Storage Assignment Optimizations through Variable Coalescence for Embedded Processors

Xiaotong Zhuang ChokSheak Lau Santosh Pande

Georgia Institute of Technology
College of Computing
801 Atlantic Drive
Atlanta, GA, 30332-0280
{xt2000, chok, santosh}@cc.gatech.edu

ABSTRACT

Modern embedded processors with dedicated address generation unit support memory access with indirect addressing mode with auto-increment and decrement. The auto-increment/decrement mode saves address arithmetic instructions.

Liao et al [2][3] categorized this problem as simple offset assignment (SOA) problem and general offset assignment (GOA) problem, which involve storage layout of variables and assignment of address registers respectively proposing heuristic solutions. Later work [6][7] proposed improvements in the performance of Liao's solution by undertaking program and storage transformations that affect access sequence.

In this paper, we propose a new approach of variable coalescence, which can reduce both instruction segment and data segment size and improve the utilization of automatic address register modification. Variable coalescence combines been observed in terms of code and data size reduction, SOA and GOA cost reduction and dynamic cycle reduction. Variables not interfering with other (not simultaneously live at any program point) into the same memory location. Coalescing allows simplifications of the access graph yielding better SOA solutions or can perhaps lead to such a few uncoalesceable memory locations that GOA solutions for them are optimal. Moreover, it can reduce the program footprint both statically and at runtime (for stack variables) in terms of data segments. Variable coalescence is orthogonal to other solutions proposed; performing variable coalescence first and then solving the SOA or GOA problem with other techniques leads to excellent solutions. In this work, we have successfully applied it to both SOA and GOA problem.

The algorithms are incorporated into and evaluated on the commercial compiler provided by Motorola to boost code generation performance on the DSP 56k chip. Compared to previous approaches, variable coalescence with *program reordering* reduces SOA costs by 48% and GOA (2AR) costs by 66% for Mediabench and SPEC benchmarks. Moreover, we show that our approach obtains theoretically optimal solution (zero cost) for the GOA problem in 87% of the cases with just 2 address registers and in 94% of the cases with 3 address registers.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors– Compilers, optimization; C.3 **[Special-purpose and application-based Systems]:** Real-time and embedded systems

General Terms

Performance, Design

Keywords

Storage Assignment, SOA, GOA, Variable Coalescence

1. INTRODUCTION AND RELATED WORK

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

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

Most modern DSP processors have at least 8 address registers. For example, the Sony pDSP chip and the Motorola DSP56300 processor each has 8 address registers. Starcore's SC140 has 16 address registers. 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. The hardware support shows 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 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. Recent study [11] also shows that on some embedded processors up to 55% of operations could use address register operations to reduce cycle counts and code size. Therefore, significant opportunities exist for optimizing address register assignments.

The storage assignment problem was first studied by Bartley [1] and Liao et al [2][3]. They identified the problem as two classes. The *simple offset assignment* (SOA) problem considers only one address register, while the *general offset assignment* (GOA) problem handles more than one address register. The problem is modeled as a graph and the objective is to find the *maximum weight path cover* (MWPC). Liao proved that finding the MWPC is NP-complete, and so heuristics are used to solve both SOA and GOA. Later, Leupers and Marwedel [10] extended Liao's work by proposing a *Tie-break* heuristic for SOA and a variable partitioning strategy for GOA to reduce the SOA and GOA costs. Atri, Ramanujam and Kandemir [12] further improved the heuristics by an algorithm called *Incremental-Solve-SOA*. Sudarsanam et. al. [5] studied the offset problem in the presence of auto-increment/decrement feature that varies from –l to +l with k address registers. Rao and Pande [6][7] attempted to reorder the memory access sequence (called *program reordering*) through algebraic transformations on the expression trees. The problem is formulated as seeking the *least cost access sequence* (LCAS). [13][14][15][16] talk about the problem of allocating address registers to array references using auto-increment/decrement mode.

In this paper, we propose a different approach to first identify the *webs* or *atomic variables*, and then coalesce them aggressively into fewer memory locations. Our study shows that the access graph of the atomic variables is sparse, and coalescence can effectively reorganize them to generate *simpler* access sequences with high-weighted path covers. Besides, aggressive coalescence can significantly reduce the static and dynamic memory space requirements for both SOA and GOA problems. Another important feature of the coalescence algorithm is it can be combined with most previous approaches to further boost the performance.

Our SOA solution gives a 33% (48% when combined with *program reordering technique [6][7]*) cost reduction and 24% data segment size reduction. For GOA problems, we show that through aggressive coalescence, over 87% of the procedures get optimal solutions (almost all intra basic block address register operation can be handled by auto-increment/decrement instructions) with only 2 address registers (ARs) and over 94% of the procedures get optimal solution with 3 ARs. In the case of 2 ARs, our solution reduces GOA cost (please refer to next section for the definition of SOA and GOA cost) by 24% (66% when combined with *program reordering [6][7]*) compared to one of the classic GOA algorithms. The data segment size is reduced by 24%, while none of the previous approaches resulted in any data segment size reduction.

The remainder of the paper is organized as follows: Section 2 briefly introduces the SOA and GOA problem, Section 3 gives a motivating example for our approach, Section 4 talks about preliminaries, Section 5 presents terminologies and main results for variable coalescence, Section 6 is the overall framework, Sections 7 and 8 present the SOA and GOA algorithms, Section 9 covers issues about global variables, Section 10 shows results, and finally, Section 11 concludes the paper.

2. SOA AND GOA PROBLEM

Offset assignment is the problem of assigning offsets (memory layout) to variables so that the number of address arithmetic instructions can be minimized by using auto-increment/decrement modes of register indirect addressing instructions. For example, Figure 2a shows the memory layout for 6 variables (address grows upwards) and generated code corresponding to Figure 1.a—we assume one address register AR0, so it is a Simple Offset Assignment (SOA) problem. For the first instruction c=a+b, after accessing b, i.e. ADD *(AR0)-, we use auto-decrement to point AR0 to the memory location of variable c, thus saving one address register modification instruction (like ADAR—add to AR, and SBAR—subtract from AR). Thus, the problem of maximizing the use of auto-increment/decrement instructions is to find a good memory layout for the *access sequence* (the order variables are accessed) such that explicit address modification instructions are (such as ADAR and SBAR) are minimized.

