteration until we are done. We traverse through the CFG once in each iteration. For each pair of stack locations, we count the number of consecutive accesses between them on the CFG. In basic block joins

or splits, we count all possible access sequences. For example, if variable A can be followed by any one of variables B, C and D, we count one for each of (A, B), (A, C) and (A, D). Then at the end of the iteration, we take the pair with the largest count, and increase the number of neighbors of the stack locations in the pair by one. Each stack location cannot have more than two neighbors. And the graph represented by the neighbors information cannot contain a cycle.

For each basic block, we only cycle through the predecessors but not the successors because they represent the same set of information. By cycling through either one of them, we can cover all CFG edges. If we cycle through both of them, we will double-count all stack accesses that are closest to each basic block boundary.

InterBlockOffsetAssign works on the same basic principle as Liao's original SOA algorithm. In that algorithm, Liao builds the access graph, then collects all the edges of that access graph and sorts them in descending order of weight. InterBlockOffsetAssign is different in that it does not collect all the edges and their weights in advance because that information changes dynamically during the algorithm execution. Both InterBlockOffsetAssign and Liao's SOA algorithm are greedy. More recent algorithms such as Incremental-Solve-SOA gives slightly better results than Liao's SOA algorithm. However, in practice, Liao's algorithm is fast and gives a solution close to the optimal solution, and Incremental-Solve-SOA requires more execution time while improving the results only slightly. Therefore, InterBlockOffsetAssign also gives a solution close to the optimal solution while requiring little execution time.

Our algorithm uses a graph G to store temporary data. In practice, we only need a special simplified graph, which consists of the number of neighbors of a particular node,

and the first and second neighbor of a node, if any exists. This is effectively a cycle-free graph that requires each node to have two or less neighbors. This is the same kind of graph data structure that we used for Liao's SOA algorithm. Towards the end of this work, we evaluate the performance of InterBlockOffsetAssign.

5.2 Offset Registers Optimization

5.2.1 Characteristics of Offset Registers

There are two different modes of using offset registers. The first mode is to apply an offset without changing the address register. The second mode is to access the address stored in the address register and then do a post-modify by the value in the offset register. In the Motorola DSP56300 processor, we can either use the first mode or the second mode, but not do both at the same time. In the first mode, if we have an instruction "MOVE (R0+N0), R1", we set the value of address register R1 to that value stored in the memory location R0+N0, but R0 does not change. In the second mode, if we have an instruction "MOVE (R0)+N0, R1", we set the value of register R1 to that value stored in the memory location R0, and then set R0 to R0+N0 after the instruction executes. The first mode uses a modified address, while the second mode modifies the address register. The DSP56300 processor does not permit both operations to be performed in the same instruction.

There are three main differences between an address register and an offset register:

1) We cannot address a memory location directly using an offset register. An offset register must always be used together with an address register within a load or store instruction.

2) An address register can usually be used as a general register but an offset register cannot be used for anything else except for specifying address offsets.

3) Auto-modify mode cannot be used to modify the value of offset registers, nor can it be used in the same instruction as an offset register.

Given the characteristics of offset registers, we concluded that the way to use them is to pre-load them with a certain known value, and then use that value together with address registers later. This becomes useful when in an access sequence, we are trying to access two stack locations placed over one word apart, but we cannot use the auto-modify modes to save an LDAR. If the offset register has a value that happens to coincide with this particular difference in offset, then we can use the offset register to save the LDAR. Therefore, our strategy is to pre-load one or more offset registers with certain fixed, known constant values, and use them throughout the CFG. We do not simply pre-load the offset register with any small arbitrary value. Rather, we select the few offset values which can be used the most number of times within the function, based on the results of offset assignment.

In the Motorola DSP56300 GCC, there are eight offset registers N0 to N7. Some offset registers were never used by the compiler because the compiler does not implement this optimization of using offset registers. This means that offset registers were designed with the goal of providing this class of optimizations by explicitly using them in a carefully-crafted manner.

5.2.2 Algorithm for Offset Registers Optimization

Figure 23 shows the pseudocode for the algorithm for using offset registers. This optimization assumes that the program is entirely self-contained, so that a global

optimization can be used. This is the same condition for the case when we are doing global register allocation. The reason for doing this in an inter-procedural basis is so that we can save on the caller and callee save instructions for saving and restoring the offset registers across function boundaries. In practice, caller and callee save instructions are more expensive than LDARs, and so we always use LDARs instead of caller and callee save instructions.

```
1   function UseOffsetRegisters ()
2     N <- number of offset registers available for use;
3     given the layout assignment, count the number of times each
4       offset was required in order to save an LDAR;
5     choose the first N most-used offsets, choose any if tie;
6     in the function prologue of main(), assign these values to
7       the N offset registers;
8     modify the code to use these offset values in place of an
9       LDAR whenever possible;
10  end function
```

Figure 23. Algorithm for Offset Registers Optimization

In order to produce the best results possible, we do offset register usage globally for an entire self-contained program. As in global register allocation, UseOffsetRegisters requires the intermediate code of all functions in the program to be available before it can be used. One should note that the offset may very well be negative when we are trying to go from a higher memory location to a lower one. The more offset registers we have, the more LDARs we will be able to save. Later in this work, we discuss on the issues concerning the implementation of such a scheme and how realistic it is in more recent DSP processors.

6. IMPLEMENTATION DETAILS

This chapter describes the nitty-gritty details of getting the offset assignment optimizations to work on the target architecture, and the problems encountered. The lessons presented here could be useful to anyone who desires to implement these optimizations on a real compiler.

6.1 Implementation Environment

Our environment is the Motorola DSP56300 processor toolset including a cycle-accurate simulator --sim56300, and a retargeted GNU C compiler [Stallman 2002], which comes with standard header and library files. Our optimization is implemented at the RTL level — GCC's IR, after the "reload pass" of GCC, and before the assembly is produced, so that we can capture all the temporaries and spill code generated by the compiler.

6.1.1 Register Set

DSP56300 has a word size of 24 bits. In the Data ALU, it has two 56-bit accumulator registers, A and B, and two 48-bit input registers, X and Y. In the Address ALU, it has eight 24-bit address registers, R0 to R7, eight 24-bit address offset registers, N0 to N7, and eight 24-bit address modifier registers M0 to M7. The address registers are also used as general-purpose registers. R6 is reserved as the stack pointer. N0 to N7 are used as code generator temporaries. M0 to M7 are unused. R2 is used as a temporary register to store the function address in a function call.

Among the eight address registers available on Motorola DSP56300, we can reserve up to four ARs for use in our optimizations. These registers are R3, R0, R4 and R5. The other registers may be used by the compiler even when they are marked as fixed registers in GCC. This is because of some irregular assumptions that they use in the code generator, which does not follow the standard semantics used by the GCC code generator.

