An Optimization Framework for Embedded Processors with Auto-Addressing Mode

11

XIAOTONG ZHUANG
IBM T.J. Watson Research Center
and
SANTOSH PANDE
Georgia Institute of Technology

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

Liao [1995, 1996] categorized this problem as Simple Offset Assignment (SOA) and General Offset Assignment (GOA), which involves storage layout of variables and assignment of address registers, respectively, proposing several heuristic solutions. This article proposes a new direction for investigating the solution space of the problem. The general idea [Zhuang 2003] is to perform simplification of the underlying access graph through coalescence of the memory locations of program variables. A comprehensive framework is proposed including coalescence-based offset assignment and post/pre-optimization. Variables not interfering with others (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; it also reduces the address register pressure to such low values that some GOA solutions become optimal. Moreover, it can reduce the memory footprint both statically and at runtime for stack variables. Our second optimization (post/pre-optimization) considers both post- and pre-modification mode for optimizing code across basic blocks, which makes it useful. Making use of both addressing modes further reduces SOA/GOA cost and our post/pre-optimization phase is optimal in selecting post or pre mode after variable offsets have been determined.

We have shown the advantages of our framework over previous approaches to capture more opportunities to reduce both stack size and SOA/GOA cost, leading to more speedup.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation; compilers; optimization*; C.1 [**Computer Systems Organization**]: Processor Architectures; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Design, Performance

Authors' addresses: X. Zhuang, IBM T.J. Watson Research Center, New York 134 & Old Kitchawan Road, Ossining, NY 10562; S. Pande, College of Computing, Georgia Institute of Technology, Rm. 253, 301 Atlantic Drive, Atlanta, GA 30332-0280; email: santosh@cc.gatech.edu.

DOI 10.1145/1734206.1734208 http://doi.acm.org/10.1145/1734206.1734208

Additional Key Words and Phrases: Offset assignment, layout assignment, digital signal processing, auto-modification addressing mode, SOA, GOA, variable coalescence

ACM Reference Format:
Zhuang, X. and Pande, S. 2010. An optimization framework for embedded processors with auto-addressing mode. ACM Trans. Program. Lang. Syst. 32, 4, Article 11 (April 2010), 41 pages.
DOI=10.1145/1734206.1734208 http:/doi.acm.org/10.1145/1734206.1734208

1. INTRODUCTION

The rapid evolution in embedded processors and DSP architectures has raised new challenges for compilers to generate code that is both efficient in terms of speed and which has a small memory footprint. In order to keep the code density high (the ratio of bits used in an instruction generated by the compiler to the ones provided by the ISA designer), embedded processors employ compact instruction sets and utilize specialized addressing modes.

Most modern embedded architectures support specialized Address Generation Units (AGUs) to facilitate the memory address computation in parallel and also to support different addressing modes to facilitate faster address arithmetic. The AGU normally provides an auto-addressing mode that performs a 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 the constraints on instruction size, traditional register-plus-offset addressing mode is either not supported (e.g., TMS320C25) or requires more instruction words (DSP56300). Therefore, transforming address arithmetic into auto-increment/decrement mode can help to generate compact and efficient code and speeds up program execution as well.

Most modern DSP processors have at least 8 address registers. For example, the Motorola DSP56300 processor [Motorola 2000] and the Sony pDSP chip each have 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-modification is supported for all these chips, and pre-modification 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 auto-increment/decrement instructions generated by the GCC compiler retargeted for the Motorola DSP 56300 chip. For most benchmark programs, less than 3% of the generated address instructions make use of the auto-increment/decrement mode before our optimizations. On the other hand, a study [Udayanarayanan and Chakrabarti 2001] shows that on some embedded processors up to 55% of memory operations could 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 classified the problem into two categories: Simple Offset Assignment (SOA) and General Offset

Assignment (GOA). The problem is modeled using an access graph and the objective is to find the Maximum Weight Path Cover (MWPC) on the graph. Liao et al. proved that finding the MWPC is NP-complete, and therefore heuristics are devised to solve both SOA and GOA. Later, Leupers and Marwedel [1996] extended Liao's work by proposing a tiebreak heuristic for SOA and a variable partitioning strategy for GOA to reduce the SOA and GOA costs. Atri et al. [2000] further improved the heuristics by an algorithm called Incremental-Solve-SOA. Sudarsanam et al. [1997a] studied the offset problem in the presence of an auto-increment/decrement feature that varies from an -1 to $+1$ with k address registers. Rao [1998] and Rao and Pande [1999] extend these approaches beyond offset assignment with memory access sequence reordering (or program reordering) through algebraic transformations on the expression trees. Kandemir et al. [2003] proposed more aggressive access sequence reordering with both intrastatement and interstatement transformations. Program reordering can better utilize the auto-addressing mode by rearranging not only the variables' offsets but also the order of memory access instructions and Kandemir et al.'s solutions prove very useful to optimize an access sequence across a group of statements (say within a basic block). A Genetic Algorithm (GA)-based approach for SOA has been presented in Leupers [1998]. It uses a simulation of natural evolution process. Finally, Leupers [2003] does a comprehensive comparison among several existing algorithms (except program reordering) and proposes a combined algorithm based on tiebreak and incremental-Solve-SOA. He also experimentally found that performances of these heuristics are quite close.

In this article, we propose an optimization framework for compiler-managed code generation based upon the auto-addressing mode on embedded processors that extends the scope of techniques across basic blocks as well as proposing a new direction for access graph simplification through coalescing. Our framework consists of two parts. Firstly, we enhance the effectiveness of offset assignment with a new technique called variable coalescence. Our work is motivated by the observation that the coalescence considerably simplifies the access graph and can effectively reorganize the program values to generate simpler access sequences with high-weighted path covers. Besides, aggressive coalescence can significantly reduce the static and dynamic memory space requirements for both SOA and GOA problems. Variable coalescence can be combined with most previous approaches to further boost the performance. Secondly, to further reduce the AR modification instructions, we add a post/pre-optimization phase to decide whether post- or pre-modification mode should be used for each access. Our post/pre-optimization phase can optimally select post or pre mode after variable offsets are determined. This part also allows intraprocedural optimization of access sequences across the basic blocks.

The remainder of the article is organized as follows: Section 2 briefly introduces background knowledge; Section 3 gives motivating examples; Section 4 talks about assumptions; Section 5 is the overall framework; Section 6 presents coalescence-based offset assignment; Sections 7 is about post/pre-optimization; Section 8 shows results; Section 9 talks about related work; and finally Section 10 concludes the article.

2. BACKGROUND

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

Traditionally, this problem is studied as an *Offset Assignment Problem*. Offset assignment is to assign offsets (memory layout) to frequently referenced stack variables so that the number of address arithmetic instructions can be minimized by using auto-increment/decrement modes of register indirect addressing instructions. Accordingly, *Single Offset Assignment* (SOA) assumes single-AR, while *General Offset Assignment* (GOA) tackles multiple-AR. For example, Figure 2(a) shows the memory layout for 6 variables (the addresses grow upwards) and generated code corresponding to Figure 1(a); we assume one address register AR0, so it is a Simple Offset Assignment (SOA) problem. To simplify discussion, in this example, we assume that variables on the right-side of the assignment statements are loaded into the registers from memory one-by-one from left to right in the evaluation order of the expression. After the evaluation, the result is stored into the left-side variable. Again to simplify the discussion, in this example, all variables are stored in memory (in case they are not, the access graph will show the order of only those accesses that correspond to the memory accesses, i.e., load/stores). For the first instruction c=a+b, after accessing b, that is, ADD *(AR0)-, we use auto-decrement to point AR0 to the memory location of variable c, thus saving one AR modification instruction (e.g., LDAR—set AR to a value, ADAR—add to AR, and SBAR—subtract from AR). Therefore, the problem of maximizing the use of auto-increment/decrement instructions is to find a good memory layout such that a maximal number of consecutively accessed variables are adjacently stored in the memory. To apply offset assignment optimization, we need to find out the *access sequence* first. An access sequence is defined as an ordered linear sequence of memory accesses [Liao et al. 1995, 1996]. For example, in Figure 1(a), we show the access sequence below for the code segment. From the access sequence, we can build an *access graph* [Liao et al. 1996] based upon the access sequence (Figure 1(b)). 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.

After the access graph is constructed, the optimization objective is to find a *Maximum Weight Path Cover* (MWPC) [Liao et al. 1995]. A MWPC is simply a *Path Cover* (PC) with maximal weight. For this problem, path cover is defined slightly differently from that in the graph theory. A path cover is an edge set such that (1) each node can at most appear as an end node on two edges in the edge set; and (2) no cycle can be constructed solely with edges in the edge set. Intuitively, the subgraph constructed using the edges from edge set must consist of one or more linear path(s), therefore can be laid out linearly in memory. Meanwhile, weights covered on the PC are proportional to the number of times auto-addressing mode can be used to access the next variable, while the sum

