Simple Offset Assignment in Presence of Subword Data

Bengu Li   Rajiv Gupta

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
{libengu, gupta}@cs.arizona.edu

ABSTRACT

Many embedded architectures support indirect addressing mode with autoincrement/autodecrement. By maximizing the use of this mode, generation of explicit instructions for performing address arithmetic can be avoided and thus reductions in code size and improvements in performance are achieved. Bartley [2] and Liao et al. [16] developed a method for finding a storage layout for program variables so that the use of autoincrement/autodecrement could be maximized. They introduced the *Simple Offset Assignment* (SOA) problem and solved it using a *Path Cover* (PC) formulation.

We observe that many media and network processing applications make extensive use of subword data. Therefore, for such applications, by packing multiple subword variables into a single word, we can generate storage layouts that further reduce the cost of address arithmetic in two ways. First the need for address arithmetic is reduced as variables that are packed together share the same address. Second opportunities for using autoincrement and autodecrement instructions are increased as layouts are now possible which place a variable adjacent to more than two variables. This approach has become feasible because of the recent trend in embedded processor design which allows subword variables that are packed together to be accessed and manipulated without incurring performance penalty. We introduce the *SubWord Offset Assignment* (SWOA) problem and solve it using a *Path Cover with Node Coalescing* (PCwNC) formulation. Node coalescing corresponds to packing of multiple subword variables into a single word while path covering corresponds to placement of variables in adjacent memory locations to enable the use of autoincrement/autodecrement. We present three heuristics to solve the PCwNC problem. Experiments show that when the program is optimized for code size, the three proposed algorithms achieve 26%, 26.9% and 32% reduction in the number of static explicit address arithmetic instructions over Liao et al.'s algorithm. The algorithms also achieve 14.5%, 22.1% and 22.7% reduction in stack frame size. If the program is optimized for performance, the algorithms achieve 24.3%, 24.7% and 30.2% reduction in the dynamic instruction count of explicit address arithmetic instructions.

CASES'03, Oct. 30–Nov. 2, 2003, San Jose, California, USA.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers*; C.3 [**Special-purpose and Application-based Systems**]: [real-time and embedded systems]

General Terms

Algorithms, Performance, Design

Keywords

subword data, storage assignment, SWOA, SOA

1. INTRODUCTION

The system-on-a-chip design of embedded microprocessors incorporates all the electronics onto a single integrated circuit. Program code resides in on-chip ROM and program size translates directly into silicon area and cost. It is an important goal of the compilers for such architectures to generate compact code to meet the constraints of the limited on-chip ROM size. A number of complimentary approaches have been employed to reduce code size including software-only techniques that reuse recurring code sequences to reduce code size [5, 3] and instruction set support that allows generation of compact code [15, 8, 9, 10].

One of the instruction set features that has received considerable attention for achieving code size reduction is the autoincrement/autodecrement feature associated with the indirect addressing modes in many embedded processors and digital signal processors (e.g., TI TMS320C25, Motorola DSP56K). Variables that are frequently accessed one after another are placed into neighboring storage locations during storage assignment so that the autoincrement/autodecrement feature can be used and explicit address arithmetic instructions to set up the contents of the address register can be avoided. The problem of finding a storage layout which maximizes the use of autoincrement/autodecrement to reduce code size using a single address register is called the *Simple Offset Assignment* (SOA) problem. When multiple address registers are exploited, this problem is called *General Offset Assignment* (GOA) problem. This problem was first formulated as a *Path Cover* (PC) problem by Bartley [2] and Liao et al. [15, 16]. Further extensions have since been developed by many researchers. In [12], Leupers and Marwedel improved Liao's work by proposing a tie-breaking heuristic for SOA and a variable partitioning strategy for GOA to further reduce storage assignment cost. Offset assignment with varying number of address registers and autoincrement by a range of values have also been studied by Sudarsanam et al. in [21] and Leupers and David in [13]. Rao and Pande [19] proposed a method to achieve better access sequence to reduce the code size by performing algebraic transformations on expression trees. Similar problem is studied by Choi and Kim using an offset assignment algorithm tightly coupling with code scheduling [4]. Kandemir et al. use both access pattern modification and memory storage reordering in [7]. *Modify registers* are also considered together with offset assignment in [13, 23]. Zhuang et al. also consider coalescing non-interfering variables into the same memory location to reduce code and data size [26].