6.2 Implementation Details for SOA

We begin by discussing SOA because it is the most fundamental and earliest optimization available for the auto-modify addressing modes. We have already seen that SOA is simply the problem of finding a stack layout for variables local to a function. The first question is: in which phase of the compiler should we perform this optimization? Due to the fact that SOA requires register allocation to be completed, and before assembly code to be generated, it has to be placed between register allocation and code generation. In GCC, this phase is call “reloading”. What does “reloading” mean? We know that after register allocation, we might have spills, which go into the stack automatically. We are assuming a coloring-based register allocator similar to Briggs’ allocator, but not exactly, as implemented in GCC. “Reloading” can be classified into either an “input reload” or an “output reload”. “Input reload” means reloading the values stored in memory into physical registers. “Output reload” means reloading the values of physical registers back into memory. The reloading phase ensures that spilled variables are loaded correctly into registers for execution, and stored back into memory if necessary. For our purposes, SOA comes right in the middle of the reload phase in GCC. Specifically, it is done after all spills and stack variables have been determined without any further changes, and before any real, physical stack offset is given to the virtual

registers. Virtual registers placed in stack are represented as a register number in the RTL, but with a corresponding non-null value in “reg_equiv_mem[REGNO]”, which gives the pseudo stack offset. This pseudo stack offset is then adjusted by a constant integer value to give the actual, physical stack offset.

To perform SOA with no coalescing, we follow these steps:

1) At compiler initialization, reserve one AR for use in SOA.
2) Identify all variables suitable for offset assignment.
3) Construct webs.
4) Build access graph.
5) Run SOA algorithm on access graph.
6) Rearrange stack variables physically.
7) Use the reserved AR to access these variables.

6.2.1 Reserving an Address Register

In Step 1, we need to reserve one address register so that the register allocator will not allocate that register to any variable. Furthermore, we need to make sure that the final code generator will not use that register also. The first problem is that some registers cannot be reserved. The Motorola GCC compiler will use these registers even when you reserve it. The way to reserve an address register is by setting the “fixed_regs[REGNO] = 1;”, so that the register allocator will not use it. Then we still need to set “regs_ever_live[REGNO] = 1” to make sure that the code generator does not use it. We found that only four of the eight ARs, R3, R0, R4 and R5, can be reserved properly, such that if reserved, these ARs will never appear in the generated code unless we write code to use them.

6.2.2 Identifying All Variables Suitable for Offset Assignment

In Step 2, we find out which variables can actually be used for offset assignment. There are six conditions that a pseudo-variable (exact same thing as virtual register) must satisfy in order for it to be suitable for offset assignment:

1) The variable must reside in stack in order to be used. This is obvious because we are trying to arrange the stack layout.

2) The variable must not have any escaping uses. We only used "reload-generated" stack variables because these variables are generated by the compiler and will never have an escaping use.

3) The variable must not be an array variable, which may occupy more than one word in memory. Arrays are not considered as part of the target of SOA optimization.

4) The variable must not be a parameter passed to called functions. We cannot rearrange such variables because they must always appear in the given order.

5) The variable must occupy exactly one word in size. The DSP56300 processor only supports auto-modify for an offset distance of one word. Whenever we have multi-word variables, we can use the offset registers to try to save LDARs.

6) The variable must not overlap with any other variables in the stack. Whenever we have an overlap, it means either that the variable itself is multi-worded, or that the variable is a portion of a multi-worded variable. Such variables cannot be used because they must be placed together with all the other variables that they overlap with.

These conditions are very specific to the compiler we used, but the general idea should apply that all variables that may violate the correct semantics of offset assignment cannot be used.

6.2.3 Constructing Webs and Access Graph

Constructing webs is a matter of performing liveness analysis, and then breaking up live ranges into atomic units. Constructing the access graph is a matter of running through the access sequence of each basic block in the CFG. Both of these are actually theoretical constructions and did not give any implementation problems.

6.2.4 Running SOA Algorithm on Access Graph

The SOA algorithm is a purely graph-theoretical algorithm that does not require any code modification during execution. So far, all the data that we gathered in the previous phases are for the sake of running the SOA algorithm. Therefore, the data structures were also designed for use in the SOA algorithm. In order to support coalescing, we had to assume that every node is a coalesced node in order for the same SOA algorithm to work both with and without coalescing information.

6.2.5 Rearranging Stack Variables Physically

The SOA algorithm produces the layout, which is the solution that we want. In order to actually use this layout, we need to rearrange all affected stack variables. Earlier we mentioned “overlapping variables”. For all variables that overlap one another, we say they belong to the same “overlap set”. Consider Figure 24 which shows the stack offsets of six variables, A to F. Variables A, B and C are in the same overlap set, because the overlap one another. Similarly, variables D, E and F are in the same overlap set. All variables in the same overlap have to be arranged in stack in the same sequence as they are in the overlap set. The point to note is that although A and B do not overlap each other, they are still considered as overlapping because C overlaps both of them.

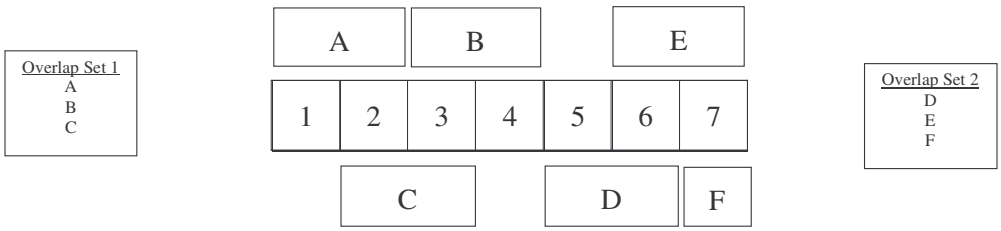


Figure 24. Illustration of Overlap Sets

When making the rearrangements, we must also take arrays into account. Arrays occupy a certain number of bytes in stack, and this space must be maintained even when arrays are moved to a different stack offset.

It was quite a hassle trying to get this part right. When we were trying to modify the RTL (register transfer language), we cannot modify the "CONST_INT" RTX (register transfer expression) directly because they might be shared among more than one RTX. The discovery of the overlap sets and excluding them all from the offset assignment consideration was also a painful process.

6.2.6 Using the Reserved AR to Access these Variables

The final part of the SOA optimization is to actually use the reserved AR to account for all the loads and stores of all variables in consideration. As given in [Liao 1995; 1996], we only do it on an intra-basic-block level. We start by running down the basic block and keeping track of the last memory load/store instruction, if any, the current base AR on which the offset is based, and the current offset stored in the AR. For example, if we are trying to use R3 for an address R6+10, then R6 is the base AR for R3 and 10 is the current offset of R3. Hence, if we encounter an "R6+10" expression down the basic block, we can simply replace it with "R3". If we see "R6+11", we use a post-

increment for the last load/store, and then replace “R6+11” with R3. If we see “R6+9”, we use a post-decrement for the last load/store, and then replace “R6+9” with R3. In this way, we use R3 for all loads and stores of all variables in consideration.

This step is achieved by traversing through the entire RTL and modifying it. Whenever we see a variable load/store instruction that is considered under offset assignment, we will use one of the reserved ARs to realize the address of the load/store. Modifying the RTL automatically affects the final generated code.

In our case, since we are considering stack variables only, the base AR is always the stack pointer R6.

There are three possible causes of problems here. Assume we are using R3:

1) Whenever the base register is written to with a non-constant value, we mark the current value of R3 as unknown.

2) Whenever the base register is modified by an auto-modify addressing mode, the current offset of R3 must be modified in the opposite direction. For example, if we get “MOVE (R6)+”, which means “R6 <- R6 + 1”, and the current base register of R3 is R6 and the current offset of R3 is 10, then we decrement the current offset of R3, so that it becomes 9.