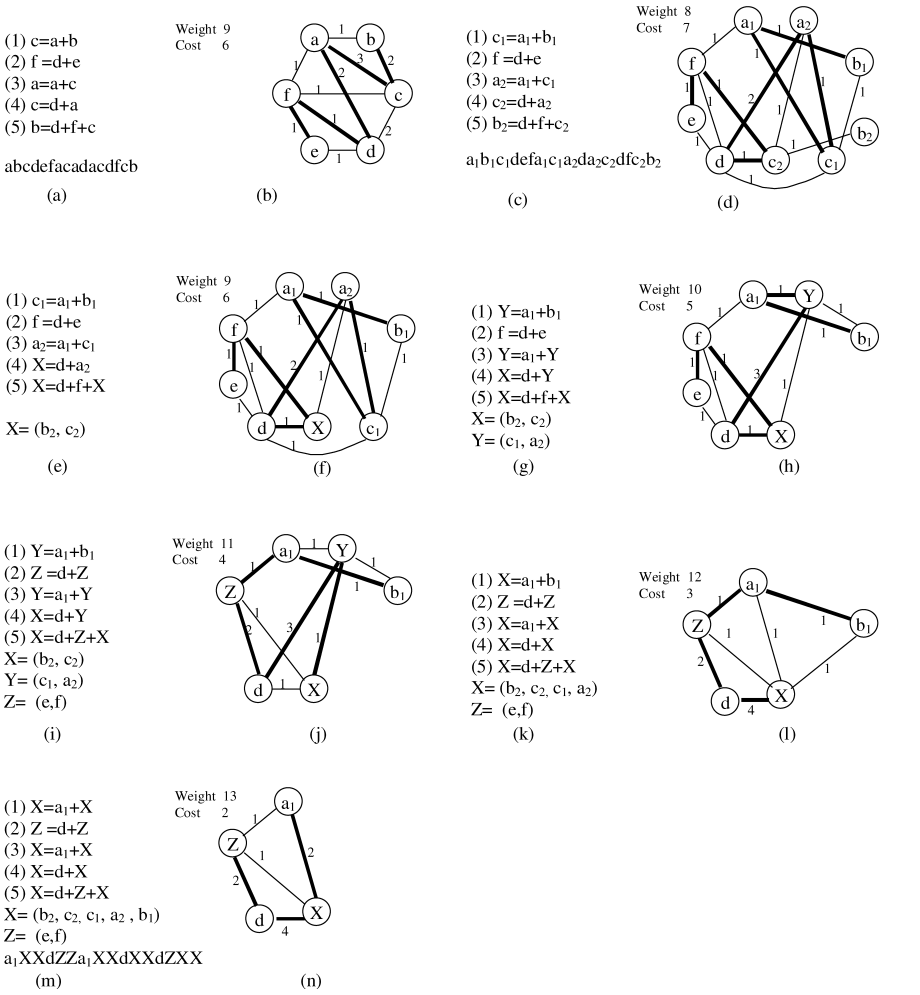
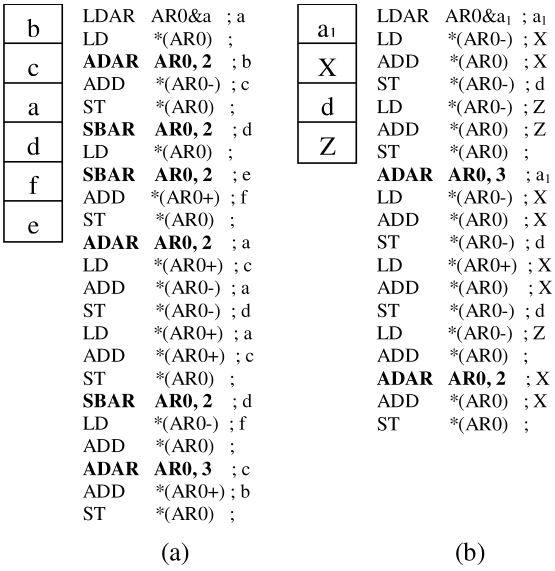


Fig. 1. Motivating example.

of the weights of all edges not on the PC is proportional to the number of times AR modification instructions should be inserted, and this sum is called the *SOA cost* [Liao et al. 1995]. Intuitively, for uncovered edges, AR modification instructions must be inserted and the edge weights now represent how many times these instructions are executed. The thick line in Figure 1(b) shows one of the PC and also a MWPC solution. The weight for the MWPC is 9 and the SOA cost is 6. Earlier approaches [Liao et al. 1995; Leupers et al. 1996; Atri et al. 2000] have shown that the MWPC problem is NP-complete and they developed heuristics to search for the path cover with a weight close to the MWPC.

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. Then for all variables assigned to the same AR, the subgraph is constructed and the problem is solved as SOA. *GOA cost* is actually the sum of



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

Fig. 2. Assembly code (a) before, and (b) after coalescence.

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 that use a particular AR. For example, in Figure 1(a) if we have two address registers AR0 and AR1, {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, an access sequence cannot be simply derived from static information available at compile time. Notice that, on the access graph, the edge weight between two variables indicates 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-addressing mode or to improve the effectiveness of offset assignment. Rao [1998] and Rao and Pande [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-addressing mode.

In this article, 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 quite different, inducing an improved solution superior even to the optimal

MWPC that is obtained without variable coalescence. Furthermore, variable coalescence can be combined with and can improve all previously mentioned offset assignment approaches and it is applicable to both SOA and GOA. Secondly, our post/pre-optimization allows the scope of extending access sequence optimizations across basic blocks (intra-procedurally). In the next section, we will show a few examples to illustrate these optimizations.

3. MOTIVATING EXAMPLES

3.1 Variable Coalescence

In Figure 1, we give an example to illustrate how variable coalescence works and how it can reduce the SOA and GOA cost.

The code segment in Figure 1(a) (taken from Rao and Pande [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 Muchnick [1997, Section 0] through variable renaming. A *web* is a du/ud chain closure of a variable and allows independent allocation of values in memory.

Figure 1(c) shows how we separate each of a, b, c into two variables. 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, c_1, which is live from instructions (1) to (3), constitutes a web, c_1 can be arbitrarily renamed regardless of other parts of the program. Figure 1(c) and Figure 1(d) show the access sequence and access graph after variable separation due to webs. The weight of the MWPC is 1 unit smaller than the one before variable separation. In Figure 1(e) and Figure 1(f), we coalesce b_2 with c_2, that is, we combine these two variables into one variable, putting them into the same memory location. Because the last use of c_2 ends before the definition of b_2, they can be safely coalesced as one variable X. Their edges are coalesced accordingly as shown in Figure 1(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 1(g) to Figure 1(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 2(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. Assume that 2 address registers AR0 and AR1 are available, for the code in Figure 1(m), we can simply assign two variables to each of them, such as {X, a_1} to AR0, {Z, d} to AR1. The access sequence for {X, a_1} as derived from the whole access sequence in Figure 1(m) is a_1XXa_1XXXXXX, thus the access graph has only one edge with

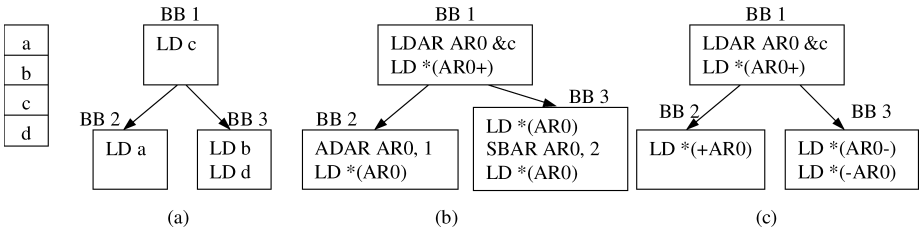


Fig. 3. Example for post/pre-optimization (a) original code and offsets (b) without post/pre-optimization (c) with post/pre-optimization.

weight 3, which is on the MWPC. Similarly, for {Z, d}, the solution is also optimal (SOA cost of 0). We will show in Section 6.6 that coalescence can often generate an optimal solution for GOA.

Figure 1(b) already shows the optimal MWPC solution for the case without coalescence, and therefore no heuristic can reduce the cost below 6 without variable coalescence. As far as program reordering is concerned, it is also applicable to the code after coalescence as shown in Figure 1(m); thus program reordering can be used to get more improvement after the 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.

3.2 Post/Pre-Optimization

We now motivate the second part of this work: post/pre-optimizations. As mentioned previously, both post- and pre-modifications are supported for some embedded processors. However, current research on offset assignment does not consider them together. Moreover, the most significant limitation of the current approaches is that they work only within a basic block, which severely limits their applicability in practical settings. The main reason behind this limitation is that the boundary conditions of address registers on the incoming edges of a basic block need to be identical. In Figure 3(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 upon the variable offsets, we can generate the final code as in Figure 3(b), where auto-addressing mode is only used once. Three LDAR instructions have to be generated to set the address register AR0. However, in Figure 3(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-modifications, we can avoid any AR modification instructions. 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 generated by the offset assignment algorithms. In short,

through the appropriate use of post/pre-modification, it was possible to extend the technique across basic blocks without incurring the use of ADAR and SBAR instructions.

4. ASSUMPTIONS

Most of the basic assumptions behind this are followed from previous work [Bartley 1992; Liao et al. 1995, 1996; Rao 1998; Rao and Pande 1999; Leupers and Marwedel 1996]. We list some specific ones as follows.

(1) This article only considers auto-increment/decrement addressing with stride 1, which means every time, the address register can only be increased or decreased by 1. Auto-addressing with stride 1 is actually the most widely supported auto-addressing mode on state-of-the-art embedded architectures; the techniques described can be extended to take advantage of nonunit increment/decrements.

(2) It is not safe to convert all the address register operations into auto-increment/decrement 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 -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 2) operations.

(3) In addition, array-index-based optimizations have comprised an active area of research and there are techniques to analyze array-indexed memory accesses, especially in loops [Ottoni et al. 2001; Araujo et al. 1996; Gebotys 1997; Leupers and Yang 1998; Zhang and Yang 2003]. However, such research 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.

5. OVERALL FRAMEWORK

Figure 4 shows the overall framework for our optimizations. To avoid interfering with a good register allocator and other optimizations before register allocation, our optimizing pass comes after virtual register allocation. Also, for user-defined variables and temporaries, *webs* are built to achieve value separation. Value separation is extremely important as the compiler normally generates lots of temporaries that are reused repeatedly. Figure 4(a) shows the optimization flowchart for single-AR and Figure 4(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. Obviously, variable coalescence can reduce the number of variables in addition to lowering the SOA or GOA cost: OpCost targets the minimization of SOA or GOA cost, while 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 Graphs (AGs) and Interference Graphs (IGs). These two graphs are necessary to guide the coalescence and the offset assignment process.

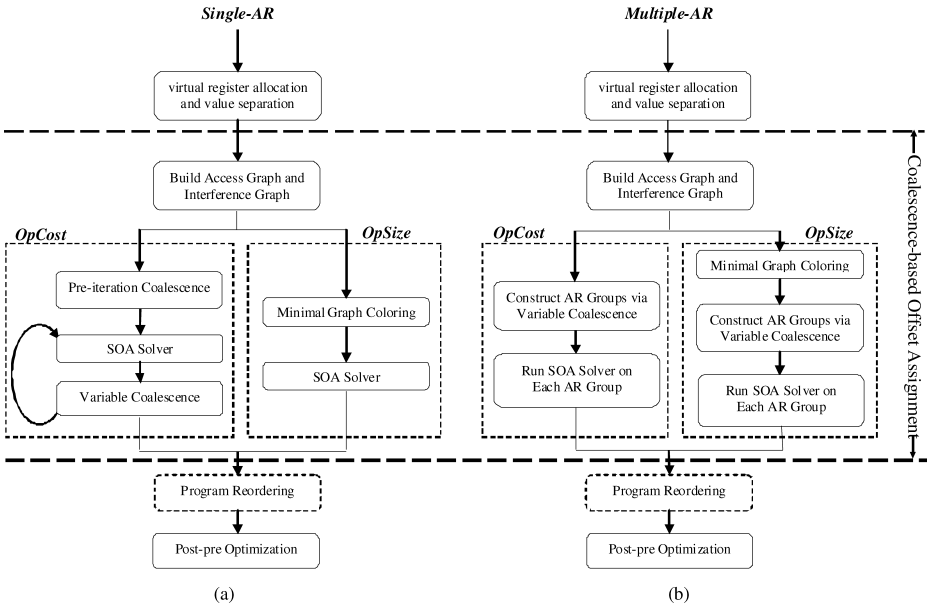


Fig. 4. Optimization framework.