Another recent trend in processor design is introduction of specialized instruction set support for handling subword data. Multiple distinct subword data items are typically held in memory locations and registers in packed form. Instructions are provided that can directly operate on these subword entities without unpacking them. Examples of processors with such support include Intel IXP [1] and Infineon network processor [18, 24]. Also limited support for subword operations can be found in XScale and further extensions of ARM processor to incorporate such support has been proposed in [14]. The motivation for providing the above support arises because subword data are very common in the embedded codes that carry out media and network processing tasks [11, 25, 17]. Although C is commonly used language in the embedded domain and it does not support variable width data, recently developed bitwidth analysis techniques can identify the presence of such data automatically [6, 20, 22].

In this paper we show that the presence of subword data in embedded applications, as well as the support of instructions for direct manipulation of subword data, presents an opportunity for achieving effective storage offset assignments that can reduce the need for explicit address arithmetic instructions. By packing multiple subword variables into a single word, we can generate storage layouts that further reduce the cost of address arithmetic in two ways. First the need for address arithmetic instructions is reduced as variables that are packed together share the same address. Second opportunities for using autoincrement/autodecrement instructions is increased as layouts are now possible which place a variable adjacent to more than two variables. We introduce the *SubWord Offset Assignment* (SWOA) problem and solve it using a *Path Cover with Node Coalescing* (PCwNC) formulation. Node coalescing corresponds to packing of multiple subword variables into a single word while a path cover corresponds to traditional placement of variables in adjacent locations. We present three heuristics to solve the PCwNC problem. Experimental results show that the developed techniques are effective.

The remainder of the paper is organized as follows. Section 2 introduces the processor model that supports the features described above and illustrates how they can be used to reduce code size. It also formalizes the layout problem. In section 3, we present the three algorithms we have developed to solve this problem. In section 4 experimental results are given to demonstrate the benefits of our approach. Conclusions are given in section 5.

2. SUBWORD OFFSET ASSIGNMENT

Processor Model and Illustration

We begin by defining the processor model and illustrating the advantages of SWOA over SOA. Noticing that there is an opportunity to improve the storage assignment if multiple subword data can be put into one memory location, we propose a new processor model, which integrates the autoincrement/ autodecrement feature and the ability to directly manipulate subword data.

The processor model we consider is an *accumulator-based* machine where, for each instruction, one operand resides in the accumulator and the other operand resides in the memory. A specialized hardware unit, called Address Generation Unit (AGU), is used to handle memory addressing. AGU contains a set of address registers. The operand in the memory is referenced through one of the *address registers* ($AR0$ $AR1$...). Address register can be set up explicitly using any instruction that can write to it. Address register can also be automatically incremented or decremented by one in parallel with the memory access operation in any instruction. Explicitly setting up address register consumes processor cycles while autoincrement/autodecrement operation does not cost any extra cycles. In assembly, autoincrement of the address register ARi is indicated by $*(ARi)+$ and autodecrement is indicated as $*(ARi)-$. In SWOA problem, only one address register will be considered. In a more general version of SWOA problem, multiple address registers can be exploited.

As an example of autoincrement/autodecrement feature, consider the following instruction:

ADD *(AR0)+.

The above instruction adds the operand in memory pointed to by $AR0$ to the accumulator and at the same time automatically increases the value in $AR0$ by the size of one memory unit.

For both the accumulator and the memory operand, *subword access* mode is available, i.e. the operands can be subword entities within the accumulator or the referenced memory location. A *subword descriptor* of the form $[l, s]$ is used to specify the subword entity such that l is the starting bit position of the subword operand and s is the number of bits in the subword operand. This addressing mode is quite similar to the immediate static descriptor proposed as part of a Bit Section Extension for ARM architecture in [14]. The main difference is that the subword operand being read resides in the accumulator or memory while in the ARM Bit Section Extension the subword operand can come from any general purpose register. To reference subword data in memory, special logic which can read and write subword in a memory location is added. In an instruction, either one or both of the operands can represent subword data. If only one operand is subword data, it must appropriately extended (zero or sign) first before the operation is performed.

The autoincrement/autodecrement feature and subword accessing mode can be used simultaneously in one instruction. As an example consider the following instruction:

ADD [16,8], *(AR0)[24,8]+.

The above instruction performs an add over the subword operand in the memory address pointed by $AR0$ starting at bit position 24 and 8 bits long and the subword operand in the accumulator starting at bit position 16 and 8 bits long. The result is stored in the accumulator and the value of $AR0$ is increased to address the next word by the autoincrement operation.