Liao et al [2] modeled the problem of SOA by an undirected graph called *access graph* (AG), which is built from the *access sequence*. In Figure 1.a, we show the access sequence (below the code) for the 5-line code segment. Here, we assume that variables on the right-side of the equation must be loaded one-by-one from left to right, then, after the evaluation, the result is stored into the left-side variable. For the time being, assume that all variables are stored in memory (in case they are not, access graph will show the order of only those accesses corresponding to memory accesses, i.e. load/stores). The access graph is shown in Figure 1.b. Each node is a variable. The edge *weight* represents the number of times the two nodes are accessed consecutively in the access sequence (the edge weight is statically estimated by the compiler by counting the number of transitions or could be built using profile information). The problem now becomes one of finding a *maximum weight path cover* (MWPC) [2] for the graph. The MWPC is a *path cover* (a path cover is one or several acyclic path passing through all the nodes such that no node has more than two neighbors on the path; it can be directly converted into a linear memory layout sequence) that has maximal weight. The thick line in Figure 1.b shows one of the MWPC solutions. The weight of the MWPC is the number of address register modification instructions saved. The sum of the weights of all edges not on the MWPC is equal to the number of times address register modification instructions should be used, and this sum is called the *SOA cost* ([2] gives details on the SOA cost, intuitively, for uncovered edges, address register modification instructions must be inserted and the edge weights now represents how many times these instructions are executed).

Earlier approaches [2][10][12] have shown the MWPC problem is NP-complete and tried to find a good path cover with a weight close to the MWPC. Also, program reordering [6][7] was used to modify variable access sequences through algebraic laws (like from a+b to b+a), such that the MWPC solution can be improved. But access graphs are dense and finding good graph-theoretic solutions to them are limited by the complexity (NP-completeness) of the problem.

For General Offset Assignment (GOA) problem, the general approach is to assign each variable to an address register (AR), where a variable assigned to an AR uses that AR only. Then for all variables assigned to the same AR, the problem is solved as SOA. *GOA cost* is actually the sum of SOA costs for all address registers. For GOA, the access sequence for variables handled by one AR is derived from the all-variable access sequence but considering only the variables using that AR. For example, in Figure 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.

As alluded previously, current techniques to solve this problem attempt to efficiently partition graph first (GOA problem) and then solve each sub-graph as SOA. But access graphs are dense and finding good graph-theoretic solutions for partitioning them (GOA) or for path cover (SOA) are limited by the complexity (NP-completeness) of the problem. This prompts our approach, which attempts to simplify the access graphs using memory coalescence of values. In the next section, we first illustrate the importance of our approach through an example.

3. MOTIVATING EXAMPLE

In Figure 1, we give an example to illustrate how our variable coalescence algorithm works and how it can reduce the cost when other methods fail.

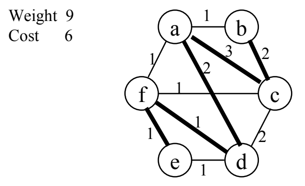
The code segment in Figure 1.a (taken from [6][7] with minor changes) contains 5 instructions. We assume this code segment is the entire program itself. In real programs, we need to do liveness analysis and variable renaming/coalescing either inside a whole procedure (for local variables) or inside a whole program (for global variables).

The coalescence algorithm actually first separates values into atomic units called *webs* (explained in Section 4.2) [9] through variable renaming. A *web* is a du/ud chain closure of a variable and allows independent allocation of values in memory [9].

(1) c=a+b
(2) f =d+e
(3) a=a+c
(4) c=d+a
(5) b=d+f+c

abcdefacadacdfcb

(a)

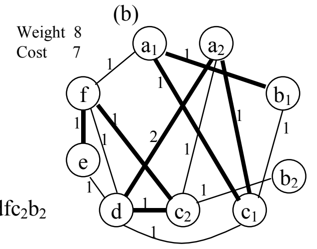


(b)

(1) $c_1=a_1+b_1$
(2) f =d+e
(3) $a_2=a_1+c_1$
(4) $c_2=d+a_2$
(5) $b_2=d+f+c_2$

$a_1b_1c_1defa_1c_1a_2da_2c_2dfc_2b_2$

(c)

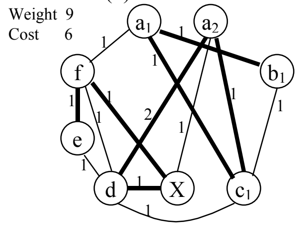


(d)

(1) $c_1=a_1+b_1$
(2) f =d+e
(3) $a_2=a_1+c_1$
(4) $X=d+a_2$
(5) X=d+f+X

$X= (b_2, c_2)$

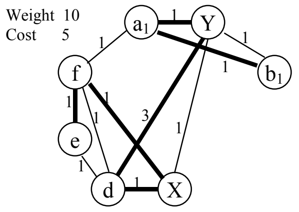
(e)



(f)

(1) $Y=a_1+b_1$
(2) f =d+e
(3) $Y=a_1+Y$
(4) X=d+Y
(5) X=d+f+X
$X= (b_2, c_2)$
$Y= (c_1, a_2)$

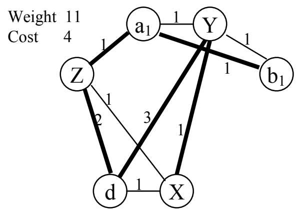
(g)



(h)

(1) $Y=a_1+b_1$
(2) Z =d+Z
(3) $Y=a_1+Y$
(4) X=d+Y
(5) X=d+Z+X
$X= (b_2, c_2)$
$Y= (c_1, a_2)$
Z= (e,f)

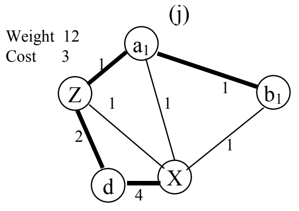
(i)



(j)

(1) $X=a_1+b_1$
(2) Z =d+Z
(3) $X=a_1+X$
(4) X=d+X
(5) X=d+Z+X
$X= (b_2, c_2, c_1, a_2)$
Z= (e,f)

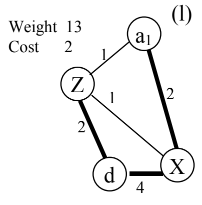
(k)



(l)

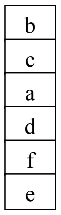
(1) $X=a_1+X$
(2) Z =d+Z
(3) $X=a_1+X$
(4) X=d+X
(5) X=d+Z+X
$X= (b_2, c_2, c_1, a_2 , b_1)$
Z= (e,f)
$a_1XXdZZa_1XXdXXdZXX$

(m)



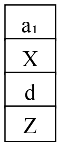
(n)