In Figure 4(a), both OpCost and OpSize invoke an SOA solver, which comes from one of the previous offset assignment algorithms without variable coalescence. The SOA solver only assigns offsets for variables and attempts to minimize the SOA cost. For the OpCost algorithm, a heuristic approach is chosen to iterate over MWPC for searching the most beneficial coalescence (see Section 6.5.1 for more details). 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 4(b), for generating the GOA solution using the multiple ARs, the algorithm classifies variables into several AR groups, such that 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. Minimizing the number of nodes frequently leads to optimal solutions. In case the optimal solution is not obtained after the graph coloring, we apply the coalescence algorithm as in OpCost.

After variable coalescence and offset assignment, a program reordering phase can be optionally invoked. Inside one basic block, for all memory access instructions being optimized, a dependency DAG is built. Then, we use the commute-3 [Rao 1998; Rao and Pande 1999], that is, the exhaustive search to find out the optimal reordering of the expressions and the instructions without

violating the dependencies. To reduce exhaustive search overhead for big basic blocks, we reorder for every 15 memory access instructions. Our results show that program reordering can greatly boost performance. Finally, we perform post/pre-optimization if both post- and pre-modification modes are supported generating an efficient intraprocedural solution across basic blocks.

Clearly, our framework incorporates more optimizations than solely assigning offsets for variables. The phases before program reordering will be explained in Section 6. We call this part "coalescence-based offset assignment," which performs variable coalescence together with offset assignment. We will make use of a SOA solver from early "offset assignment-only" approaches. Post/pre-optimization is discussed in Section 7.

6. COALESCENCE-BASE OFFSET ASSIGNMENT

6.1 Variable Renaming, Webs and Variable Separation

We first perform a simple alias analysis [Aho et al. 1986] to determine the variables that might be referenced via pointers. To guarantee safety of code generation, if a pointer dereference can point to multiple variables at a given program point, then it must be excluded from the auto-addressing mode optimization. The remaining r-value references are included in the optimization passes. In order to separate memory references, which can be independently considered for allocation, we then rename variables and construct *webs* (as in Figure 1(c) and Figure 1(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, that is, one must bind all the definitions and use them within a web to a single memory location. In this manner, we are able to achieve effective value separation at different program points. The compiler normally reuses temporaries repeatedly. Decoupling these variables through renaming gives us more freedom to coalesce them in a proper way to maximize the profit of offset assignment optimizations. This is shown to be effective in the example from the previous section. If we do not separate the variables into the ones in Figure 1(b), the follow-up step cannot solve SOA and GOA effectively.

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

6.2 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 determine the overlap 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.

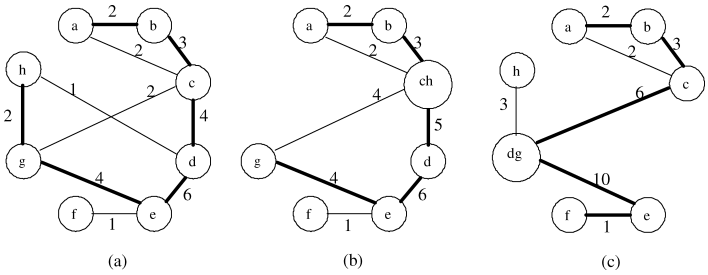


Fig. 5. Profitability of variable coalescence.

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 vice versa. During actual implementation, we only need one of them.

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.

6.3 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 the access sequence through judicious choice of coalescing is, however, a nontrivial 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 more via graph topology than the edge weights as far as MWPC is concerned. This is due to the fact that in 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 MWPC as much as reduction in node degrees, which is a function of graph topology more than edge weights. Coalescence has been tackled in the register allocation setting as well; however, the coalescing in register allocation should not be clique-forming whereas coalescence in our setting should be MWPC-assisting. This is the reason why one can not use or adapt coalescence techniques developed in the register allocation setting to our problem.

Figure 5(a) shows the original access graph and the current status of MWPC, that is, a-b-c-d-e-g-h and f with total weight 21. If the coalescence graph permits nodes c and h to be coalesced, we can coalesce the two nodes and get an MWPC

(a-b-ch-d-e-g and f) in Figure 5(b), where the weight is 20. After coalescence, the MWPC is worse. This is because node c already has 4 neighbors. Adding more neighbors from h may actually hurt the solution. In contrast, in Figure 5(c), we coalesce nodes d and g. The MWPC is a-b-c-dg-e-f and h with weight 22. This example tells us that coalescence cannot be done arbitrarily without considering the topology of IG and AG.

6.4 Problem Formulation

The objective of variable-coalescence-based offset assignment 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.

Definition (*Coalesced Node* (*C-Node*)). A C-node is a set of live ranges (webs) in the AG or 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.

Definition (*Coalesced Edge* (*C-Edge*)). The C-edge is an edge set defined for a pair of C-nodes. A C-edge $<c_1,c_2>$ between two C-nodes c_1 and c_2 is a set defined as:

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

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

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

Definition (*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.

Definition (*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 indicates that the two C-nodes have interfering live ranges, therefore cannot be coalesced.

Definition (*Coalesced Path Cover* (*C-PC*)). On a C-AG, a C-PC consists of a sequence of C-nodes $c_1, c_2, \ldots c_k$, where $< c_i, c_{i+1} >$ is a C-edge between C-nodes 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.

Definition (*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 eliminated from the graph.

Definition (*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.

Definition (*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.

Definition (*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.

The algorithm starts with the original AG, where each node is labeled as a C-node. Both C-AG and C-IG are updated when we coalesce more and more C-nodes. We first show that finding the best solution through coalescing (called C-MWPC) is a hard problem. Next we attempt two heuristic approaches.

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 uncoalesced live range after value separation and C-PC is equivalent to PC. A fully connected interference graph is made possible when all live ranges interfere with each other. Thus, the C-MWPC problem is NP-complete. □

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

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

6.5 Coalescence-Based Offset Assignment for Single-AR

Since the C-MWPC problem is NP-complete, heuristic algorithms must be taken to seek solutions in a reasonable amount of time. As mentioned in Section 5, two types of heuristics can be introduced to achieve different objectives: either to reduce the cost on the access graph or to get a smaller memory footprint.

6.5.1 *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. Every time two C-nodes are selected for coalescing and the SOA solver (we use the tiebreak SOA algorithm [Leupers and Marwedel 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 suboptimal C-PC.

Pre-iteration Coalescence Rules. The pre-iteration rules are applied before we do iterative coalescing. Applying these rules is guaranteed not to worsen the SOA cost. All these rules are with respect to the Access Graph (AG). Note that we can only coalesce a pair of C-nodes if the C-nodes do not have an interference edge between them.

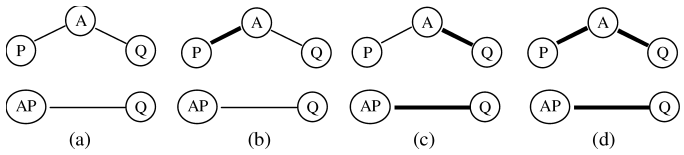


Fig. 6. Profitability of Rule 3.

Rule 1. Coalesce all degree-0 C-nodes with any other C-node. This does 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 6. 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 which is found without performing coalescence of A with P. Figure 6(a) to Figure 6(d) show four cases depending on whether <A,P> and <A,Q> are on the C-PC. In Figure 6(a), none of the 2 C-edges is a part of the C-PC, so the coalescence will add *Weight(<A,P>)* to the overall solution and will be beneficial. In Figure 6(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 add *Weight(<A,P>)*. If both of them are on the C-PC (Figure 6(d)), the cost is unchanged. Therefore, in all cases, 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.

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 maximal saving and coalesce them. The basic idea is to use the current C-PC offset assignment to estimate the saving once the 2 C-nodes are coalesced. For example, Figure 7(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. Thus, 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 8, we illustrate 3 different cases to coalesce J with I. Figure 8(a) shows 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, that is, C-edges <C,J> and <P,J> if they exist.

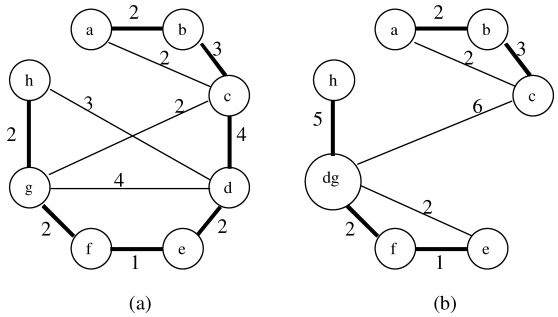


Fig. 7. Cases to calculate the savings.

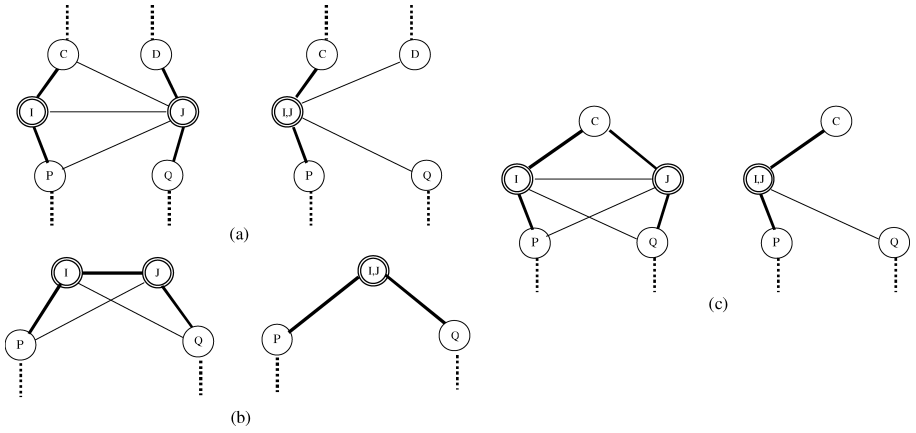


Fig. 8. Coalescence cases based on previous C-PC.

We lose:

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

Figure 8(b) is a special case when 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 8(c), I and J have a common neighbor C. Then, the weight of the C-edge <C,J> is not a loss. The saving for J coalesced to I is different from the saving for I coalesced to J. We take the bigger one as the saving for their coalescence.