To illustrate the benefits of above instructions we next present an example. Consider the code fragment in Figure 1(a). Further assume that variables a and d are subword variables of length 16 bits and all other variables are 32 bits. Figure 1(b) shows the access sequence of the variables. Using the algorithm for solving the SOA problem proposed by Liao et al. [15], the layout shown in Figure 1(c) is obtained. Notice that this algorithm assigns variables a and d to separate locations. The assembly code corresponding to this offset assignment is shown in Figure 1(d), where the instructions shown in bold type are explicit address arithmetic instructions.

Now let us assume that we exploit the knowledge that a and d are subword variables. By packing the two variables together into a single location, we obtain the storage offset assignment shown

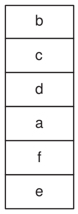
Figure 1 Simple Offset Assignment (SOA).

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

(a)

a b c d e f a d a d a c d f a b

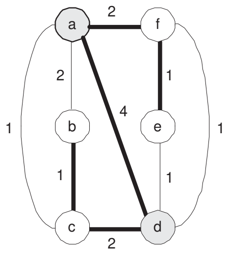
(b)



(c)

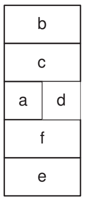
```
LDAR  AR0, &a ; a
LOAD  *(AR0)
ADAR  AR0, 3   ; b
ADD   *(AR0)-  ; c
STOR  *(AR0)-  ; d
LOAD  *(AR0)
SBAR  AR0, 3   ; e
ADD   *(AR0)+ ; f
STOR  *(AR0)+ ; a
LOAD  *(AR0)+ ; d
ADD   *(AR0)-  ; a
STOR  *(AR0)+ ; d
LOAD  *(AR0)-  ; a
ADD   *(AR0)
ADAR  AR0, 2   ; c
STOR  *(AR0)-  ; d
LOAD  *(AR0)
SBAR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

(d)



(e)

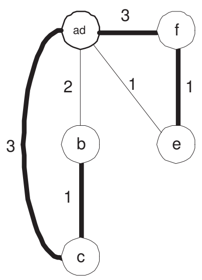
Figure 2 SubWord Offset Assignment (SWOA).



(a)