Figure 1. Motivating Example.



```
LDAR  AR0&a   ; a
LD    *(AR0)  ;
ADAR  AR0, 2  ; b
ADD   *(AR0)- ; c
ST    *(AR0)  ;
SBAR  AR0, 2  ; d
LD    *(AR0)  ;
SBAR  AR0, 2  ; e
ADD   *(AR0)+ ; f
ST    *(AR0)  ;
ADAR  AR0, 2  ; a
LD    *(AR0)+ ; c
ADD   *(AR0)- ; a
ST    *(AR0)- ; d
LD    *(AR0)+ ; a
ADD   *(AR0)+ ; c
ST    *(AR0)  ;
SBAR  AR0, 2  ; d
LD    *(AR0)- ; f
ADD   *(AR0)  ;
ADAR  AR0, 3  ; c
ADD   *(AR0)+ ; b
ST    *(AR0)  ;
```

(a)



```
LDAR  AR0&a1  ; a1
LD    *(AR0)- ; X
ADD   *(AR0)  ; X
ST    *(AR0)- ; d
LD    *(AR0)- ; Z
ADD   *(AR0)  ; Z
ST    *(AR0)  ;
ADAR  AR0, 3  ; a1
LD    *(AR0)- ; X
ADD   *(AR0)  ; X
ST    *(AR0)- ; d
LD    *(AR0)+ ; X
ADD   *(AR0)  ; X
ST    *(AR0)- ; d
LD    *(AR0)- ; Z
ADD   *(AR0)  ;
ADAR  AR0, 2  ; X
ADD   *(AR0)  ; X
ST    *(AR0)  ;
```

(b)

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

Figure 2. Assembly code (a) before, and (b) after Coalescence

Figure 1.c shows how we separate each one 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 closed 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. 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, i.e. 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). However, coalescing if not done properly can jeopardize the access sequence; we propose a coalescing algorithm, which is access-sequence sensitive to generate good solutions.

We now discuss the effect of coalescing on GOA for multiple address register (ARs). Suppose 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, e.g. {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 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 8 that coalescence can often generate an optimal solution for GOA.

Figure 1.b already shows the optimal solution of MWPC for the case of no coalescence, and therefore no heuristic can reduce the cost below 6 without variable coalescence. As far as program reordering is concerned, it is also applicable to the code after coalescence as shown in Figure 1.m, so 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 (cycles) and code size.

4. PRELIMINARIES

We first introduce some key concepts behind our framework.

4.1. Assumptions

Most of the basic assumptions are followed from previous works [1][2][3][6][7][10]. Others specific to our approach are as follows:

1. We do a simple alias analysis [17] to determine the variables that might be referenced via pointers.
2. Not all address register operations can be converted into auto-increment/decrement instructions. 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 calculation sometimes involves multiplication or shifting. Our algorithm does not tackle this.
4. The first address register instruction in a basic block cannot be tackled if the control can come from several predecessors and the last variables accessed are not the same. Although, in some way these cases can be solved with combination of pre and post modifications, it is out of the scope of this paper (and is being tackled in a journal version of this paper).

4.2. Webs and Variable Separation

In order to separate memory references, which can be independently considered for allocation, we use the concept of *web*, i.e. a group of connected definitions and uses. A *web* [9] or *live range* is defined as the maximal union of du-chains. Each web builds a separate variable after renaming, i.e. one must bind all the definitions and uses within a web to a single memory location. Therefore, we also call it *atomic variable*. In this manner, we are able to achieve effective value separation at different program points. Value separation is extremely important as the compiler normally generates lots of temporaries that are re-used repeatedly. Decoupling these variables that are disjoint in terms of values through re-naming gives us more freedom to coalesce them in a proper way to maximize the profit of storage assignment optimizations. This is shown to be effective in the example from 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 that can make use of the auto-increment/decrement instructions are re-cycled temporaries and the data segment size for them can increase after web identification. However, coalescing phase which follows greatly reduces the data segment size and bring about an overall size reduction when compared to the original data segment size.

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

4.3. Interference Graph and Coalescence Graph

After values are separated into atomic variables, our coalescence algorithm needs to further determine which variables are coalesceable.

An *interference graph* (IG) is built to represent the potential overlapping of the live ranges between different variables. The IG is defined as a graph where each node is an atomic variable and an edge between a pair of nodes means the two nodes share overlapping live ranges in the program, i.e. at a certain program point, the two atomic variables are simultaneously live, so they cannot be coalesced.

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

In our 10 benchmark programs, the IGs after value separation are sparse. Intra-procedurally, the average degree for each node is 8.17 on the IG and 210 for the CG. The strong connectivity on the CG means atomic variables have plenty of choices 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.

5. VARIABLE COALESCENCE

5.1. Profitability of Variable Coalescence

The high degrees of nodes of CG allow us enough freedom to make good coalescing decisions for simplifying the access graph (AG) considerably. Simplifying access sequence through judicious choice of coalescing is a non-trivial problem. Coalescence must be performed so that the access graph is simplified in terms of its path cover and resulting MWPC solution for SOA problem. A key observation is that, increased edge weights due to coalescing are unrelated to the overall weight increase in the path cover. Coalescing 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.

Figure 3.a shows the original access graph and the current status of MWPC, i.e., a-b-c-d-e-g-h and f with total weight 21. If the coalescence graph permits node c and h to coalesce, we can coalesce the two nodes and get a MWPC (a-b-ch-d-e-g and f) in Figure 3.b, the weight is 20. After coalescence, the MWPC is worse. The reason is because node c already has 4 neighbors. Adding more neighbors from h is not going to gain much. In contrast, in Figure 3.c, we coalesce node d and g. The MWPC is a-b-c-dg-e-f and h with weight of 22. This example tells us coalescence cannot be done arbitrarily without consideration of the topology of the IG and AG.

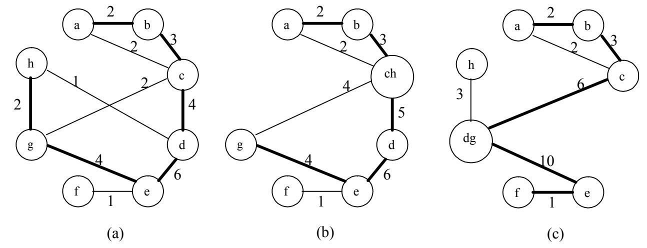


Figure 3. Profitability of variable coalescence.

5.2. Problem Formulation

The problem of storage assignment through variable coalescence is to find both the coalescence scheme and the MWPC on the coalesced graph. Here are some terminologies and lemmas for variable coalescence.

DEFINITIONS:

***Coalesced Node (C-Node)*:** A C-node is a set of atomic variables (*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 atomic variable is a C-node by itself.

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

$\{<n_1,n_2> \mid n_1 \in c_1, n_2 \in c_2, <n_1,n_2> \text{ is an edge on AG or IG}\}$.

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

Coalesced Path Cover (C-PC): 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-node c_i and c_{i+1}. The C-PC covers all C-nodes exactly once, contains no cycles, and no C-node has a degree larger than two in the C-PC. C-PC always refers to a PC on a C-AG.