Tiebreak for the Same Saving. If two or more pairs of C-nodes have the same coalescence saving, we apply a *tiebreak* rule. This tiebreak rule is similar to the one used in Leupers and Marwedel [1996] to select equal-weight edges during the construction of path covers. In our case, for each coalescence candidate $\{c_1, c_2\}$, the tiebreak weight T is calculated as

$$T = \Sigma weight\,(all\ C\text{-}edges\ joined\ to\ c_1\ and\ /\ or\ c_2).$$

A smaller T has higher priority, as explained in Leupers and Marwedel [1996]. 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.

```
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. }
```

Fig. 9. Coalescence-based offset assignment for single-AR.

Coalescence Algorithm. The complete coalescence algorithm is shown in Figure 9. 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 there is no more calculated saving to coalesce. 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 one of the SOA solvers (we implement Liao et al.'s SOA algorithm [Liao et al. 1995] enhanced with tiebreak [Leupers and Marwedel 1996]) 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 rerunning the SOA solver, we may get a different C-PC which may have an even lower SOA cost.

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 be less likely increase the node degree density. In this manner, we try to drive coalescence via a limited graph topology property, that is, the

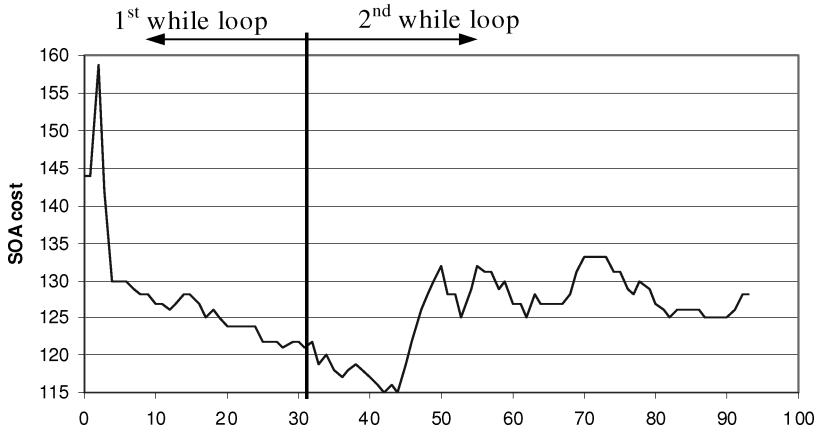


Fig. 10. SOA cost fluctuation along with iterations for procedure "findcost."

node degree; more complicated solutions are possible but may not yield much benefit due to the complexity of the problem.

To illustrate how SOA cost fluctuates during the two while loops, we show the SOA cost versus number of iterations in Figure 10. 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 while 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.

6.5.2 *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 phases without iteration. The first phase attempts to use a minimal coloring algorithm to color the interference graph. Nodes with the same color are coalesced on both AG and IG. The following lemma says minimal coloring of the IG is equivalent to achieving the 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 converted 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, there

is, no edge 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 Chaitin-style register allocation [Chaitin et al. 1981; Chaitin 1982]. In Chaitin-style register allocation, when removing nodes from the interference graph and pushing to the stack, we always remove the one with the lowest degree. Since coloring is performed on the interference graph, nodes with the same color are guaranteed to be coalesceable. After the coloring phase, an existing SOA solver (no coalescence) is applied on the resulting C-AG and C-IG to assign offsets for coalesced nodes.

Aggressive coalescing might lead to higher SOA cost, 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 memory size reduction.

6.6 Coalescence-Based Offset Assignment for Multiple-AR

The multiple-AR model allows more than one AR to utilize the auto-addressing mode. With the trend in embedded processor design to increase the number of ARs, the multiple-AR model is playing a more and more important role in optimizing compilers to generate efficient code. In Motorola DSP56300, one of the 8 ARs is used as stack pointer, and another one is used as the base address register. Other ARs can be allocated for other purposes to hold variables, as address registers are one of the register classes during register allocation. 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 et al. 1995; Leupers and Marwedel 1996] all attempt to separate variables into several groups, 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 with 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 Figure 4(b), OpSize has an additional phase to minimally color the interference graph. We will discuss these phases in the following sections.

Figure 11 shows the algorithm called Coalesce_OA_Multiple_AR. This algorithm is invoked by both OpCode and OpSize. The only difference is, for OpSize, a graph coloring algorithm first coalesces nodes on the graphs aggressively, then Coalesce_OA_Multiple_AR works afterwards if optimal solution cannot be obtained immediately (Section 6.6.2).

6.6.1 *Coalescence Algorithm for Multiple-AR.* Initially, the algorithms store 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

```
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 ∈ G1∩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. }
```

Fig. 11. Coalescing algorithm for multiple-AR.

mini_set. It attempts to put each node to each AR group and calculates the extra cost that will be incurred by calling Coalesce_OA_Single_AR (Figure 9) on that AR group. We should find a (v, i) pair so that assigning node v to AR group G_i 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 tiebreak scheme. It shares some features with the tiebreak GOA algorithm in Leupers and Marwedel [1996]. We calculate two values for tiebreak. Value *w1* is calculated for each entry in *mini_set*. If v is selected for G_i, we sum all the edges on 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 *w1*. If this still cannot break all ties, we try another value *w2*. *w2* is calculated for each node v as the number of neighbors that are still on *IG*.

Larger *w2* means more interference with the nodes that have not been added to one of the AR groups. We prefer a smaller $w2$, which means more nodes on the *IG* later can be coalesced with v. If both tiebreaks 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 9) for each AR group. It returns a node to C-node mapping and a C-node to AR group number mapping.

6.6.2 *OpSize Algorithm for Multiple-AR.* Since aggressive variable coalescence can greatly reduce the number of C-nodes on the graph, in many cases, we can actually get the optimal solution. The following lemmas specify when the optimal can be reached.

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, that is, GOA cost equal to zero by assigning no more than 2 C-nodes to each address register.*

PROOF. Following the previous lemma, the SOA problem for each address register is optimal, so there is 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 (or CG) 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, that is, GOA cost equal to zero 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 11) 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. It is noteworthy that the graphs after graph coloring are still coalesceable, because our graph coloring algorithm is very simple and nonoptimal.

To quantify the number of times we can get optimal solutions with a certain number of address registers, we did experiments on the 10 benchmark programs. All data are collected for local variables. We count the number of procedures that can be optimally solved in cases of: (1) after IG coloring; (2) after both coloring and Coalesce_OA_Multiple_AR, that is, the final number of optimal solutions. As mentioned in the previous section, Corollary 1 only gives a sufficient condition, that is, even if an AG has more than two nodes,

Table I. Percentage of Optimal Solutions for Multiple-AR

#AR	epic	gsm	g721	Mpegd	mpege	bzip2	gzip	mcf	twolf	vpr	ave
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

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 gotten from IG coloring.

Table I shows the percentage of optimal solutions for different numbers 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, in 84.9% procedures IG can be colored by 4 colors. But with 3 ARs, 90.57% of the procedures are 6-colorable. Rows 3 and 5 are the final number of optimal solutions. We observe a higher percentage for each benchmark. 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.

7. POST/PRE-OPTIMIZATION

After discussing coaleascing we now move to the second optimization: post/pre-optimization. Post/pre-optimization needs to decide whether post- or pre-addressing mode should be used for each memory access instruction (in this section, we implicitly restrict "memory access instructions" to those accessing the variables on the access graph) so as to minimize the number of AR modification instructions. This optimization comes after offsets are assigned to the variable. The biggest value of this optimization is that it allows the scope to be extended to intraprocedural one. The auto-addressing optimizations described in the literature are typically limited to basic blocks because the cost of AR modification instructions to be inserted at the basic block boundaries takes away the benefit of optimizations. In this work, develop an effective algorithm to minimize the AR modification instructions through the use of pre- and post-modification modes. Using the assigned offsets, we first find out the offset difference between the adjacent memory accesses. Given that attempting all possibilities of post/pre mode and AR modification insertion could 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.

Offset Distance. We now discuss the technique in details. 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

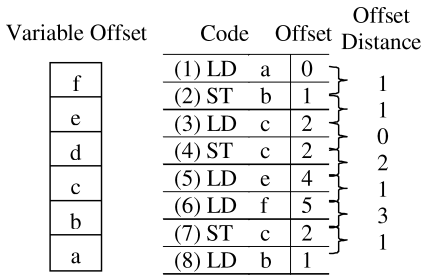


Fig. 12. Example for offset distance.

instructions. In Figure 12, we show the variable offsets, selected code segment with the memory accesses, and the 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 should do pre-modification to change the AR before its memory access.

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 12 does not perform post-modification, that is, post-increment, the 2nd instruction ST b must do pre-increment to avoid an extra AR modification instruction. 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-modification is needed for the 3rd instruction and no pre-modification is needed for the 4th instruction. On the other hand, the 3rd instruction might use pre-modification depending on the other neighbor, but this is independent of the addressing mode of 4th instruction. Similarly, the 4th instruction might use post-modification, 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 AR modification instruction is unavoidable to modify the AR between these two instructions. After the AR modification instruction 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 13. Up until 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, such as the example in Figure 3, when one basic block has multiple predecessors and successors; we will discuss such constraints later.

Basic Block Splitting and Canonical Form. Following the identification of offset distance, in this section, we will discuss how to split basic blocks and transform the CFG to *canonical form* as defined next.

Definition (*Canonical Form, Canonical CFG*). If a CFG has offset distance equal to 1 inside all basic blocks, it is in canonical form. The CFG is called a canonical CFG.

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

+This means no post or pre mode is required.
*An AR modification instruction is required.

Fig. 13. Addressing modes between two adjacent memory access instructions.

Canonical form facilitates the formulation of post/pre-optimization. Based on the table in Figure 13, we can easily transform a CFG to its canonical form through basic block splitting. Furthermore, basic block splitting can greatly reduce the problem complexity. 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 further tackle the offset distance 2 case by removing it. The motivation behind such splitting is that when offset distance is 0, no AR modification is needed and when it is greater than 2, loading the AR with a new address is inevitable.