```
LDAR  AR0, &ad                ; a
LOAD  *(AR0)
ADAR  AR0, 2                  ; b
ADD   [32,16],*(AR0)-         ; c
STOR  *(AR0)-                 ; d
LOAD  *(AR0)
SBAR  AR0, 3                  ; e
ADD   [16,16],*(AR0)+         ; f
STOR  *(AR0)+                 ; a
LOAD  *(AR0)                  ; d
ADD   [32,16],*(AR0)[16,16]   ; a
STOR  *(AR0)[32,16]           ; d
LOAD  *(AR0)                  ; a
ADD   [16,16],*(AR0)[32,16]+  ; c
STOR  *(AR0)-                 ; d
LOAD  *(AR0)-                 ; f
ADD   [16,16],*(AR0)+         ; a
ADD   *(AR0[32,16])
ADAR  AR0, 2                  ; b
STOR  *(AR0)
```

(b)



(c)

in Figure 2(a). The corresponding assembly code is shown in Figure 2(b) which contains fewer explicit address arithmetic instructions. As we can see, the number of explicit address arithmetic instructions is reduced from 5 to 3. The packing of variables helps in two ways. First a and d can be accessed in sequence without modifying the contents of the address register as they belong to the same storage location. Second since a and d are both simultaneously made adjacent to variables c and f, it is possible to handle access sequences af and df using autoincrement and access sequences ac and dc using autodecrement.

Problem Formulation

Now let us further discuss the formulation of the layout determination problem based upon which the above layouts are derived. First let us consider the formulation of SOA proposed by Liao et al. An undirected edge-weighted access graph is constructed from the access sequence where the vertices of the graph are the variables and the weights of the edges are the number of times in the static code that two variables are accessed in sequence. Figure 1(e) shows the access graph of this example. 16 bit variables are shown as shaded nodes while unshaded nodes represent 32 bit variables. A layout corresponds to *path cover* of the graph. The edges included in the path cover can be handled using autoincrement and autodecrement while the edges that are not covered lead to insertion of explicit address arithmetic instructions. To minimize the insertion of additional instructions, we should select a *maximum weight path cover*. However, this problem is NP-complete and therefore a greedy heuristic was proposed by Liao et al. to find the path cover. The thick edges shown in Figure 1(e) represent the path cover generated by the greedy algorithm. Since the sum of the weights of edges that are not covered is 5, 5 explicit arithmetic instructions are inserted in the generated code. The sum of the weights of edges that are not covered is also called the *cost* of the storage layout.

In order to address the SWOA problem, the above formulation is enhanced as follows. We allow application of *node coalescing* – two nodes can be coalesced into one if they together can fit into a single word. For example, the graph shown in Figure 2(c) is obtained by coalescing nodes for variables a and d, creating the node ad. Following coalescing, if we apply the Liao et al.'s greedy heuristic, we obtain the path cover shown by the thick edges in Figure 2(c). As we can see, by coalescing we have essentially covered the edge $a - d$ and the path cover further exploits the benefits of this coalescing leaving two edges uncovered with collective weight of 3. Since the cost of the layout following path covering with coalescing is 3, 3 explicit address arithmetic instructions are generated following SWOA.

From the above discussion it is clear that we now need to solve the *Path Cover with Node Coalescing* (PCwNC) problem where coalescing operations can be applied to the graph and a path cover needs to be found for the transformed graph such that the cost of the resulting storage layout is the minimum. While in the example shown, only two variables could be coalesced together as each was 16 bits, in general more than two variables may be coalesced into a single location assuming that their combined bitwidths do not exceed 32 bits. While a single coalescing operation involves exactly two nodes, by iteratively performing coalescing, more than two variables can be made to share the same location.

Definition 1. (Node Coalescing) Given an access graph $G\langle V, E\rangle$, nodes $u, v \in V$ can be coalesced into uv if the following condition holds:

$$Bitwidth(u) + Bitwidth(v) \leq 32 \; bits.$$

The transformed graph $G'\langle V', E'\rangle$ obtained after coalescing of u and v is obtained as follows:

V':
  $V' = V - \{u, v\} \cup \{uv\}$
  $Bitwidth(u, v) = Bitwidth(u) + Bitwidth(v)$
E':
  for each node $x \in V - \{u, v\}$ **do**
    if $u - x, v - x \in E$ **then** $uv - x \in E'$ and
      $weight(uv - x) = weight(u - x) + weight(v - x)$
    elseif $u - x \in E$ and $v - x \notin E$ **then** $uv - x \in E'$ and
      $weight(uv - x) = weight(u - x)$
    elseif $v - x \in E$ and $u - x \notin E$ **then** $uv - x \in E'$ and
      $weight(uv - x) = weight(v - x)$
    elseif $x - y \in E$ and $\{x, y\} \cap \{u, v\} = \phi$ **then**
      $x - y \in E'$
    endif
  endfor

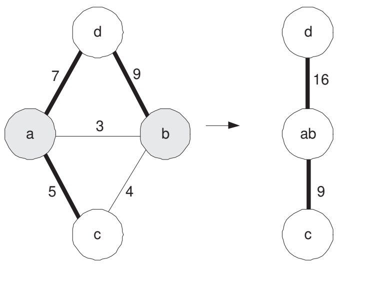
The node coalescing operation is formally defined above. The definition above specifies the precondition under which nodes u and v can be coalesced and replaced by node uv. Furthermore, edge weight $weight(x-y)$ is the number of times variables x and y are accessed one after another. To facilitate the checking of the precondition, with each node n, we associate its bitwidth $Bitwidth(n)$. The graph update following a coalescing operation is also specified.