3) Whenever we come across a function call when traversing down a basic block, we mark the current value of R3 as unknown. It does not matter whether R3 is a caller or callee save because we already know that caller/callee save code is more expensive than LDARs. This is because a caller/callee save requires at least two instructions, while an LDAR is only one instruction.

Note that this is perhaps the simplest possible scheme of using address registers for offset-assigned variables. The advanced version of this scheme is the post-pre optimization, which requires an elaborate analysis at the CFG level.

6.2.7 Conclusion for SOA Implementation

As like most other compiler optimizations, everything has to be perfectly correct in order for the optimization to work correctly. We cannot miss out any one of the mentioned details. If we miss out any one little detail, the whole thing will not work correctly and the compiler will not generate correct code.

One should find that the above-described seven steps are simple enough to understand and implement, because SOA is the simplest implementation among all the different optimization techniques discussed in this work. The rest of the techniques are built upon the work of SOA, and involves several more considerations and major steps.

The lesson is that we have to understand the important points about the bulk of the compiler code that we did not write, if we were to actually write extensions to the compiler. If we start writing code and making modifications without knowing the compiler well enough, then we might run into subtle problems later. These problems could be fundamental in that we might have to rewrite a large chunk of code later because the base methodology was faulty (which really happened in our case). Also, we found that writing code that deals with the CFG usually gives much fewer problems than writing code that deals with the code generator because the code generator is inherently far more complex.

6.3 Implementation Details for Coalescing

Coalescing is really implemented as an additional step to SOA. In OpCost SOA, the coalescing and SOA are performed simultaneously. In OpSize SOA, the coalescing is performed before SOA, and then the coalesced access graph is fed to the standard SOA algorithm as input. Therefore, the standard SOA algorithm we use should always consider the nodes as "coalesced nodes" (as mentioned before). This means that we need a coalescence mapping of old to new variables, which is simply an array of integers because each variable is represented as an integer.

The steps to perform SOA with coalescing are very similar to those in SOA, except for the ones marked with an asterisk:

1) At compiler initialization, reserve one AR for use in SOA.
2) Identify all variables suitable for offset assignment.
3) Construct webs.
4) Build interference graph. *
5) Build access graph.
6) Run OpCost or OpSize SOA algorithm on access graph and interference graph.
7) Renumber coalesced virtual registers. *
8) Rearrange stack variables physically. *
9) Use the reserved AR to access these variables.

6.3.1 Building the Interference Graph

In Liao's SOA, the interference graph is not necessary at all. However, for coalescing, clearly we have to construct the interference graph so that we know which nodes we cannot coalesce. Building the interference graph in GCC is not difficult because

it was already done in “flow.c”, and we just have to use a slightly modified version of the liveness analysis code it already has.

6.3.2 Renumbering Coalesced Virtual Registers

To model the coalescing, we map all the coalesced virtual register numbers into the C-node virtual register number in the RTL. The C-node virtual register number is always the number of an already existing virtual register. By doing so, we automatically make them into the same variable, thus coalescing them.

6.3.3 Rearranging Stack Variables Physically

This step is essentially the same as prescribed for SOA, except that now we have to possibly put several variables into the same stack location. Coalescing is done in the renumbering step right before this step. Whenever we perform any coalescing, we reduce the stack size needed. Therefore, the only additional step is to decrease the stack size accordingly, which is only a matter of subtracting the value of an integer “frame_offset” in our case.

6.4 Implementation Details for Using Offset Registers

Offset registers are designed mainly to provide for addressing-based optimizations, and hence they proved really useful in what we are doing. Our compiler uses N0 to N7 as “code generator temporaries” only [Motorola 2000]. Specifically, this means that these registers are used to store an immediate offset value that goes beyond the range of -64 to 63, because address registers, when modified by a constant immediate value, can only be modified by this range. Address registers cannot be modified by using

accumulator registers A and B, nor by x-memory and y-memory input/output registers X and Y. Therefore, the way to modify an address register by a large offset value is to assign a register's value or an immediate constant value to an offset register, and then modify the address register by the value in the offset register.

6.4.1 Example of Using an Offset Register

For example, say we want to add an address register R3 by 500. In DSP56300, this requires two instructions. First, we do "MOVE #500, N3", which is to set the value of N3 to 500. Then we do "MOVE (R3)+N3", which is to post-add R3 by the value of N3, effectively increasing R3 by 500. N3, used as a code generator temporary, is not live before and after these two instructions. In immediate representation (RTL), these two instructions are represented as "SET R3 ← R3 + 500".

In this way, we can see that the code generator does not need to use all eight offset registers, since each time the offset register is used, it is only live in two instructions. In fact, the code generator uses only one offset register, N6. Since any offset register can be used with any address register, N6 is more or less an arbitrary choice, mainly because R6 was set aside for a special purpose, so was N6. Thus, we have seven offset registers available for use.

6.4.2 Methodology for Using Offset Registers

We already explained that caller and callee save instructions are more expensive than using LDARs themselves, so we will consider that we will not save the value of any offset registers across function boundaries. In a static context of a whole executable program being compiled, we assign offset registers globally, initializing their values at

the start of function “main” and then using these same values throughout the entire program.

GCC does not support the framework of global optimizations particularly well because it compiles one function at a time, from the parsing of the source code to the emission of assembly code before proceeding to the next function. Any function that is marked “inline” will be saved and used for inlining when they are encountered, but the memory for storing all other functions will not be retained after their compilation is completed. We need the information from all the functions to determine the best offset register values to use. Therefore, we can save all the RTL generated (by not freeing memory) and then emit the code only after we obtained all the functions. However, doing so will require some kind of substantial change to the compiler. Instead, we perform the offset register optimizations in two passes. In the first pass, we keep track of the number of LDARs that we can save by having each offset register having a particular constant integer value. Then between the first and second passes, we can choose the seven offset values that can save us the most number of LDARs, and then assign these values to the offset registers in the “main” function prologue. In the second pass, we simply use these offset registers with these known constant values to save LDARs.

In a real compiler, we can save all intermediate code and do global optimization last. But in this experimental environment, we are only concerned with the effects of the optimization, and hence we chose the simpler implementation which also works fine.

6.4.3 Recent Trends in Offset Registers

We understand that the DSP56300 compiler might be peculiar in that it does not use seven offset registers, which is really a waste of available hardware resources. This is

because the compiler does not contain such an optimization as the one we are proposing in this work. One should expect that in other DSP compilers, there could be a different number of offset registers and they could be reserved for different uses.

The StarCore architecture is a successor of the Motorola DSP series of processors that features VLES (variable-length execution set) execution. StarCore has only four offset registers, while having 16 ARs. Recall that DSP56300 has eight offset registers and eight address registers. Therefore, we can easily see that Motorola processor designers feel that ARs are in more demand than offset registers, and we do not need as many offset registers as ARs. We expect the trend in newer DSP processors to be towards having more address registers than offset registers. Therefore, optimizations using offset registers will prove to be a very different problem in more modern DSP processors. We also expect that the savings that we obtained in DSP56300 due to using offset registers could be much larger than that in other architectures.