Weight of a C-Edge: The weight of a C-edge is the sum of all edge weights in the C-edge. 0-weight C-edges are eliminated from the graph.

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

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

***C-MWPC (Coalesced Maximum Weight Path Cover)*:** The C-MWPC is the C-PC with the maximum weight for all possible C-PCs on the C-AG.

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 (edges are from the AG).

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 (edges are from the IG). A C-edge between two C-nodes means the two C-nodes has interference live ranges, and cannot be coalesced.

The algorithm starts by considering the AG as a starting point where each node is labeled as a C-node and by using IG, updates the C-nodes in both through coalescing leading to C-AG and C-IG which keeps on changing dynamically as we coalesce more and more C-nodes. We first show that optimal coalescing for best MWPC (called C-MWPC) is a hard problem. Next we attempt heuristic solution for it through a set of coalescing rules.

LEMMA 1: The C-MWPC problem is NP-complete—Proof in Appendix A.

LEMMA 2: Solution to the C-MWPC problem is no worse than the solution to the MWPC—Proof in Appendix A.

6. OVERALL FRAMEWORK

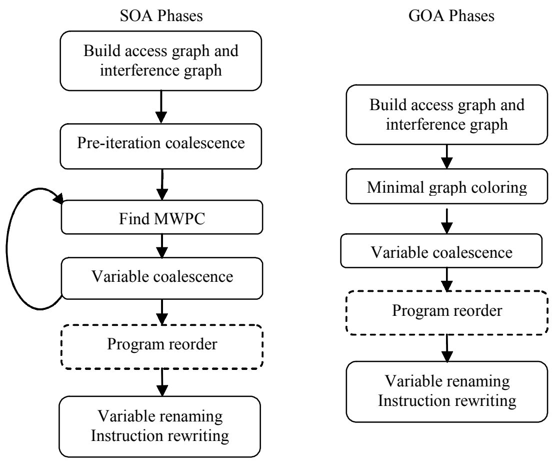

SOA Phases
GOA Phases
Build access graph and interference graph
Pre-iteration coalescence
Find MWPC
Variable coalescence
Program reorder
Variable renaming Instruction rewriting
Build access graph and interference graph
Minimal graph coloring
Variable coalescence
Program reorder
Variable renaming Instruction rewriting

Figure 4. Framework for the SOA solver.

Figure 4 shows the overall framework of our coalescing based SOA and GOA solver. Both solvers begin with building the access graph (AGs) and interference graphs (IGs). For SOA, a heuristic approach is chosen to iterate over MWPC searching and variable coalescence after the pre-iteration coalescence (explained in the next section) is done. As presented in the next section, in every iteration, the heuristic algorithm finds 2 C-nodes to coalesce if possible. Then, the two C-nodes are coalesced and the C-AG and C-IG changed. An existing MWPC solver is run to find a C-MWPC solution. The solution with the least cost ever achieved is saved and used as the final solution.

For GOA, in most cases, we aggressively coalesce the nodes by minimally coloring the IG, since the minimal number of C-nodes can possibly lead to an optimal solution. Then we use existing algorithms to find a GOA solution on the C-AG. We also propose a coalescence algorithm for GOA, when optimal solution is not available. The program reorder phase [6][7] can be optionally added for comparison. In our implementation, the program reordering is slightly different from [6][7]. It comes after the storage layout has been decided by our SOA or GOA solvers. For all memory access instructions (those related to auto address register modification) inside the same basic block, a dependency DAG is built. Then, we use the commute-3 [6][7], i.e. the exhaustive search to find out the optimal reordering of these instructions without violation to the dependencies. To reduce exhaustive search time for big basic blocks, we reorder for every 15 memory access instructions. Our results show that program reordering greatly boosted performance. Finally, we rename the variables in the code and change the address register instruction to auto-increment/decrement form regenerating the code.

7. STORAGE ASSIGNMENT THROUGH VARIABLE COALESCENCE FOR SOA

Since the C-MWPC problem is NP-complete, we have to use a heuristic algorithm to find solutions in a reasonable amount of time.

Our heuristic algorithm is separated into 2 parts. First, a set of pre-iteration coalescence rules are applied to capture cases that are definitely profitable that do not need algorithmic solution. 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 [10]) 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 sub-optimal C-PC.

7.1. Pre-iteration Coalescence Rules

The pre-iteration rules are applied before we do iterative coalescing. Applying these rules will not worsen the SOA cost in all cases. All these rules are with respect to the access graph (AG).

Note that we can only coalesce a pair of C-node if the C-nodes do not have an interference edge between them.

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

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

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

Rule 3 is explained in Figure 5. For C-nodes A, P, and Q, suppose the C-edge <A,P> is heavier than the C-edge <A,Q>. According to Rule 3, we should coalesce A with P. Assume there is a C-PC solution **without** coalescing A with P. Figure 5.a to Figure 5.d show 4 cases of that C-PC for C-edge <A,P> and <A,Q>. In Figure 5.a, none of the 2 C-edges is a part of C-PC, so the coalescence will gain *Weight(<A,P>)*. In Figure 5.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 5.c), we gain *Weight(<A,P>)*. And, if both of them are on the C-PC (Figure 5.d), the cost is unchanged. Therefore, in each case, coalescing A with P can only improve (or cause no change to) the total weight of the C-PC before A and P are coalesced but will never worsen the solution.

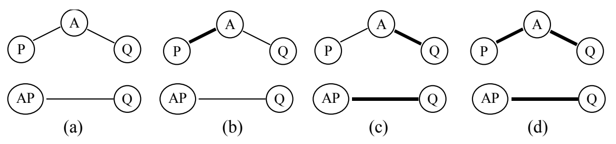


Figure 5. Profitability of Rule 3 Coalescence.

7.2. Saving Due to Coalescence

After applying pre-iteration rules, we start iterating. In each step of the iteration, we pick two C-nodes of maximum calculated savings to coalesce.

The basic idea is to use the current C-PC offset assignment to estimate savings if the 2 C-nodes were coalesced.

For example, Figure 6.a shows a C-AG with 8 nodes. The thick line is the current C-PC of the C-AG. If we coalesce d with g, C-edge <h,d> will now be on the C-PC, and C-edges <c,d> and <d,e> will be eliminated. C-edge <g,d> is also saved after d is merged with g. So, the total saving is W(h,d)+W(g,d)-W(d,e)-W(d,c) = 1, where W(<i,j>) is the weight of a C-edge <i,j>.

In other words, we reduce the SOA cost by 1 if we coalesce d with g. In Figure 7, we illustrate 3 different cases to coalesce J with I. Figure 7.a is a general case.