LEMMA 6. *Inside one basic block, if two consecutive memory access instructions have offset distance* 0 *or* >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.*

PROOF. Notice that after splitting a basic block at a program point between the two instructions, the 1st instruction becomes the last memory access instruction in its basic block Likewise, the 2nd instruction becomes the first memory access instruction in that basic block. In the first case, assume the offset distance is 0 between the two instructions, the optimal solution should not require post-modification for the 1st instruction, nor should the pre-modification for the 2nd instruction be needed. This is also enforced on the split CFG. For the second case, that is, the offset distance is greater than 2, one and only one AR modification instruction must be inserted in the optimal solution, and no post-modification is needed of the 1st instruction, since the AR modification instruction is enough to change the AR. Also, no pre-modification is necessary for the 2nd instruction. Thus, after CFG splitting the optimal solution for the new CFG is at least as good as the original optimal. □

The main reason behind basic block splitting discussed earlier is that it allows *a separation* of the two basic blocks and that allows them to be separately fed to the solver for optimization purposes; in other words, they become decoupled as far as the solution space is considered. Based on Lemma 6, after step 1, all basic blocks only have offset distance 1 or 2. In the second step, we

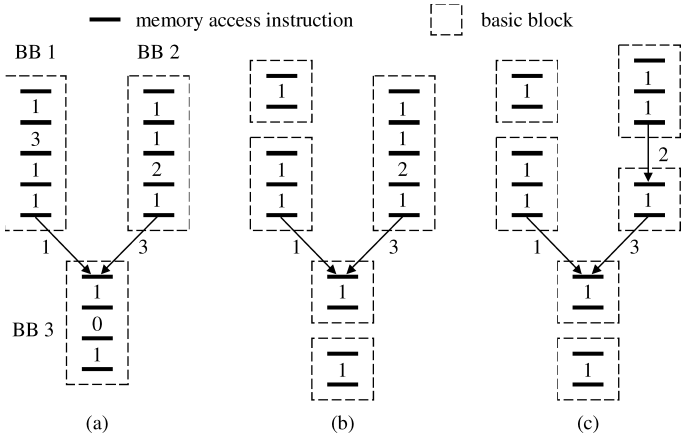


Fig. 14. Example for canonical form transformation (a) original code (b) after step 1 (c) after step 2.

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 should be considered together and that is why for solution purposes they are still coupled as shown by the edge. Figure 14 shows an example with three basic blocks. After step 1, in Figure 14(b), BB1 and BB3 are split, and after step 2 in Figure 14(c), BB2 is split into two basic blocks at the point with offset distance 2. The two new basic blocks in Figure 14(c) are still coupled. Notice that the offset distance between two basic blocks are not tackled during basic block splitting, but will be considered when we start to solve the canonical CFG.

After splitting and decoupling as discussed before, the resulting CFG is likely to be disconnected and contains many small, connected components that can be separately optimized, reducing the problem complexity. We can first split basic blocks, then solve optimally. To get the solution for the original CFG, basic blocks are reconnected at the split points and one AR modification instruction is added at each point with offset distance greater than 2. For the example in Figure 14(c), the CFG is in canonical form and it is now split into 3 connected components. After solving the canonical CFG optimally, we need to add the cost by 1 AR modification instruction, since the splitting in BB 1 is at offset distance 3, therefore we should make up that AR modification instruction.

Solving the Canonical CFG with Branch and Bound. We now discuss how to find a solution to canonical CFG. 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. First, the following lemma tells us the optimal solution can be in the form that AR modification instructions are all inserted at the basic block entry and exit points.

LEMMA 7. *On a canonical CFG, AR modification instructions only need to be inserted at the basic block entry and exit points.*

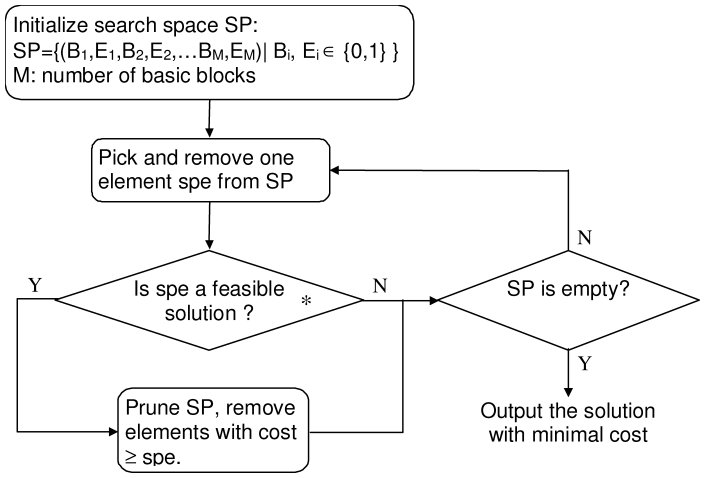


Fig. 15. Flow graph for solving the canonical CFG.

PROOF. Suppose an AR modification instruction is inserted between two memory access instructions within a basic block, we now prove it can be moved below the 2nd memory access instruction. It is easy to observe, the 1st instruction does not perform post-modification and the 2nd instruction has no pre-modification, otherwise the AR modification instruction is not needed at all. If the 2nd memory access does not have post-modification, we can change it to pre-modification and the AR modification instruction is saved, otherwise we can move the AR modification instruction after the 2nd memory access instruction and do a $+1$ or -1 operation on the AR. Now, we can continue to move down AR modification instructions until they are all at the end of the basic block and merge them to one instruction. □

With Lemma 7, we can reduce the number of insertion points greatly. For a connected component on the canonical CFG with M basic block, the search space is 2^{2M}, that is, 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 15. The search space *SP* is initialized to contain all of the 2^{2M} 2M-bit vectors. Every time, one element *spe* is selected from *SP* and checked whether 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, that is, 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.

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 a proper value before reaching a memory access instruction. It either points to the variable being accessed or can be pre-incremented or -decremented to point to that variable. First, we give 3 definitions.

Definition (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. Notice that EO must be a unique value.

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

Definition (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 obeying the following restrictions.

Restrictoin 1. If $B_i = 0$, for basic block i's predecessors p1, p2... pk, we have $EO_{p1} = EO_{p2} = EO_{pk} \equiv BO_i \in \{BVO_i - 1, BVO_i, BVO_i + 1\}$.

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

Restrictoin 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, that is, $B_i = 0$. Also, the position pointed by the AR should be at most 1 slot away from the first memory access instruction's offset, that is, BVO_i, such that pre-modification 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, when leaving the basic block, the value of AR should be one of {EVO_i-1, EVO_i, $EVO_i + 1$}. Finally, Restriction 3 says either the first memory access instruction does pre-modification or the last memory access instruction does post-modification or none of them, but not both. This restriction is illustrated in Figure 16(a). If the 1st memory access needs pre-modification, the last memory access must be inhibited from post-modification to avoid an extra AR modification instruction. Similarly, if the last memory access does post-modification, then the 1st one cannot afford pre-modification.

As an example, Figure 16(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, that is, 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

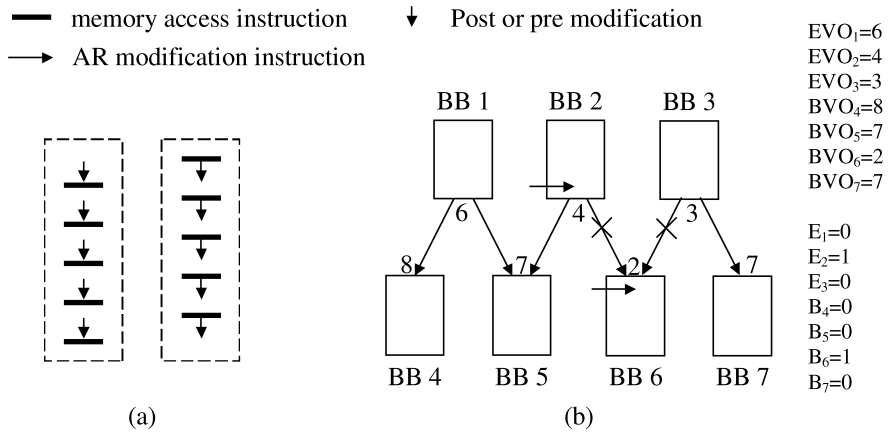


Fig. 16. Illustrations for feasibility checking.

are equal based on Restriction 1. Notice that we can build a transitive closure through the predecessor/successor relationship. In this 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, that is, $6 \leq BO_7 \leq 8$, $2 \leq EO_3 \leq 4$. However, these two ranges have no overlap. Eventually, our feasibility checking concludes that this insertion scheme is infeasible.

8. PERFORMANCE EVALUATIONS

8.1 Experimental Environment and Benchmarks

Our environment is the Motorola 56300 processor toolset including a cycle-accurate simulator, sim56300, and a retargeted GNU C compiler [Stallman 2002a], which comes with standard header and library files. Our optimization is implemented at RTL level; GCC's IR, after the "reload pass" of GCC, and before the assembly is produced, thus we can capture all the temporaries and spill code generated by the compiler.

As mentioned earlier, among the eight address registers available on Motorola 56300, one is dedicated for stack pointer and another as the base address pointer. Therefore, we can use up to six ARs for auto-addressing mode. All simulations and compilations take place on a Pentium III 1.0 GHz machine.

A total of 10 benchmarks were used for evaluation. Among them, 5 are from Mediabench, 2 from MiBench, and 3 from Spec2000int. These benchmarks represent a combination of real DSP-related applications. We use access graphs

Table II. Statistics for the Benchmarks

	Code Size	SOA Cost	Stack Slot Number
Adpcm	281k	121	216
Epic	260k	747	1287
G721d	107k	90	297
Gsm	295k	862	1686
Jpeg	338k	5125	2468
Mcf	179k	256	756
Mpeg2d	403k	891	2307
Mpeg2e	499k	1299	3606
Twolf	1471k	2571	6687
Vpr	917k	2970	7659
Average	475k	1493.2	2696.9

built using profile information for all results, that is, access graphs are based on information gathered in test runs.

Table II shows some properties for the benchmarks. The second column is the code size in bytes. The third column shows the baseline SOA cost (as mentioned in Section 6.5.1, we use the incremental tiebreak SOA (INC-TB SOA) algorithm [Leupers 2003]). We will compare our approaches against it. The fourth column is the initial stack slot numbers without coalescence. As discussed earlier, SOA costs are the most important metric to evaluate the effectiveness of this optimization since they represent dynamic counts of AR modification instructions. Since the problem is one of minimization of AR modification instructions, SOA cost reduction gives a measure of effectiveness of the optimization technique.

Next, we present results which can be categorized into 3 types. Section 8.2 and 8.3 are for SOA and GOA, respectively. We will show comparison along 3 axes, that is, cost, stack size, and cycle reduction. And finally, Section 8.4 compares compilation time for each phase.

8.2 Results for Coalescence-Based Offset Assignment with Single-AR

We first look at how coalescence-based offset assignment performs when only one AR is considered. Two optimizations that is, either OpCost or OpSize, are compared together with the original code (without any optimization for auto-addressing mode) and baseline INC-TB SOA [Leupers 2003] algorithm.