6.5 Implementation Notes for Other Optimizations

In the previous sections, we did not discuss anything about the implementation details for Post-pre optimization, Inter-basic-block offset assignment, and GOA. This is because each of these optimizations are just slight variants of the already-discussed optimizations. For Post-pre optimization, the only additional step we have is to determine the auto-modify addressing mode to use for each load/store of each offset-assigned variable. For Inter-basic-block offset assignment, we only need to run a different graph-theoretic SOA algorithm, except that this algorithm requires the CFG as input. For GOA, instead of running the SOA directly, we first run them through the GOA set partitioning algorithm. Therefore, in terms of the implementation intricacies, there is not much we can

discuss except for the data structures used, and that is not worthy of discussion in here. Our focus on all these discussions has been on generating correct code and dealing with compiler issues, because those issues are much harder to figure out.

6.6 Conclusion for Implementation Details

The main bulk of the work has been in resolving issues that has to do with the compiler. When implementing our own proposed algorithms, they are relatively fast to complete because we derive every detailed step of it from scratch. But when dealing with the compiler, we face many thousands of lines of code that we did not write, and hence we do not know for sure that we can add code on top of it that runs correctly.

Thus, most of the time is spent in fixing up the compiler issues, and only a little portion of the time has been spent in actually implementing our own algorithms. In all, it has been a difficult task just to get the compiled programs to execute correctly with the optimizations.

7. PERFORMANCE EVALUATIONS

7.1 Measuring LDAR Counts

Existing work uses the metric of "SOA cost" to measure the effectiveness of layout assignment algorithms. In this work, we consider SOA cost as a purely theoretical number that does not predict the LDAR count accurately. Since our optimization objective is to minimize the number of LDARs, the LDAR count becomes a natural metric to use in order for us to know how well our algorithm performed. In our experimental evaluations, we only consider the LDAR count, but not the SOA cost. The LDAR count is a realistic measure of the effectiveness of a layout assignment algorithm. We do encourage any future work in this area of research to present results in LDAR counts instead of SOA cost because a theoretically good solution might not map to a realistically good solution.

7.2 Benchmarks Description

A total of 9 benchmarks were used for evaluation. Among them, 4 are from Mediabench, 4 are from MiBench and 1 from Spec2000Int. These benchmarks represent a combination of real DSP-related applications, such as adpcm and g721d, and also practical utility programs such as bzip2 and strsrch. All benchmarks are run up to 2 million cycles. Limiting the execution time is necessary because large benchmarks may take an unreasonable amount of time to finish execution (months). Many benchmarks could not be included in our experimental runs because they run inherently complex algorithms that could not finish in a reasonable amount of time. However, they can be

successfully compiled. We use access graphs built using profile information for all results, i.e. access graphs are based on information gathered in test runs.

Table 2 shows some properties for the benchmarks. The second column is the code size in bytes. The third column shows the BaseSOA (as mentioned in Chapter 3.6.1, we use the Tie-break SOA algorithm [Leupers 1996]). We will compare our approaches with it. Notice that, since we are not able to optimize the library code, all statistics in Table 2 are for user code only. Our optimization does not affect the assembly code data section size, and hence we only list the text section size. The LDAR count corresponds to using one AR without any layout assignment optimization. The rightmost column is the initial stack slot count before coalescence.

Table 2. Statistics for the Benchmarks

	test suite	.text size	LDARs	# slots
adpcm	mediabench	6413	46	12
bmath	mibench	11486	28	12
bzip2	spec2000	25512	1521	530
crc32	mibench	6003	30	10
epic	mediabench	23569	1297	304
g721d	mediabench	10469	397	198
mpeg2d	mediabench	34732	1741	735
patricia	mibench	12400	181	49
strsrch	mibench	7530	132	40
average		15346	597	210

In the following sections, we present the results for stack size, LDAR count, code size, and execution cycle count for some combinations of optimizations applied. Due to the high number of combinations we can have with the different algorithms we proposed, and also the varying number of ARs for each result set, we can only selectively include certain result sets, but not all of them.

7.3 Results for Stack Size Reduction

We first look at how coalescence-based offset assignment performs when only one AR is considered. Two optimizations i.e. either OpCost or OpSize are compared together with the original and baseline Tie-break SOA algorithm.

Figure 25 shows the stack size reduction. BaseSOA does not change the stack size, because no coalescence is engaged. The average stack size reduction is 11.5% and 12.0% for OpCost and OpSize respectively. GCC generates a large number of temporaries, and these temporaries have short live ranges, therefore their stack slots can be easily coalesced with other variables. OpSize is more powerful in reducing the stack size. As mentioned earlier, the OpSize algorithm first attempts to coalesce the stack slots as much as possible, then invoke the SOA solver, leading to a smaller footprint on the stack than OpCost. However the difference is not very significant between OpCost and OpSize, showing that coalescence also contributes heavily in cutting down the SOA cost.

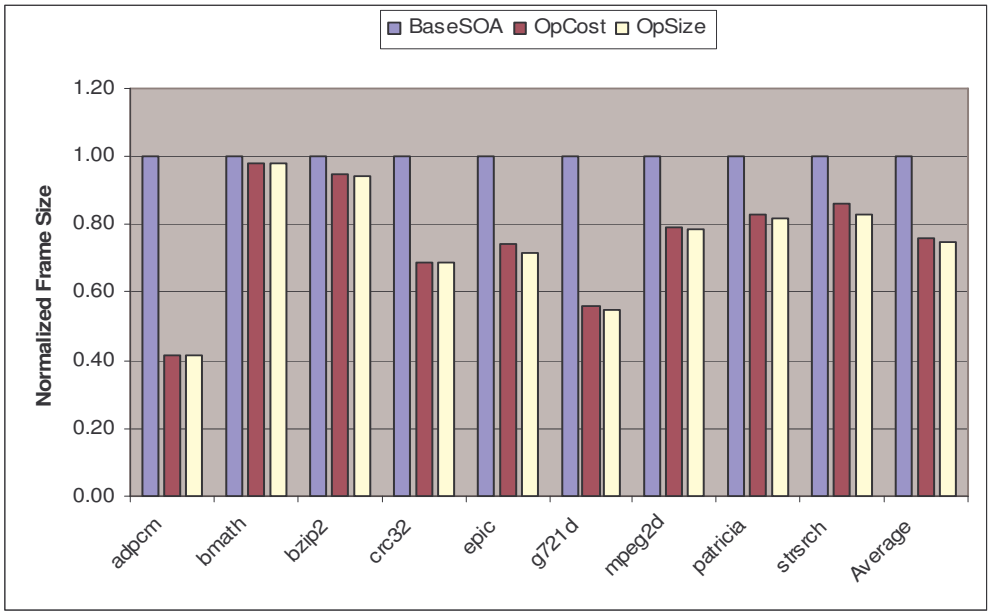


Figure 25. Stack Size Reduction

In this work, stack size savings is achieved by using coalescence only. The other optimizations do not yield any stack size savings. Hence, figures in stack size reduction is presented only once here. Note that all offset-assignment-irrelevant stack memory such as arrays cannot be reduced, but are counted with the total stack size figures shown here.

7.4 Results for Single-AR

In SOA, we only use one address register for all memory accesses of offset-assigned variables. We look at how the optimizations affect the LDAR count, code size and execution cycles.

7.4.1 Results for Single-AR LDAR Count

In Figure 26, all LDAR counts are normalized to the original ones. The LDAR count for BaseSOA is usually smaller than that for the unoptimized code for all benchmarks, however coalescence-based approaches do not improve that by much. This is mainly because BaseSOA achieves a solution close to the optimal solution, and it is not easy to go beyond this near-optimal solution even with coalescing or Inter-Block SOA. Actually we had expected some savings here, but as we mentioned earlier, the savings in SOA cost does not reflect the savings in LDARs accurately.