We save:

a. The weight of the C-edge between I and J.

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

We lose:

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

Figure 7.b is a special case where if I and J are already neighbors on the C-PC, then the weights of both C-edges <I,Q> and <J,P> are saved. In Figure 7.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 one for I to coalesce to J. We take the bigger one as the saving for I and J’s coalescence.

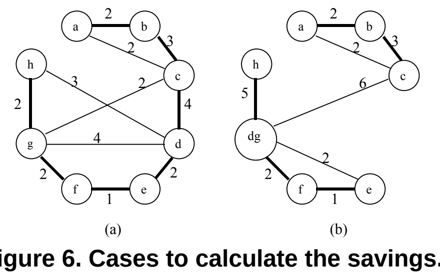


Figure 6. Cases to calculate the savings.

7.3. Tiebreak for the Same Savings

If two or more pairs of C-nodes have the same coalescence savings, we apply a *Tie-break* rule. This tie-breaker is the same as that in [10] for selecting equal-weight edges in building the MWPC. In our case, for each coalescence candidate $\{c_1, c_2\}$, the tiebreak weight T is calculated as:

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

A smaller T has higher priority, as explained in [10]. 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.

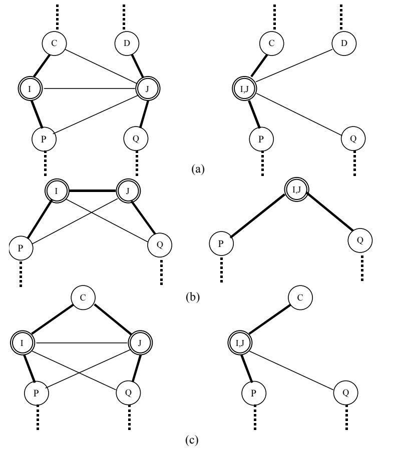


Figure 7. Coalescence cases based on previous C-PC.

7.4. Coalescence Algorithm for SOA

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

coalesce_soa(C-AG, C-IG) {
  Apply_pre_iteration_rules();
  min_soa_cost = soa_cost (C-AG);
  min_node_map = a one to one map

  do{
    find two C-Nodes satify: a.Do not interfere
                             b.Connected on C-AG
                             c.With max_saving
    if(max_saving>0){
      coalesce C-Nodes, update C-AG,C-IG,
      if(soa_cost(C-AG)< min_soa_cost)
        record as min_soa_cost, min_node_map.
    }
  } while(max_saving>0)

  while(there are C-Nodes we can coalesce){
    find two C-Nodes satisfy: a.Do not interfere
                              b.With max_saving
    coalesce C-Nodes, update C-AG,C-IG,
    if(soa_cost(C-AG)< min_soa_cost)
      record as min_soa_cost, min_node_map.
  }
  return min_soa_cost, min_node_map;
}
```

Figure 8. Coalescing Algorithm for SOA.

The whole coalescence algorithm is shown in Figure 8. *coalesce_soa* takes a C-AG and a C-IG as input, and returns the minimal SOA cost and a node to C-node mapping. The original AG and IG is passed to this function. From the node mapping, we can easily generate the final C-AG, C-IG and C-PC solution.

coalesce_soa 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 solver (we implement Liao’s SOA algorithm [2] enhanced with Tiebreak

[10]) to find the SOA cost for the current C-AG. Notice that, the second loop coalesces even when the calculated saving is not positive. This is because our savings calculation is only a heuristic formula. After re-running the SOA solver, we may get a different C-PC, which may have an even lower SOA cost.

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

8. STORAGE ASSIGNMENT THROUGH VARIABLE COALESCENCE FOR GOA

In GOA, we have more than one address register (AR) that can be used to do auto-increment/decrement. With the trend in embedded processor design to increase the number of ARs, GOA is becoming more important. 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.

Since variable coalescence can greatly reduce the number of C-nodes on the graph, in many cases, we can actually get optimal solutions for GOA. The following lemmas give the conditions for the optimal GOA solution.

LEMMA 3: If there are only two C-nodes in the C-AG, then the SOA solution is optimal**—**Proof in Appendix A.

LEMMA 4: 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* for the GOA problem by assigning no more than 2 C-nodes to each address register—Proof in Appendix A.

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). The following lemma says that the minimizing problem is the same as the minimal coloring of the IG.

LEMMA 5: 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 in Appendix A.

COROLLARY 1: If we can color an IG with 2K colors, then there is an optimal GOA solution for 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. Minimal graph coloring is a classic NP-complete problem. In our problem (unlike graph coloring), we would like to minimize the number of colors (Address Registers). This is due to the fact that unused address register could be used as regular registers to hold values improving code quality further. In the register allocation setting, we are simply interested in any feasible solution, which has least spill cost but which uses *any* number of registers unlike minimum number of them. In our solution of the problem, we used a simple heuristic [9] similar to the one used for register allocation but which attempts to reduce the number colors once it finds a feasible solution.

To quantify the number of times we can get optimal solutions with 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 the GOA solver—the dynamic number of optimal solutions. As mentioned in the previous section, Corollary 1 only gives a sufficient condition, i.e. even if an AG has more than two nodes, its SOA cost can still be zero, or the GOA cost can still be zero if the IG is not 2K-colorable. So, the actual number of optimal solutions after the GOA solver could be larger than the one got from the number of colors.

Table 1. Percentage of Optimal Solutions for GOA

#AR	Epic	Gsm	G721	Mpeg2d	Mpeg2e
2 (color)	84.9	85.56	76.92	82.68	63
2 (final)	86.8	90	96.15	90.55	77.23
3 (color)	90.57	93.33	96.15	91.34	81
3 (final)	94.34	97.78	100	94.49	88.12

#AR	Bzip2	Gzip	Mcf	Twolf	Vpr	Average
2 (color)	52.38	85.15	80	62.94	65.83	73.94
2 (final)	87.18	90.1	93.33	79.19	82.01	87.25
3 (color)	87.2	90.1	93.34	76.1	85.25	88.44
3 (final)	92.31	96.04	100	89.85	94.24	94.72

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

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

8.1. Heuristics for solving GOA for Non-optimal cases

If the minimal color is larger than 2K, the algorithm in Figure 9 can still employ a heuristic algorithm to find a solution. The algorithm has two parts. It shares some features with the GOA algorithm in [10]. After variable coalescence, we may find that many minimal *addon* costs are the same, so a more powerful tiebreaker is implemented to handle it properly. Firstly, we calculate the minimal cost to add one of the existing nodes on IG/AG to one of the ARs. The function coalesce_soa is run for the group of nodes of that AR to get the *addon* cost. All such nodes with minimal *addon* costs are recorded in MINISET. In the second part, our algorithm tries to break the ties if the minimal ones are not unique. We calculate two values for tiebreak. Value w1 is calculated for each node v in MINISET. If v is selected for G_i, we sum all the edges on AG from v to a node that is in $G_1 U G_2 .. U G_k$ - G_i. Since, the edge from v to node in another AR is eliminated as we illustrated in the motivation example, we prefer a larger w1. If this still cannot break the 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 ARs. We prefer smaller w2, which means more nodes on the IG later can be coalesced with v. Finally, we pick one randomly if there are more than one node in MINISET; our experiments show this rarely happens.