In Figure 17, all SOA costs are normalized to the original ones. The SOA cost for INC-TB SOA is much smaller than the original one for all benchmarks, however, coalescence-based approaches further improve that. On average, INC-TB reduces the SOA cost by 41%, while OpCost and OpSize achieve 72% and 65% SOA cost reduction, respectively. It is not surprising that the OpCost algorithm is more effective in cutting down the SOA cost than OpSize, since during coalescence iterations, only the one with minimal cost is kept. Therefore the final solution given by the OpCost algorithm is probably not the one resulting from the most aggressive coalescence. Also, we observe from Figure 17 that normally if the INC-TB reduces more cost, the OpCost and OpSize achieve more reduction as well. This can be explained as follows: since the INC-TB is involved in

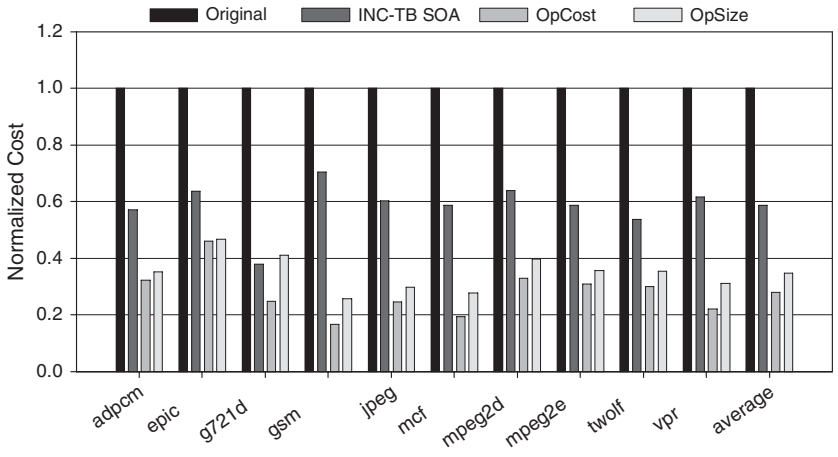


Fig. 17. SOA cost comparison.

the OpCost and OpSize algorithms, its effectiveness can partly influence the other two algorithms.

In Figure 18(a) and Figure 18(b), we look at the SOA cost when all optimization phases are invoked. They show how program reordering and post/pre-optimization incrementally reduce the SOA cost. Figure 18(a) evaluates OpCost-based algorithms. Again, the numbers are normalized to the original cost. We observe both Program Reordering (PR) and Post/Pre (Post/Pre)-optimization further cut down the SOA cost and the latter seems more effective. It is perhaps due to the optimality of our algorithm and the prevalence of cross BB opportunities. Combining both phases achieves a very low cost. On average, program reordering (shown as PR in the figures) reduces the cost by 5% over the OpCost-only approach, while post/pre-optimization adds an additional 8% reduction. Combing them together, the reduction is about 13% (normalized cost is 0.13).

Figure 18(b) shows the same set of data for OpSize. A similar cost reduction trend can be noticed for OpSize except that we see a higher percentage of reduction. It might come from the extra chances left out by the OpSize algorithm as against OpCost. On average, PR and Post/Pre get extra cost reduction of 7% and 15%, respectively. Enabling both contributes 20% more reduction over the OpSize-only approach (normalized cost is 0.15). We then look at the stack size reduction as shown in Figure 19. INC-TB does not change the stack size, because no coalescence is engaged. We can see both algorithms shrink the stack size significantly from the original benchmarks. The average stack size reduction is 70% and 74% for OpCost and OpSize, respectively. This is probably due to the large number of temporaries generated by the GCC compiler. 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 to a maximum, then invoke the SOA solver, leading to a smaller footprint on the stack than OpCost. However, the difference is not very

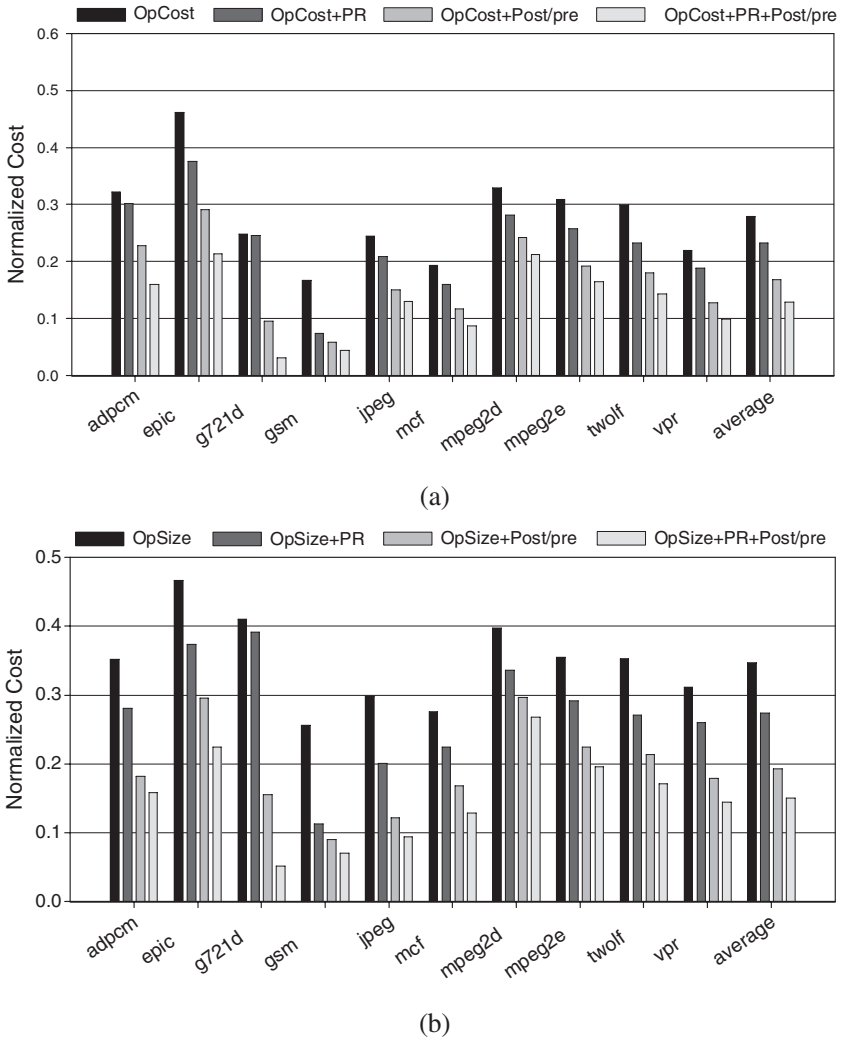


Fig. 18. SOA cost comparison—all phases included (a) OpCost (b) OpSize.

significant between OpCost and OpSize, showing that coalescence also contributes heavily in cutting down the SOA cost.

Next, we show the amount of cycle reduction for OpCost and OpSize combined individually with PR and/or post/pre-optimization. In Figure 20(a), we found that the cycle reduction is diversified across benchmarks. On average, INC-TB is able to reduce cycle counts by only 2.647%, while OpCost boosts it to 3.84%. The other 3 cases, that is, PR only, Post/Pre, only, and PR+Post/Pre, achieve 4.58%, 5.17%, and 5.59% cycle reduction respectively. This largely corresponds to the cost reduction results presented early on. Some benchmarks like vpr, mpeg are insensitive to our optimization partially due to the lack of variables that can be tackled. In Figure 20(b), the same set of comparisons are made for OpSize. We notice that the numbers are actually quite close given

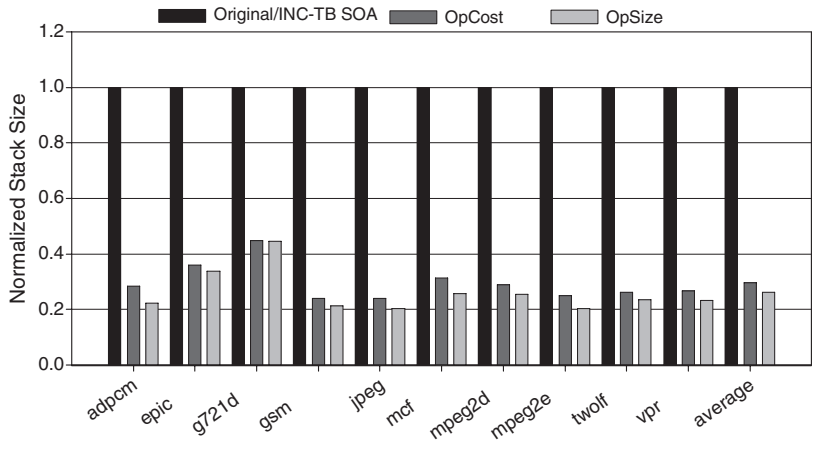


Fig. 19. Stack size reduction.

the fact that cost calculation is only among part of the memory access instructions, making the difference smaller than what we observe for the SOA cost. On average, OpSize gets 3.79% cycle reduction, while PR-only, Post/Pre-only and PR+Post/Pre achieve 4.16%, 4.88% and 5.3%, respectively. Although, these numbers are not dramatic, it must be noted that they are significant given this is an optimization that is applied after other significant optimization phases are already applied (such as register allocation, etc.). In other words, the baseline for this optimization is already quite optimized.

We also notice another research on coalescence-based offset assignment by Ottoni et al. [2003], which came later than our conference publication [Zhuang et al. 2003], and a journal publication [Ottoni et al. 2006] with extension to GOA. A comparison with some of the algorithms were made in their journal publication [Ottoni et al. 2006], therefore we can cross-compare the two approaches. First, our conference publication only contains the OpCost SOA algorithm. The results seem to match [Ottoni et al. 2006], that is, the SOA cost and stack reduction are almost on par with Ottoni et al. [2006]. However, with the addition of OpSize, program reordering and post/pre-optimization, the SOA cost can be reduced by an additional 13–20%, and cycles can be reduced by an additional 3–4%. Therefore, our results are clearly superior.

8.3 Results for Coalescence-Based Offset Assignment with Multiple-AR

With mutliple-AR, we expect better performance results; however, actually investing more ARs is not always rewarding. In this section, 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-addressing, less address registers will be available for other purposes like heap accesses.