On average, BaseSOA and InterBlock SOA reduces the LDAR count by 2.77%, while OpCost and OpSize both achieve 2.75% reduction.

Applying post-pre optimization with BaseSOA reduces the LDAR count by 15.24%. This high number shows that the often-neglected pre-increment and pre-decrement addressing modes, when fully utilized, can potentially bring about large savings. Note that our target environment, DSP56300, only supports pre-decrement, and

does not support pre-increment. If we do have pre-increment as well, the savings should increase by a little.

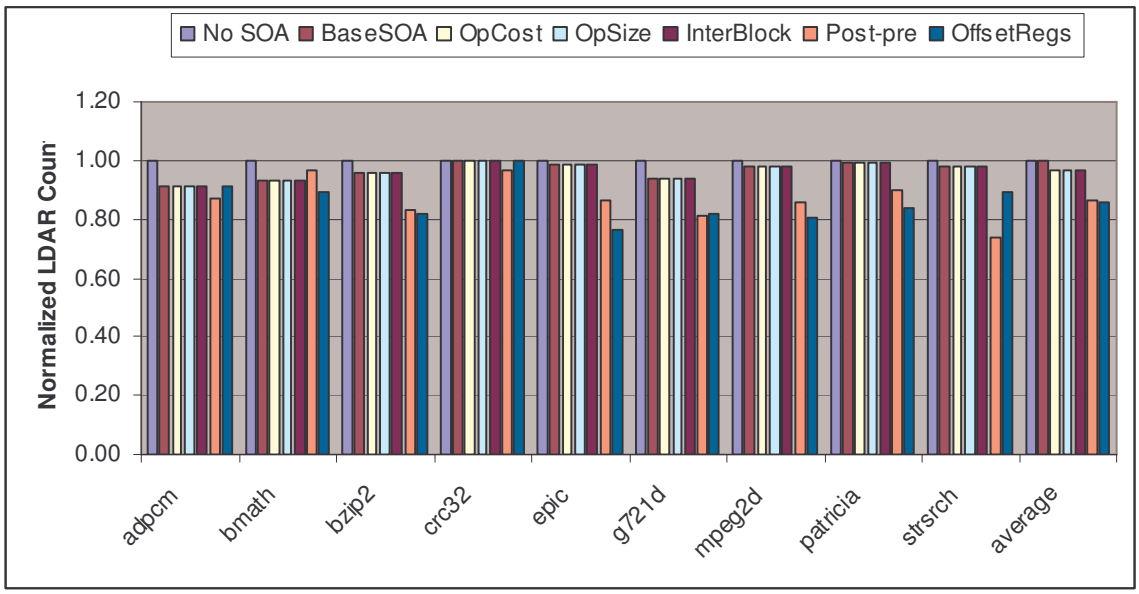


Figure 26. Results for Single-AR LDAR Count

When we use the offset registers globally with BaseSOA, we reduce the LDAR count by 19.5% over layout-unoptimized code. We should not be too optimistic about this figure because we noted earlier that the current trend is to build in less offset registers because their potential had never been fully realized.

7.4.2 Results for Single-AR Code Size

Figure 27 shows the effects of SOA on code size. The code size reductions are 0.32% for BaseSOA, 0.33% for OpCost, 0.33% for OpSize, 0.32% for InterBlock, 1.80% for post-pre and 2.27% for using offset registers. Code size savings is generally small.

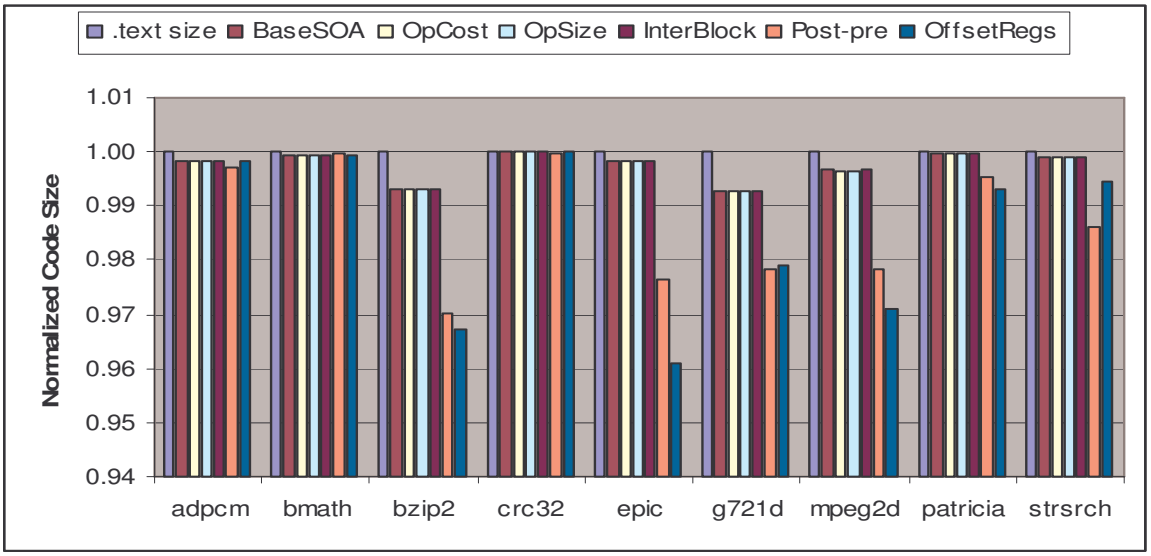


Figure 27. Results for Single-AR Code Size

7.4.3 Results for Single-AR Execution Cycles

In Figure 28, percentage numbers are pictured for the benchmarks. Reductions are 0.38% for the first four SOA algorithms, 3.30% for post-pre, and 4.10% for using offset registers. We can thus see that having a good layout arrangement is not adequate for improving the execution speed. We need to bring in other forms of optimizations in order to achieve some kind of savings.

Benchmarks bzip2 and epic get higher speedup because less library code are involved. All library code comes in pre-compiled form and did not go through our optimizations. Moreover, memory access instructions make up about 1/3 of the instructions in the generated code. If we had used a register-scarce architecture in our tests, there would be more spills, thus creating more memory access instructions. Thus, if more memory instructions can be handled by our algorithm, we will probably gain a

bigger cycle reduction. Therefore, our algorithms can be more effective on register-scarce architectures or memory intensive applications.