```
Input: AG, IG, K (#AR)
Output:
  a. The minimal goa cost.
  b. A node map from node to its C-node.
  c. A map from C-node to AR number.

V: node set //contain all nodes initially
G1..Gk: C-node sets //for each AR

coalesce_goa(AG, IG, K) {
  $G_1=G_2=..=G_k=\Phi$;
  call min_graph_color(IG) and get color groups
  $C_1,C_2,…C_n$.
  If (n<=2k)return goa_cost=0, $C_1,C_2,…C_n$.as C-node
  groups, 2 C-nodes to each AR.

  while(V<>$\Phi$){
    MINISET=$\Phi$; min_cost=MAX_INT;
    //build MINISET
    foreach node v in V{
      cost=minimal addon cost to put in one of
           the $G_i$ by running coalese_soa on $G_i$.
      if (cost = min_cost){
        add (v,i) to MINISET;
      }else if(cost<min_cost){
         MINISET={(v,i)}; cost->min_cost;
      }
    }
    //tiebreak
    foreach pair (v,i) in MINISET{
      w1(v)=sum(weight<u,v> on AG) $u \in G_1 \cup G_2 .. \cup G_k - G_i$
      w2(v)=number of v's neighbor on the IG
    }
    keep only pairs with maximal w1 in MINISET.
    If(still not unique)
      Tie break on w2 (keep only smallest in MINISET)
    If still have tie, pick one randomly.
    for select pair(v,i) add v to $G_i$
    remove v from AG and IG
  }
  run coalese_soa on all $G_i$ and
  return 1)the goa cost as the sum of all soa cost
         2)map from node to C-node to AR, derived
           from $G_1$ to $G_k$ and each AR's soa solution
}
```

Figure 9. Coalescing Algorithm for GOA.

9. COALESCENCE FOR GLOBAL VARIABLES

For global variables, we do inter-procedural liveness analysis to find out variables with separable live ranges. This is done through a call graph, to find out the places where the global variable is defined/used in each function. A data flow algorithm then builds the live ranges for each global variable. With respect to the aliasing issues, our approach is conservative. After building the AG and IG for global variables, the same SOA and GOA algorithms is similarly applied to get the memory layout solution.

For global and static variables, the code generator allocates memory locations in the data segment. It is possible that some of the global variables have initial values. Those simultaneously live at the program entry point will not get coalesced, so we only need to assign at most one initial value to each coalesced node.

10. PERFORMANCE EVALUATIONS

10.1. Experimental Environment

Our environment is the Motorola 56300 processor toolset including a cycle-accurate simulator called sim56300, and a retargeted GNU C compiler (GCC), which comes with standard header and library files. Our pass is implemented after the reload pass of GCC, just before the generation of the RTL (GCC's IR), whose output is assembly code, so we can capture all the temporaries and spill codes generated by the compiler.

Among the 8 address registers, one is dedicated for stack pointer and another one is dedicated for base address pointer. Among the remaining 6 address registers, we reserve 3 of them for local variables and 3 of them for global variables.
In our evaluation, a total of 10 benchmarks were used, among them, 5 from Mediabench and 5 from Spec2000int. Code cycle counts were obtained by limiting simulated execution to about 500 million cycles, taking about 3 hours for each benchmark. Limiting the execution time is necessary because large SPECint2000 benchmarks may take months to finish simulation. We use access graphs built using profile information for all results.

Table 2. Statistics for the Benchmarks

	#insn	#procs	stack size	soa cost	goa cost 2AR
Epic	7084	53	135	367	91
Gsm	14664	90	215	864	311
G721	3451	26	91	67	16
Mpeg2d	18530	127	325	544	221
Mpeg2e	28690	101	677	1076	445
Bzip2	12717	81	268	475	119
Gzip	15873	101	268	613	119
Mcf	5073	45	116	141	23
Twolf	99289	197	1664	3081	777
Vpr	57222	278	1255	2493	703

Table 2 lists some statistics for the benchmarks. Column 2 is the total number of instructions. Column 3 is the total number of procedures. Column 4 shows the stack slot size for all procedures. Column 5 and 6 are the sums of all Tie-break SOA costs and GOA (2AR) costs for all procedures.

10.2. Results for SOA

We use *soa* (Tiebreak SOA [10]) as our base comparison. A '*c*' prefix denotes coalescing. A '*cost*' or '*size*' suffix denotes cost-optimized (*coalesce_soa* algorithm) or size-optimized (coloring) coalescing. '*pr*' denotes the use of program reordering.

Table 3 shows that *c-soa-cost* achieves 64.4% stack size reduction and c-soa-size achieves 69.1%. This large reduction shows that many variables can be coalesced. This is not surprising because most variables are compiler-generated temporaries that have a very short live range, and thus they can easily be coalesced with other variables.

Table 3. Stack Size Reductions

	original	c-soa-cost	% reduc	c-soa-size	% reduc
epic	215	129	40.0	122	43.3
gsm	750	184	75.5	171	77.2
g721	202	79	60.9	64	68.3
mpeg2d	688	269	60.9	260	62.2
mpeg2e	1757	462	73.7	408	76.8
bzip2	651	211	67.6	175	73.1
gzip	776	255	67.1	219	71.8
mcf	252	95	62.3	93	63.1
twolf	5547	1648	70.3	868	84.4
vpr	3185	1107	65.2	920	71.1
Average	**1402**	**444**	**64.4**	**330**	**69.1**

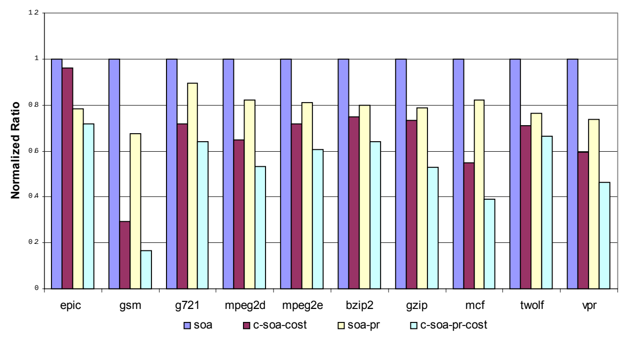


Figure 10. SOA cost comparison (Op-cost).