In Figure 21, we compare the GOA cost along two dimensions. Again GOA costs measure the dynamic count of AR modification instructions in the presence of multiple ARs and thus, are a direct measure of overhead that is being

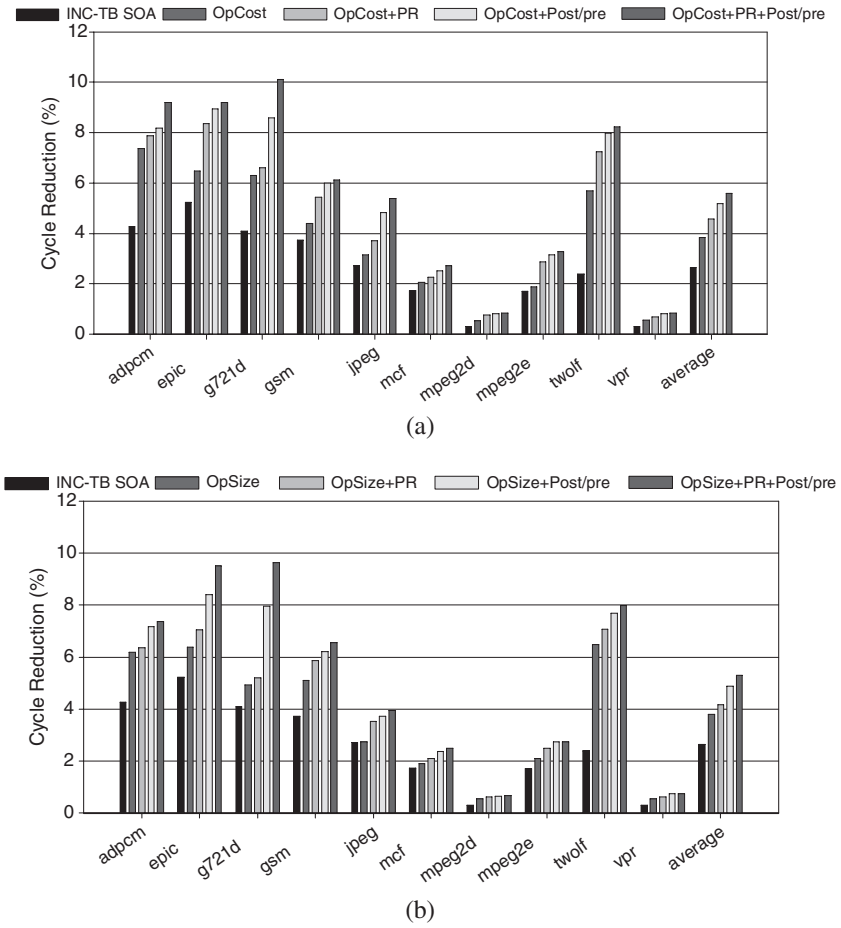


Fig. 20. SOA cycle reduction comparison (a) with OpCost (b) with OpSize.

tackled by the optimization. We vary the number of address registers from 2 to 4 and three algorithms are evaluated. The baseline algorithm is Leupers and Marwedel's [1996] GOA. It is denoted as L&M SOA. The other two are OpCost and OpSize. Therefore, we show 9 bars for each benchmark. For each benchmark, values are normalized to the first bar, that is, 2-AR L&M GOA. In most cases, we observe lower cost when 3 ARs are used instead of 2 ARs. However, using 4 ARs most likely worsens the cost. For g721d, the GOA cost for 4-AR is even higher than 2-AR. Not shown in the figure, GOA cost further increases for more than 4 ARs. On the other dimension, not surprisingly OpCost is better than OpSize for every setting of AR numbers, which confirms the superiority of OpCost over OpSize in reducing the cost. However, the coalescence-based offset assignment is still very successful in reducing the cost over the L&M GOA. On average, with 2-AR, OpCost reduces the cost by 49% over the L&M GOA; and OpSize is able to reduce it by 37%. For 3-AR, the reductions from the L&M GOA are 36% (OpCost) and 23% (OpSize); for 4-AR, 34% (OpCost) and 20% (OpSize).

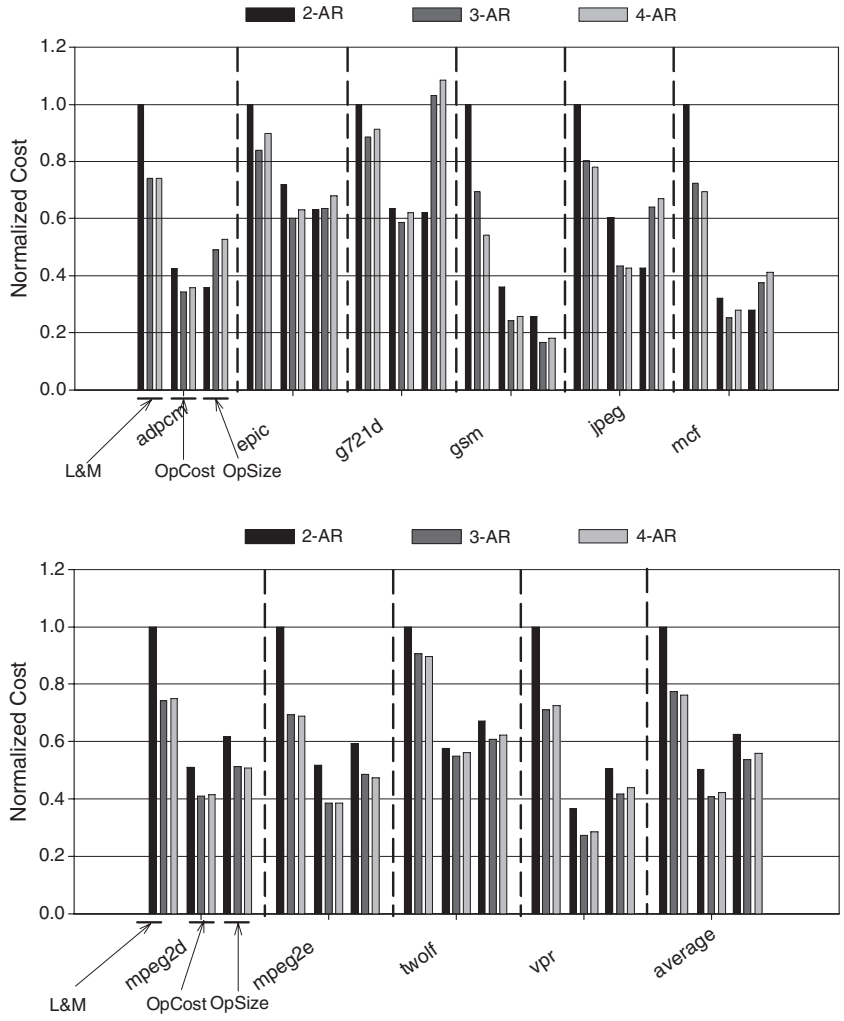


Fig. 21. GOA cost comparison—L&M vs. OpCost and OpSize.

Figure 21 tells us that the best performance results should be obtained through the OpCost algorithm and with 3 ARs. Therefore, we will focus on such setup. Our experiments also show that the OpSize algorithm is close to OpCost except that it performs a little worse (as shown for single-AR).

In Figure 22, we compare costs for a number of phase combinations. All numbers are normalized to that of the L&M GOA with 3 ARs. Generally the cost reduction is not as significant as in single-AR, possibly because the L&M GOA has been successful in cutting a higher percentage of costs. However, we still observe the effectiveness of both PR and post/pre-optimization except that the two are closer than for single-AR. Given that less variables are managed by each AR, it is more likely that some basic blocks contain few memory access instructions for a certain AR, which makes it a little difficult to satisfy the

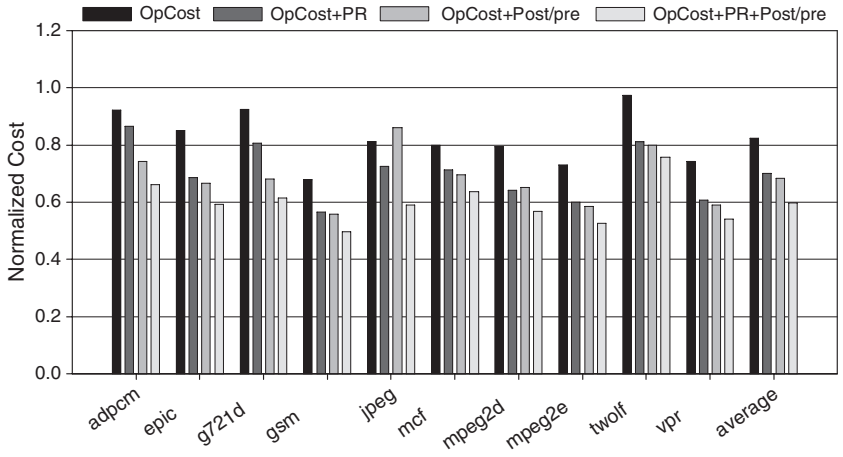


Fig. 22. GOA cost comparison: OpCost, 3-AR.

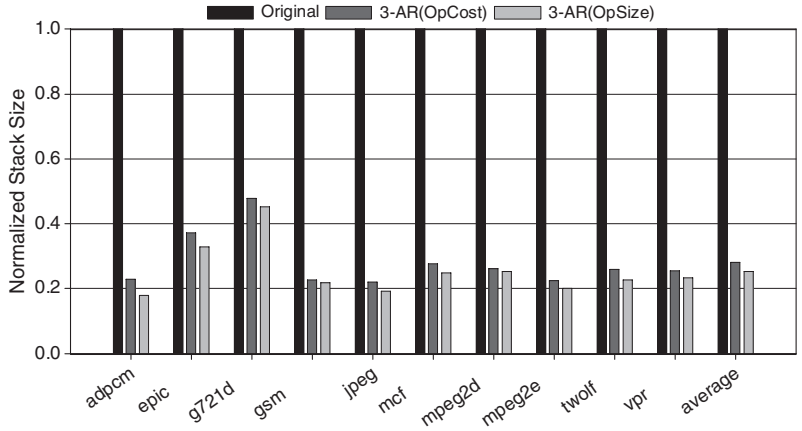


Fig. 23. GOA stack size comparison: OpCost, 3-AR.

conditions given in the Section 8.7 for a better utilization of both post- and pre-increment mode across basic blocks. To quantify, the average extra reductions over the L&M for PR, Post/Pre, and PR+Post/Pre are 14%, 16%, and 22%.

Figure 23 compares stack sizes for OpCost with 3 ARs. The reductions are 72% (OpCost) and 74% (OpSize), respectively. We do not need to show results for PR and post/pre-optimization, since they have no impact on stack size. In contrast to single-AR, the algorithms for multiple-AR are more effective in stack size reduction because less restrictions are imposed for coalescence. Normally, coalescing aggressively can reduce the number of edges to a point that optimal solutions are achieved (see Table I). In other words, aggressive coalescence most likely can lead to a lower GOA cost, therefore OpCost tends to proceed with more nodes coalesced. On the other hand, our OpSize algorithm constructs AR groups via variable coalescence, which comes after minimal graph coloring. Thus, more nodes can be coalesced after the minimal graph coloring heuristic,

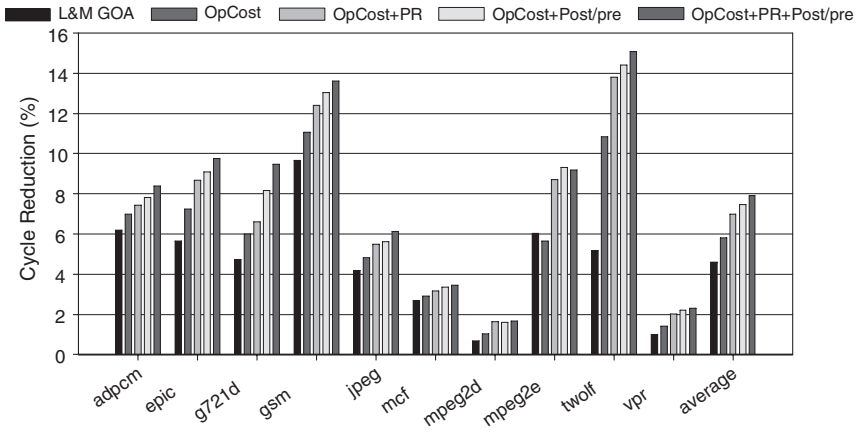


Fig. 24. GOA speedup comparison: OpCost, 3-AR.