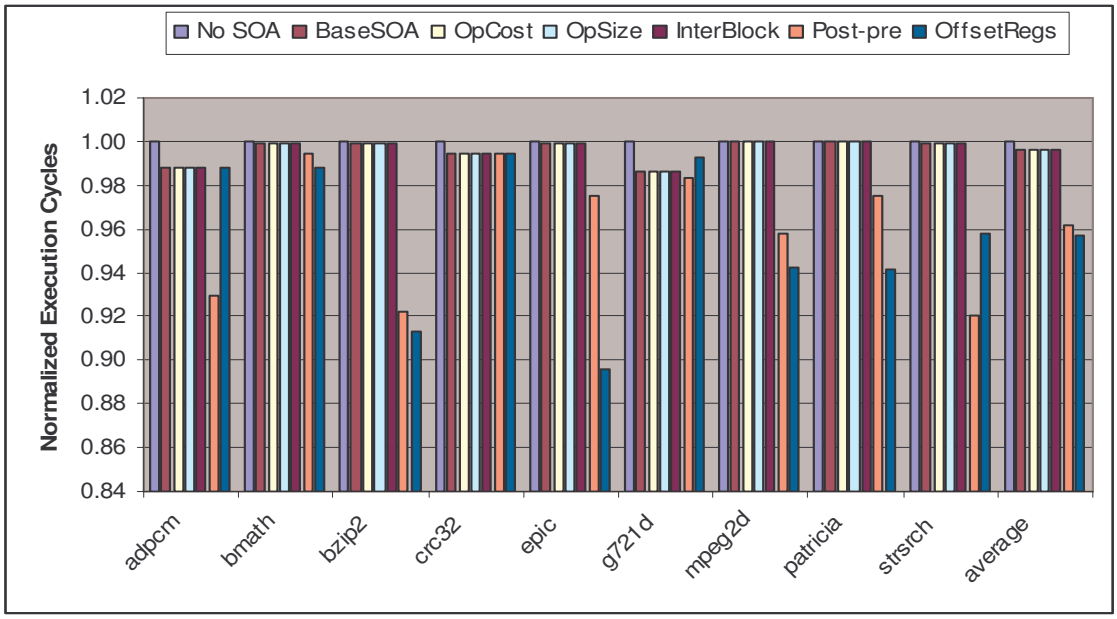


Figure 28. Results for Single-AR Execution Cycles

7.5 Results for Multiple-AR

With multiple-AR, we expect better performance results, however investing more ARs is actually not always rewarding, because the optimization space will reach a plateau once we use a certain number of ARs. Here, we vary the number of ARs to look at the sensitivity towards several performance metrics. Notice that, the total number of address registers is fixed. Therefore if more address registers are reserved for auto-modify modes, less address registers will be available for other purposes like heap accesses.

In Figure 29, we compare the GOA cost along two dimensions. We vary the number of address registers from 2 to 4 and use the three algorithms BaseSOA, OpCost and OpSize. Therefore we show 9 bars for each benchmark. For each benchmark, values

are normalized to the first bar, i.e. 2-AR base-GOA. The leftmost three bars correspond to 2AR, the center three to 3AR, and the rightmost three to 4AR. In most cases, we observe lower cost when we use more ARs.

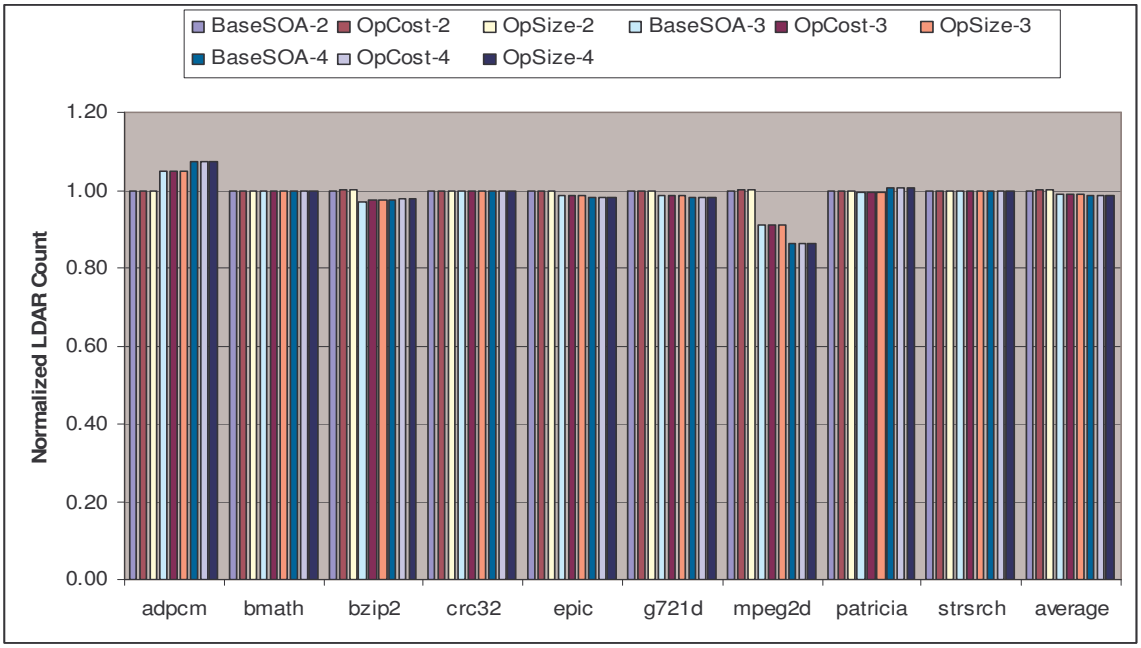


Figure 29. Results for Multiple-AR LDAR Count – 2 to 4 ARs

Code size and execution cycles are not shown here because they bear a close correspondence to LDAR count. The LDAR count enables us to calculate the generated code size. Also, because we know how many LDARs we saved, we can roughly estimate how much speed-up we can obtain in the generated program.

7.6 Results for Overall Performance Comparison

Here, we evaluate the overall performance, including coalescence-based offset assignment together with post-pre optimization and using offset registers.

In order to obtain the strongest optimization combo from the techniques of this work, we perform the following optimizations in sequence:

1) OpCost SOA

2) Post-pre optimization

3) Use offset registers

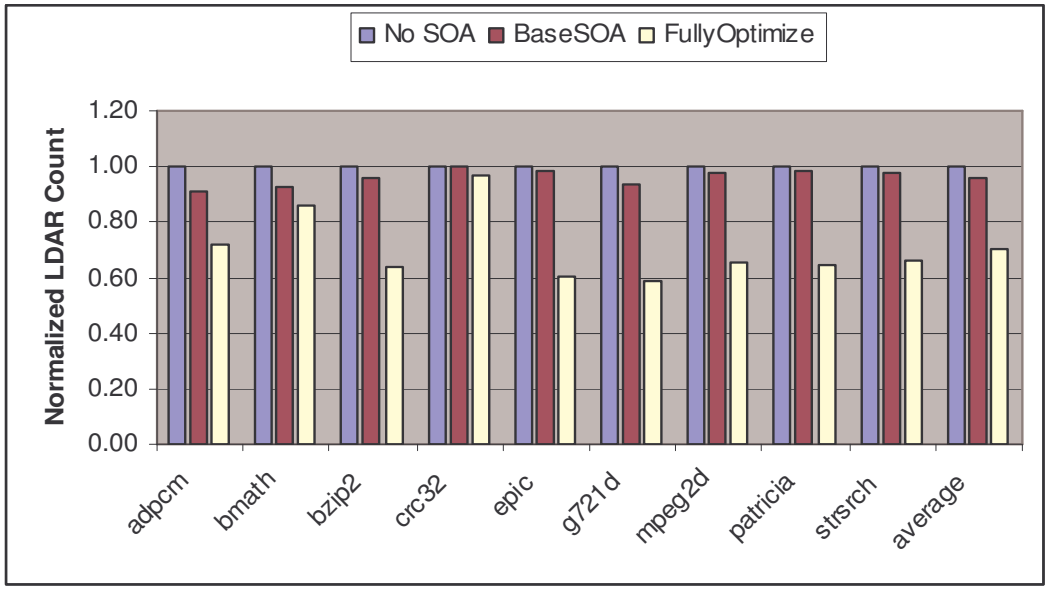


Figure 30. Overall LDAR Comparison between BaseSOA and Full Optimizations

On average, we obtain 2.77% LDAR reduction with BaseSOA, and 36.5% LDAR reduction with full optimizations turned on. This is quite a significant number. We would suspect that implementing these same optimizations on other architectures would yield a lesser percentage reduction because some DSP processors do not have any pre-modify addressing modes, and some DSP processors have less offset registers.