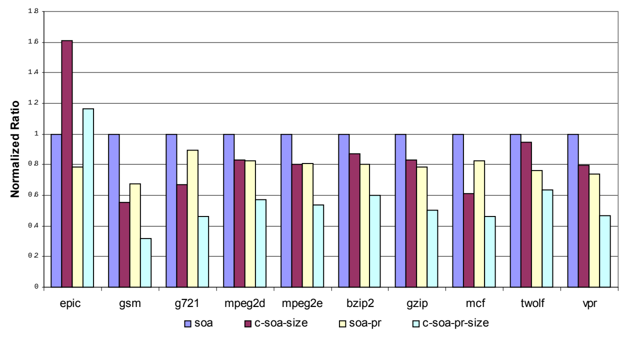


Figure 11. SOA cost comparison (Op-size).

Figure 10 shows *c-soa-cost* generally performs better than *soa-pr*. The average cost reduction is 33.3% for *c-soa-cost*, 22.0% for *soa-pr* and 47.5% for *c-soa-pr-cost*. This shows that coalescing greatly reduces SOA cost (33.3%). Program reordering reduces this cost even further.

Figure 11 shows SOA cost reduction when we try to minimize the stack size. *c-soa-size* has a higher SOA cost than *c-soa-cost*. The average cost reduction is 14.8% for *c-soa-size* and 42.9% for *c-soa-pr-size*. Program reordering actually reduces cost close to that for Op-cost (47.5%, Figure 10).

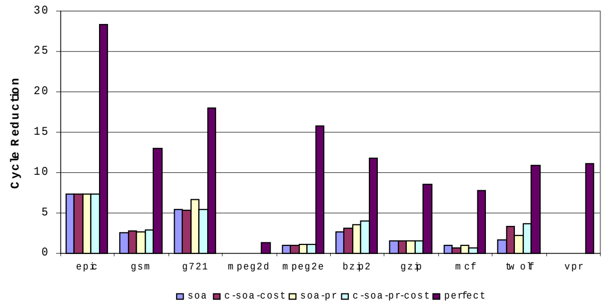


Figure 12. SOA cycle reduction (Op-cost).

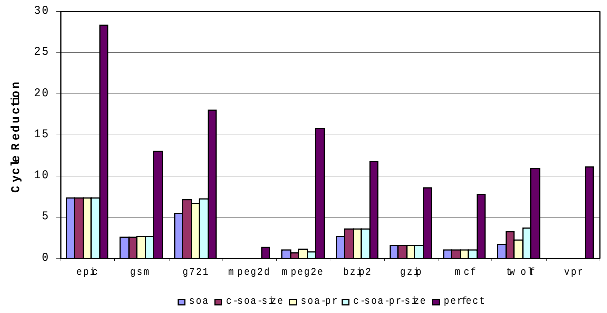


Figure 13. SOA cycle reduction (Op-size).

Figures 12 and 13 shows dynamic cycle count reduction. This reduction includes only local variables. All the numbers are cycle count reduction percentages in comparison to the program's original cycles. In each program, we show the perfect case as the rightmost bar to indicate the upper bound, indicating all address-arithmetic instructions are saved. This perfect case is constant for each benchmark program.

In Figure 12, on average, using *soa* alone reduces the dynamic cycle count by 2.31%; *c-soa-cost* by 2.52%; *soa-pr* by 2.60%; *c-soa-pr-cost* by 2.66%. Compared to *soa*, *c-soa-cost* reduces the cycle count by 9.1% and *c-soa-pr-cost* by 15.1%. Of all instructions, memory access instructions make up 32%. This is more than the average perfect cycle count reduction of 12.6% (cases such as aliased accesses etc. are not safe to be accessed without address register modification). Hence, if we have more memory instructions, we can gain a bigger cycle reduction. If we had used a register-scarce architecture in our tests, there would be more spills, thus creating more memory access instructions. Therefore, the cycle reduction would be even greater on register-scarce architectures and on memory intensive applications.

In Figure 13, the cycle reduction is 2.71% for *c-soa-size* and 2.79% for *c-soa-pr-size*. This shows that the Op-size algorithm can achieve a greater cycle reduction than Op-cost, although this does not always happen.

10.3. Results for GOA

We compare results between Leupers and Marwedel's GOA [10] (*goa*) and our GOA solver (*c-goa*). '*pr*' indicates the use of program reordering.

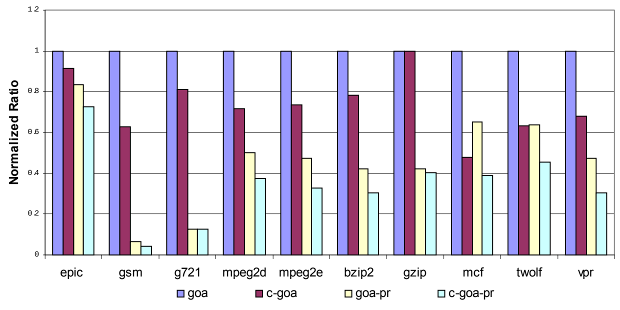


Figure 14. GOA cost comparison-2AR.

Figure 14 shows GOA cost for 2 address registers (2AR). Over *goa*, *c-goa* reduces costs by 24%; *goa-pr* by 54%; *c-goa-pr* by 66%. This shows that coalescing reduces GOA cost dramatically. With 2 ARs, the GOA cost is only about 1/10 of the cost of SOA.

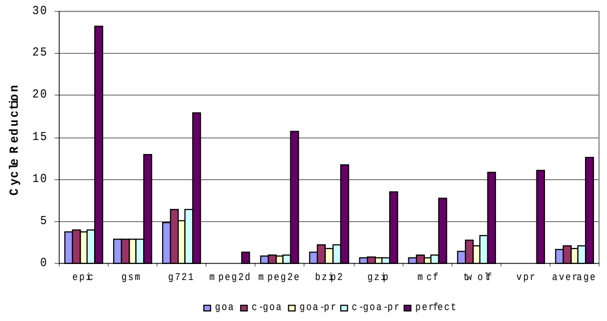


Figure 15. GOA cycle reduction-2AR.

Figure 15 shows percentage code cycle reductions for GOA-2AR. The average cycle reductions for *goa*, *c-goa*, *goa-pr* and *c-goa-pr* are 1.66%, 2.01% (+26.7% over *goa*), 1.78% and 2.14% (+29.1% over *goa*) respectively. The perfect case is 12.6% i.e. such optimizations can achieve an ideal cycle reduction of 12.6% in our benchmarks.

Due to space limitations, we do not show the results for 3 address registers. Generally, using more than 2ARs produces diminishing returns since we are approaching the optimal solution. Also, using too many ARs might worsen the solution because each AR always requires a first address-arithmetic instruction that cannot be saved.

10.4. Results for Global Variables

Global variables are not the main optimizing focus of this paper because they account for less than 15% of all memory accesses. In Table 4, column 2 lists the number of global variables used for each benchmark program. We exclude 2 benchmarks with less than 20 global variables as their results are not informative (0% or 100% cost reduction). If we optimize for size, we can save 83% of the data segment size. *c-soa-cost* reduces SOA cost by 8.5% over *soa*. *c-goa-cost* saves 77.9% on size and 49.6% on cost over *goa*.

Global variables have a lesser cost reduction than that for local variables partially because we did not do program reordering, as it was done for the local variables.

Table 4. Results for Global Variables

	#var	coloring %size reduction	c-soa-cost vs soa		c-goa-cost vs goa (2AR)	
			%size	%cost	%size	%cost
Epic	6	--	--	--	--	--
Gsm	35	74.3	74.3	20	82.9	50.0
G721	18	--	--	--	--	--
Mpeg2d	168	88.7	65.5	21.2	83.3	67.6
Mpeg2e	131	80.2	36.6	1.2	77.1	51.0
Bzip2	49	79.6	51.0	7.2	65.3	33.3
Gzip	133	87.2	40.6	3.8	77.4	54.5
Mcf	8	--	--	--	--	--
Twolf	304	88.2	14.1	4.8	82.6	28.6
Vpr	102	85.3	11.8	1.2	76.5	62.3
Ave.	87.3	83.4	42.0	8.5	77.9	49.6

10.5. Compilation Time

Our largest benchmark, Twolf (about 300KB binary), takes about 4 minutes to compile on a 1GHz Pentium III, using Leuper and Marwedel's GOA. Full coalescing reduces this time to only 11 seconds. Thus coalescing reduces the execution time of GOA algorithms, and most programs compile under 1 second.

11. CONCLUSION AND FUTURE WORK