Table III. Compilation Time (in seconds)

Bench-mark	Orig.	1 AR			2 AR			3 AR			PR + Post/pre
		INC-TB	OpCost	OpSize	INC-TB	OpCost	OpSize	INC-TB	OpCost	OpSize	
twolf	7.6	8.3	99.5	16.1	241	140.2	21.4	271	139.2	20.7	3.9
vpr	3.5	3.6	5.1	3.9	16.1	7.5	5.0	18.3	7.6	5.6	0.88

which may miss out coalescence chances due to its nonoptimal nature. Also, we observe the difference between 2-AR, 3-AR, and 4-AR to be very small. Actually, even with more ARs, it yields diminishing returns since we are approaching the limit due to the aggressive coalescence.

Finally, we compare speedups for OpCost with 3 ARs. The data include those for PR and post/pre phases. GOA algorithms achieve larger cycle reduction in almost all cases. Post/pre-optimization is still better than PR if both are solely enabled, however, the difference becomes much smaller than for single-AR. Moreover, the advantage of a combined approach is also narrower, possibly due to the fact that the improvements are close to saturation. The average cycle reductions are 4.63%, 5.99%, 7.41%,7.85%, and 8.28% for L&M GOA, OpCost, OpCost+PR, OpCost+Post/Pre, and OpCost+PR+Post/Pre respectively.

With regard to Ottoni et al.'s algorithm in Ottoni et al. [2003], the comparison made was based on our conference publication, which does not contain additional optimizations such as OpCost, program reordering, and post/pre-optimizations. Adding those optimizations, the extra reduction in GOA cost ranges from 14–22% and an extra speedup between 3–4%.

8.4 Compilation Time

Table III shows the compilation time of each optimization phase on a 1 GHz 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). The second column shows the compilation time for the original code, while the last column stands for the time on program reordering

and post/pre-optimization. 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 INC-TB SOA, OpCost, and OpSize. For single-AR, the INC-TB SOA is the 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. The graph coloring heuristic we take can finish very fast. 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.

9. RELATED WORK

The offset assignment problem was initially introduced by Bartley [1992]. The paper identified the SOA problem and proposed a solution to find *Maximum-Weight Path Cover* (MWPC). Liao et al. [1996] proved that finding MWPC is NP-complete. Liao et al. [1996] adopted a SOA solution using a heuristic similar to the classic Kruskal's shortest path algorithm. In addition to SOA, they also considered the case when multiple address registers are available (called GOA).

Later on, Leupers and Marwedel [1996] extended Liao et al.'s [1996] heuristic by adding a tiebreak algorithm. When edges with the same weight can be selected on the access graph, the tiebreak algorithm kicks in to decide which edge should be added to MWPC. The tiebreak algorithm can be used for both SOA and GOA. Leupers and David [1998] proposed a genetic algorithm for SOA. They directly calculate offset assignments by simulating an evolutionary process without building the access sequence.

Rao and Pande [1999] advocated a technique to reorder the memory accesses for a better solution of MWPC. They used algebraic transformations in the expression tree to change memory accesses. Sudarsanam et al. [1997b] proposed a technique to coalesce variables for SOA. They aim to reduce the memory footprint of the application, but their paper did not show that the optimization can reduce SOA cost.

Another improvement made by Atri et al. [2000] is called incremental SOA. It starts with an initial SOA solution, for example, by Liao et al.'s algorithm, then it tries to perform incremental improvements through exchanging of local edges. Experimental results indicate that incremental SOA can improve the initial solution by 3–8% in terms of the SOA cost.

Leupers [2003] proposed a new combination of SOA algorithms called INC-TB SOA, which stands for incremental-tiebreak SOA. It combines two fast SOA algorithms: the tiebreak SOA and incremental SOA. In this combination, the tiebreak SOA is used for constructing the initial solution. As shown in his

paper, this combined algorithm results in the best heuristic algorithm among all algorithms considered in the paper. There is INC-TB SOA is also picked in our framework as the baseline SOA solver.

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. Also, The SOA solver used in this framework can be replaced with any existing SOA algorithms proposed in literature, such as the incremental SOA [Atri et al. 2000], a genetic algorithm [Leupers et al. 1998], etc. 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. 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.

As mentioned earlier, we also notice an independent work on coalescence-based SOA [Ottoni et al. 2003] which came later than our conference publication [Zhuang et al. 2003], and a journal publication [Ottoni et al. 2006] with extension to GOA. In their paper, the coalescence algorithm for SOA is more ad hoc in terms of the selection of node pairs to coalesce and the simplified iteration phase. Actually, similar approaches have been attempted during our early experiments. Due to the fluctuation of the solution quality, we later added the iteration phase 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. For both SOA and GOA, we offer two optimization objectives: OpCost and OpSize. To further boost the performance, we incorporated a post/pre-optimization phase to decide whether post- or pre-modification mode should be used for each access, which works across basic block boundaries. This part is not addressed by any of the previous work. A performance comparison has been made in Section 8, which indicates that with more optimization targets, more optimization phases and an iterative phase, the coalescence-based algorithm proposed in this article can achieve better performance for both SOA and GOA.

Finally, another type of the offset assignment problem, called Array Reference Allocation (ARA), attempts to optimize accesses to array variables using auto-increment/decrement mode [Araujo et al. 1996; Gebotys 1997; Leupers 1998; Ottoni et al. 2001]. Our focus in this work is to optimize using scalarized version of the intermediate form, therefore we do not consider arrays in this work.

10. CONCLUSION

This article proposes a framework for better utilizing the auto-addressing mode on embedded processors. Our optimization framework includes two enhancements to existing work, that is, 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 costs. By incorporating it seamlessly with a SOA solver,

our framework can work in tandem with any SOA solvers, making it very 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 article demonstrates the capability of variable coalescence and post/pre-optimization to provide a new technique to get around these limitations and provide new insights into the overall solution space of this problem.

Our results show that the cost can be reduced by up to 72% (OpCost) for single-AR, almost doubling the cost reduction for a baseline solver with tiebreak SOA. Including all phases can increase the reduction to over 80%. On the other hand, a coalescence-based approach can also shrink the stack size a lot. As observed from the OpSize heuristic, the stack size reduction is up to 67%. Eventually we achieve a cycle reduction of over 5.33% for both OpCost and OpSize when all optimization phases are present.

For multiple-AR, we found that adding too many address registers actually increases the cost, because the total number of registers is fixed. Allocating more for auto-addressing mode deprives the processor of registers for other purposes. Compared with the baseline GOA algorithm, variable coalescence is equally effective for multiple-AR. We observed 14% to 23% cost reduction over the baseline GOA for 3-AR. Moreover, the stack size reduction for multiple-AR is more phenomenal due to its aggressiveness in coalescing variables. With 3-AR, the stack size reduction mounts to 72%$\sim$74%. Finally, the cycle reduction as observed in our experiments can reach 8.28% when all phases are enabled for 3-AR.

In summary, performing variable coalescence and other optimizations after offset assignment like the post/pre-optimization gives new opportunity to exploit auto-addressing 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. The savings in stack area and saving in dynamic counts of AR modification instructions are dramatic and could be of big significance to embedded processors that have very limited stack area allocated. We anticipate that our scheme would be of even more value on register-scarce architectures and/or for memory-intensive applications.

REFERENCES

AHO, A.V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.

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.

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 Conference on Languages and Compilers for High-Performance Computing*.

BARTLEY, D. H. 1992. Optimizing stack frame accesses for processors with restricted addressing modes. *Softw. Pract. Exper. 22*, 2, 101–110.

CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. W. 1981. Register allocation via coloring. *Comput. Lang. 6,* 1, 47–57.

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

GANSSLE, J. G. 1992. *The Art of Programming Embedded Systems*. Academic Press, San Diego, CA.

GEBOTYS, C. 1997. DSP address optimization using a minimum cost circulation technique. In *Proceedings of the International Conference 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 12*th *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 Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE.

LEUPERS, R. AND MARWEDEL, P. 1996. Algorithms for address assignment in DSP code generation. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. 109–112.

LEUPERS, R. AND DAVID, F. 1998. A uniform optimization technique for offset assignment problems. In *Proceedings of the International System Synthesis Symposium (ISSS).*

LEUPERS, R. 2003. Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03).*

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 Conference on Programming Language 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. Program. Lang. Syst. 18*, 3, 235–253.

MOTOROLA, INC. 2000. Motorola DSP56300 family manual, revision 3.0.

MOTOROLA, INC. 2001. SC140 DSP core reference manual, revision 3.0.

MOTOROLA, INC. Motorola DSP56300 family optimizing C compiler user's manual. Motorola, Inc.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufman, San Francisco, CA.

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 Conference on Compiler Construction (CC)*.

OTTONI, D., OTTONI, G., ARAUJO, G., AND LEUPERS, R. 2006. Offset assignment using simultaneous variable coalescing. *ACM Trans. Embed. Comput. Syst. 5*, 4, 864–883.

RAO, A. 1998. Compiler optimizations for storage assignment on embedded DSPs. M. S. thesis, Department of Electrical and Computer Engineering and Computer Science, University 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 Conference on Programming Language Design and Implementation (PLDI)*. 128–138.

STALLMAN, R. 2002a. Using the GNU compiler collection. In *User's Manual*. Free Software Foundation. Boston, MA.

STALLMAN, R. 2002b. GNU compiler collection internals. In *Reference Manual*, Free Software Foundation, Boston, MA.

SUDARSANAM, A., LIAO, S., AND DEVADAS, S. 1997a. 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. 1997b. Optimization of embedded DSP programs using post-pass data-flow analysis. In *Proceedings of the International Conference 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 Conference (DAC)*.

ZHANG, Y. AND YANG, J. 2003. Procedural level address offset assignment of DSP applications with loops. In *Proceedings of the International 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 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

Received July 2008; revised May 2009; accepted August 2009