7.7 Compilation Time

Table 3 shows the compilation time of each optimization stage on a 1GHz Pentium III machine. Here, only Twolf and VPR are listed, because the other benchmarks usually finish compilation within a small amount of time (less than 2 seconds). Both Twolf and VPR could not be used as our primary benchmarks because they cannot finish execution on the simulator within any reasonable amount of time. However, due to their huge size, they are perfect for use in measuring compilation time.

Table 3. Compilation Time (in seconds)

Bench-mark	Orig.	1 AR			2 AR			3 AR			PostPre + Offset
		Base	OpCost	OpSize	Base	OpCost	OpSize	Base	OpCost	OpSize	
twolf	7.6	8.0	98.6	15	226	132.6	18.6	237	130.6	18.6	3.7
vpr	3.5	3.52	4.0	3.58	13.5	6.2	4.3	15.5	6.8	4.2	0.9

Column “Orig.” shows the compilation time for the original code, while the rightmost column stands for the time on post-pre optimization with using offset registers. We only give the number for single-AR, since this number only varies slightly across different configurations. The columns in the middle are grouped according to the number of ARs. For each group, we show the compilation time with BaseSOA, OpCost and OpSize. For single-AR, BaseSOA is fastest, while for multiple-AR, it takes a long time to finish. In general, OpSize is much faster than OpCost, because the OpSize algorithms first do a minimal graph coloring to aggressively coalesce nodes on the graph without considering the SOA/GOA cost. Stack-based graph coloring [Briggs 1989] finishes execution quickly. After this step, the resulting access graph and interference graph are much smaller. Hence later steps for OpSize, although they are quite similar to OpCost, can be executed in a shorter time period due to reduced problem size. Besides, for

Simple-AR the OpCost algorithm involves a loop that calls the SOA solver multiple times, causing longer compilation time. Finally, after analyzing the compilation process for Twolf, which is most time-consuming among all benchmarks, we found that actually the majority of the compilation time is spent on several extraordinarily big procedures, because OpCost has time complexity $O(N^4)$, where N is the number of offset-assignment-relevant variables. Thus, in a typical program with smaller functions, compilation time will be very fast.

7.8 Access Sequence Lengths

In an attempt to explain the reasons behind the performance figures we obtained, one of the factors we dug into was the access sequence length. Existing literature on offset assignment optimizations tend to use some long pseudo access sequence of 10 or more memory accesses as illustrations. By running the optimization algorithm on those access sequences, one can often obtain pretty satisfactory results. Here, we would like to present some numbers of the access sequence lengths in Table 4.

Table 4. Average and Longest Access Sequence Lengths

Benchmark	Average Length	Longest Length
adpcm	1.23	3
bmath	1.39	3
bzip2	1.88	13
crc32	1.21	2
epic	2.18	23
g721d	1.82	19
mpeg2d	2.08	110
patricia	1.40	5
strsrch	1.54	7
average	1.64	20.56

As Table 4 shows, surprisingly, mpeg2d ("d" means decoder) has a longest access sequence length of 110 consecutive memory accesses of offset-assignment-relevant variables. In the big picture, we see that the average access sequence of each program is only 1.64. This means that many access sequences consist of only one memory access, and cannot be optimized for no matter what kind of layout assignment we have. We have to use an LDAR to realize that one address needed.

The main reason for such short access sequences is that access sequences are always broken by function calls and function boundaries. We already explained earlier that caller and callee save instructions are more expensive than LDARs, both in terms of code size and execution cycles. Therefore, whenever we come across a function call, we have to break the access sequence. We do not have a clear-cut solution for lengthening the access sequences while making the code better. mpeg2d having a long access sequence implies that it has a very long stretch of code that contains no function calls, which as we can see is a very unusual way to write programs.

With this information in hand, we are able to explain the limitation of the effectiveness of our varying SOA algorithms, OpCost, OpSize and InterBlock SOA. Having short access sequences is the primary reason why most of the offset assignment algorithms produce roughly the same results even though they are theoretically different. We cannot expect that an entirely theoretical solution will always yield a practically feasible solution. Sometimes it might not do so. In this case, we learnt our lesson through experimentation. We hope that this information can serve to provide an insight to the reader regarding offset-assignment-based optimizations.

8. RELATED WORK AND CONCLUSION

8.1 Related Work

Clearly, our framework incorporates some of the earlier work such as Tie-break SOA [Leupers 1996]. Also, The SOA solver used in the framework can be replaced with any existing SOA algorithms proposed in literature, such as the incremental SOA [Atri 2000], genetic algorithm [Leupers 1998] and those combined ones in [Leupers 2003]. As [Leupers 2003] pointed out, the performance difference is not very significant among existing SOA solvers and there are trade-offs between compilation time and the amount of SOA cost reduction, therefore our framework nicely separates out the SOA solver for users' own choosing and makes it very flexible to incorporate new and better SOA solvers in the future. For GOA, all existing approaches are actually quite fundamental. Also, due to the large percentage of optimal solutions obtained in this work, we can reasonably claim we are very close to the limit of this problem, leaving little space for further improvements.

We notice an independent research work on coalescence-based SOA [Ottoni 2003] came slightly later than our conference publication [Zhuang 2003]. In their paper, the coalescence algorithm is more ad hoc in terms of the selection of node pairs to coalesce and the simplified iteration stage. Actually, similar approaches have been attempted during our early experiments. Due to the fluctuation of the solution quality, we later include the iteration stage that can keep track of the best result during the coalescence process. Moreover, in an effort to reduce the regression of the intermediate solution, we decide to gradually improve it upon the previous C-PC. As an extended version, this

work talks more about GOA and newly includes the post-pre optimization, which has not been addressed by any of the previous work.

8.2 Conclusion

This work proposes a framework for better utilizing the auto-modify modes on embedded processors. Our optimization framework includes two enhancements to existing work, i.e. coalescence-based offset assignment and post-pre optimization. We have shown the advantages of coalescence over previous approaches to capture more opportunities to reduce both stack size and SOA/GOA cost. By incorporating seamlessly with an SOA solver, our framework can work with any SOA solvers, make it more flexible.

This work represents a shift in approaches that solve offset assignment problem; the ongoing research is focused on developing new heuristics for solving MWPC and program reordering which has diminishing returns due to the high density of access graphs and hardness of the problem in graph-theoretic space. This work demonstrates the capability of variable coalescence and post-pre optimization to break the performance bottleneck.