This paper proposes a framework based on coalescing for both local and global variables to utilize the auto-increment/ decrement instructions in DSP processors. We have shown the advantages of coalescence over previous approaches to capture more opportunities to maximally reduce both static/dynamic code and data size and to speed up the program execution.

This work represents a shift in approaches that solve storage 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 paper demonstrates the capability of variable coalescence to break the performance bottleneck. Compared to previous approaches, variable coalescence with program reordering reduces SOA costs by 48% and GOA (2AR) costs by 66%. Especially, for the GOA problem, coalescence can get almost all (87-94%) optimal solutions quickly together with other benefits. The dynamic stack size reduction is 69% without stack slot reuse. In our experiments, we showed that coalescing can reduce the dynamic code cycle count by 2.8% (*c-soa-pr-size*), corresponding to a 43% reduction in SOA cost.

In short, performing variable coalescence proves to be beneficial on multiple counts dramatically improving the solution space of this important problem faced in a wide variety of DSP processors (ranging from older TI TMS series to the newest StrongARM).

12. ACKNOWLEDGEMENT

The authors gratefully acknowledge the support of the National Science Foundation, which funded this work via grants CCR-0220262 (ITR) and CCR-0073512.

13. REFERENCES

[1] D.Bartley, Optimizing Stack Frame access for processors with restricted addressing Modes, *Software – Practice and Experience*, 22(2): 101-110. Feb 1992.

[2] S.Liao, et al. Storage assignment to decrease code size, *In ACM (PLDI)*, pp. 186-195, 1995.

[3] S.Liao et al. Storage assignment to decrease code size, *ACM Transactions on Programming Languages and Systems,* 18(3): 235-253, May 1996.

[4] A. Sudarsanam, S. Malik, S. Tjiang, and S. Liao. Optimization of Embedded DSP Programs Using Post-pass Data-flow Analysis. *In Proc. of ICCAD'97, Speech, and Signal Processing.* 1997.

[5] A.Sudarsanam, S.Liao, and S. Devadas, Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures. *In Proc. ACM/IEEE DAC*, pp. 287-292, 1997.

[6] A. Rao and S. Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs. *In ACM (PLDI),* pp.128-138, 1999.

[7] A. Rao Compiler Optimizations for Storage Assignment on Embedded DSPs. Master's thesis, Dept. of ECECS, Univ. of Cincinnati, Oct. 1998.

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

[9] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufman, 1997.

[10] R. Leupers and P. Marwedel. Algorithm for Address Assignment in DSP Code Generation, *Proc. ICCAD*, 1996

[11] S. Udayanarayanan and C. Chakrabarti. Address code generation for DSPs. *In Proc. the 38th Design Automation Conference (DAC)*, June 2001.

[12] S. Atri, J. Ramanujam, and M. Kandemir. Improving variable placement for embedded processors, *In Proc. Languages and Compilers for High-Performance Computing,* 2000.

[13] G. Ottoni, S. Rigo, G. Araujo, S. Rajagopalan, S. Malik, "Optimal Live Range Merge for Address Register Allocation in Embedded Programs", International Conference on Compiler Construction (CC01), 2001.

[14] Araujo, G., Sudarsanam, A., and Malik, S. Instruction set design and optimizations for address computation in DSP processors. In 9th International Symposium on Systems Synthesis (November 1996), IEEE, pp. 31-37.

[15] Gebotys, C. DSP address optimization using a minimum cost circulation technique. In Proceedings of the International Conference on Computer-Aided Design (November 1997), IEEE, pp. 100-103.

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

[17] A. V. Aho and R. Sethi, J. D. Ullman, *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.

APPENDIX A

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, a C-node is an atomic variable and the C-PC only consists of atomic variables i.e. the C-PC is the same as the PC. A fully connected interference graph is possible, when all atomic variables have overlapping live ranges. Hence, the C-MWPC problem is NP-complete. □

Lemma 2: 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). □

Lemma 3: If there are only two C-nodes in the C-AG, then the SOA solution 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 4: 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* for the GOA problem by assigning no more than 2 C-nodes to each address register.

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

Lemma 5: The minimal number of C-nodes after node coalescence is equal to the minimal number of colors required to color the IG. Furthermore, a coloring scheme of the IG is equivalent to a legal C-node formation.

Proof: A coloring scheme of the IG can be directly applied to a C-node formation by assigning nodes with the same color in the IG to the same C-Node. The number of C-nodes is the number of colors for the IG. Similarly, a C-node formation can be directed to a coloring scheme by coloring the nodes in the same C-node with the same color and nodes in different C-nodes with different colors. Since nodes in the same C-node do not interfere with each other, i.e. no edge 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. □