Our results show that the LDAR count can be reduced by up to 19.5% (offset registers) for Single-AR, which is much more than the LDAR reduction for a baseline solver with Tie-break SOA. On the other hand, coalescence-based approach can also shrink the stack size by a reasonable amount. As observed from the OpSize heuristic, the stack size reduction mounts to 12.0%. This percentage would be larger in a compiler that does not already reuse stack slots. For Multiple-AR, we pointed out that having too many address registers might not improve the code, because the access sequence is bounded by

function calls, which happen very frequently. Allocating more for auto-modify modes deprives the processor of registers for other purposes. Compared with the baseline GOA algorithm, variable coalescence is equally effective for Multiple-AR.

We evaluate add-on optimization stages after coalescence-based offset assignment and observe up to 36.5% LDAR reduction with both post-pre optimization and offset registers enabled. The amount of cost reduction is quite stable as indicated by our experiments with combination to the either the baseline SOA or OpCost algorithm. In short, performing variable coalescence and other optimizations after offset assignment like the post-pre optimization gives new opportunity to exploit auto-modify mode on a wide variety of DSP processors, dramatically improves the solution space of this important problem and achieves significant enhancements as demonstrated in our results.

We hope that our discussion on using LDAR counts instead of SOA cost have been refreshing, and our finding on the typical average access sequence length of 1.64 shines some light on the true nature of offset assignment optimizations.

REFERENCES

Aho, A. V., Sethi, R. and Ullman, J. D. 1986. Compilers: Principles, Techniques and Tools. Addison-Wesley, Reading, Mass.

Araujo, G., Sudarsanam, A. and Malik, S. 1996. Instruction set design and optimizations for address computation in DSP processors. In Proceedings of the 9th International Symposium on Systems Synthesis, IEEE, 31–37.

Araujo, G., Ottoni, G. and Cintra, M. Global Array Reference Allocation. ACM TODAES, April 2002.

Atri, S. 1999. Improved code optimization techniques for embedded processors. Master's thesis, Department of Electrical and Computer Engineering, Louisiana State University.

Atri, S., Ramanujam, J. and Kandemir, M. 2000. Improving variable placement for embedded processors. In Proceedings of the Languages and Compilers for High-Performance Computing.

Bartley, D. H. 1992. Optimizing stack frame accesses for processors with restricted addressing modes. Software – Practice and Experience, 22, 2 (Feb.), 101–110.

Bowman, R. L., Ratliff, E. J., and Whalley, D. B. Decreasing Process Memory Requirements by Overlapping Program Portions. In Proceedings of the Hawaii International Conference on System Sciences, Jan 1998, vol 7, pg 115-124.

Briggs, P., Cooper, K., Kennedy, K. and Torczon, L. Coloring heuristics for register allocation. In Proc. of the ACM SIGPLAN 1989 on Conference on Programming Language Design and Implementation. pp. 275-284. July 1989. (PLDI)

Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M. E. and Markstein, P.W., 1981. Register allocation via coloring, Computer Languages Vol.6, No.1, pp.47-57.

Chaitin, G. J. 1982. Register allocation and spilling via graph coloring. In Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction.

Ganssle, J. G. 1992. The Art of Programming Embedded Systems. Academic Press Inc., Reading, San Diego, CA.

Gebotys, C. 1997. DSP address optimization using a minimum cost circulation technique. In Proceedings of the International Conf. on Computer-Aided Design (ICCAD), IEEE, 100–103.

Kandemir, M., Irwin, M. J., Chen, G. and Ramanujam, J. 2003. Address Register Assignment for Reducing Code Size. In Proceedings of the Twelfth International Conference on Compiler Construction (CC'03).

Lee, C., Potkonjak, M. and Mangione-Smith, W. H. 1997. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In Proceedings of the International Symposium on Microarchitecture.

Leupers, R., Basu, A. and Marwedel, P. 1998. Optimized array index computation in DSP programs. In Proceedings of the ASP-DAC (Feb.), IEEE.

Leupers, R. and Marwedel, P. 1996. Algorithms for address assignment in DSP code generation. In Proceedings of the International Conf. on Computer Aided Design (ICCAD), 109–112.

Leupers, R. and David, F. 1998. A Uniform Optimization Technique for Offset Assignment Problems, In Proceedings Int'l System Synthesis Symposium (ISSS).

Leupers, R. 2003. Offset Assignment Showdown: Evaluation of DSP Address Code Optimization Algorithms. In Proceedings of the Twelfth International Conference on Compiler Construction (CC'03).

Leupers, R. and Araujo, G. Address Code Optimization. http://www.address-code-optimization.org/

Liao, S. Y., Devadas, S., Keutzer, K., Tjiang, S. and Wang, A. 1995. Storage assignment to decrease code size. In Proceedings of the ACM SIGPLAN Conf. on Program. Lang. Design and Implementation (PLDI), 186–195.

Liao, S. Y., Devadas, S., Keutzer, K., Tjiang, S. and Wang, A. 1996. Storage assignment to decrease code size. ACM Trans. on Program. Language and. Systems. 18, 3 (May), 235–253.

Motorola Inc. Motorola DSP56300 Family Manual, Revision 3.0, Nov. 2000.

Motorola Inc. SC140 DSP Core Reference Manual, Revision 3.0, Nov. 2001.

Motorola Inc. Motorola DSP56300 Family Optimizing C Compiler User's Manual. Motorola Inc., User's Manual.

Muchnick, S. S. 1997. Advanced Compiler Design and Implementation. Morgan Kaufman, Reading, San Francisco, CA.

Naveen S. and Sanjiv Kumar, G. Optimal Stack Slot Assignment in GCC. GCC Developers Summit, Ottawa, May 2003.

Ottoni, D., Ottoni, G., Araujo, G. and Leupers, R. 2003. Improving Offset Assignment through Simultaneous Variable Coalescing. In Proceeding of the International Workshop on Software and Compilers for Embedded Systems (SCOPES).

Ottoni, G., Rigo, S., Araujo, G., Rajagopalan, S. and Malik, S. 2001. Optimal live range merge for address register allocation in embedded programs. In Proceeding of the International Conf. on Compiler Construction (CC).

Rao, A. 1998. Compiler optimizations for storage assignment on embedded DSPs. M. S. Thesis, Dept. of ECECS, Univ. of Cincinnati.

Rao, A. and Pande, S. 1999. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In Proceedings of the ACM SIGPLAN Conf. on Program. Lang. Design and Implementation (PLDI), 128–138.

Stallman, R. 2002. Using the GNU Compiler Collection. Free Software Foundation, User's Manual, Boston, Mass.

Stallman, R. 2002. GNU Compiler Collection Internals. Free Software Foundation, Reference Manual, Boston, Mass.

Sudarsanam, A., Liao, S. and Devadas, S. 1997. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. In Proceedings of the ACM/IEEE Design Automation Conference (DAC), 287–292.

Sudarsanam, A., Malik, S., Tjiang, S. and Liao, S. 1997. Optimization of embedded DSP programs using post-pass data-flow analysis. In Proceedings of the 1997 International Conf. on Acoustics, Speech, and Signal Processing (ICCAD).

Udayanarayanan, S. and Chakrabarti, C. 2001. Address code generation for digital signal processors. In Proceedings of the 38th Design Automation Conf. (DAC).

Zhang, Y. and Yang, J., 2003. Procedural level address offset assignment of DSP applications with loops. In Proceedings of the Int'l Conference on Parallel Processing. (ICPP).

Zhuang, X., Lau, C. and Pande, S. 2003. Storage assignment optimizations through variable coalescence for embedded processors. In Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems. (LCTES). San Diego, CA. June 2003. pp. 220-231.