While the problem of finding the maximum weight path cover is already known to be NP-complete, providing the flexibility to perform node coalescing further complicates the design of a heuristic algorithm. The obvious approach that comes to mind is to develop a greedy prepass for performing node coalescing. The transformed graph can then be passed on to the Liao et al.'s greedy path covering heuristic. However, there are intricate interactions between coalescing and path covering that make this approach not so attractive. In particular, coalescing is not always good – since coalescing can result in nodes with higher degrees, it can adversely effect our ability to find a good path cover. In the remainder of this section we will characterize the situations in which coalescing is beneficial and situations in which it is harmful. This analysis will guide the design of algorithms which will be presented in the subsequent section.

Positive and Negative Coalescence

Consider the example in Figure 3(a). By applying Liao's SOA algorithm, the path cover obtained includes edges $b-d$, $a-d$, and $a-c$ shown as thick edges in the graph. Edges $a-b$ and $b-c$ with a collective cost of 7 are not covered. If we coalesce a and b which are both 16 bit variables shown as shaded nodes in the graph, all edges are covered in the resulting graph. Thus the impact of coalescing is positive. On the other hand if we consider the example in Figure 4 something completely different happens. Before coalescing, edges $b-f$, $a-d$, $a-c$ and $b-e$ are covered by the path cover and as a result only one edge, $a-b$ with cost of 3, is not covered. However, after shaded nodes of 16 bit variables a and b are coalesced, two edges, $c-ab$ and $e-ab$ with collective cost of 9, are not be covered. Thus, node coalescing has a negative impact. A quick look at these examples reveals the key difference between them. In Figure 3 the nodes a and b both have degree of 3 while the degree of resulting node ab is 2 which is lower. On the other hand, in Figure 4, the nodes a and b both have a degree of 3 while the degree of ab is 4 which is higher.

Figure 3 Positive Coalescence

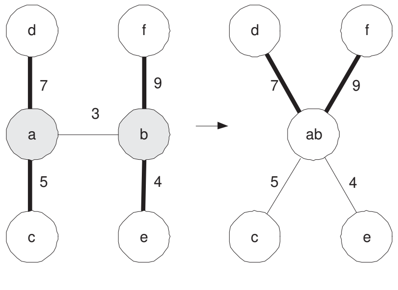


(a) Before Coalescing (b) After Coalescing

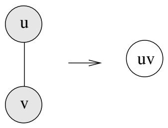
In order to characterize the possible positive or negative impact of node coalescing, it is useful to consider the impact of coalescing two nodes on the edges connected to these nodes by dividing them into the following three categories.

1. *Edge between coalesced nodes.* This edge will be removed by node coalescing and thus this is a positive effect of node coalescing. The bigger the weight of the edge between a pair of coalesced nodes, the greater is the benefit of eliminating the edge through coalescing. However, this is not the only effect of coalescing. We must also consider other effects of coalescing as described next.

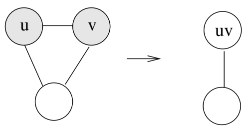
Figure 4 Negative Coalescence



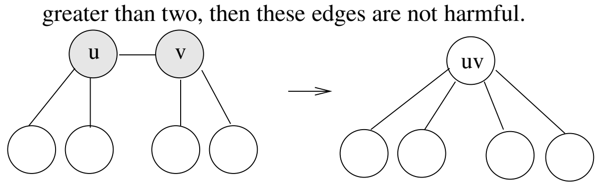
(a) Before Coalescing (b) After Coalescing



2. *Edges between coalesced nodes and their common neighbors.* By node coalescing, the two edges between nodes being coalesced and their common neighbor will become merged into one edge. The weight of the new edge is the sum of the two original edges. Since one edge each connected to the original nodes is replaced by exactly one edge connected to the coalesced node, such edges are not a contributing factor to the formation of a coalesced node with a higher degree than the degrees of the nodes before coalescing. Thus, these edges are not considered harmful.



3. *Edges between coalesced nodes and their non-common neighbors.* It is the presence of these edges that results in the creation of a coalesced node which has a degree higher than the degrees of the original nodes. Since only two of the neighbors of a node can ever be in the path cover, the amount by which the degree of a coalesced node exceeds two is the number of edges connected to the coalesced node that will not be covered. However, if the degree of the coalesced node is not greater than two, then these edges are not harmful.



The above qualitative analysis sheds light on the factors which should influence coalescing decisions. We should maximize the positive factors and avoid the negative factors while doing node coalescing. In the next section, we develop algorithms for incorporating coalescing into path covering.

3. ALGORITHMS

We have developed three algorithms to solve the PCwNC problem. The discussion below outlines the motivation and differences in the three algorithms.

Alg. I - Cover First, Coalesce Later. This algorithm first uses Liao et al.'s greedy algorithm to find a path cover and then performs coalescing to handle as many of the uncovered edges as possible. The advantage of this approach is that coalescing will only improve the solution produced by Liao et al.'s algorithm. Each coalescing operation will eliminate at least one uncovered edge. The drawback of this approach is that it is conservative in that, while it completely avoids the negative impact of coalescing, it does not fully exploit the positive potential of coalescing.

Alg. II - Coalesce First, Cover Later. This algorithm first carries out all the coalescing operations, and then applies Liao et al.'s algorithm for finding the path cover for the transformed graph. The advantage of this approach is that when the path cover is selected, we can take advantage of the coalescing decisions made earlier. In contrast to the first algorithm, this algorithm is more aggressive in carrying out coalescing and thus it can potentially lead to coverage of more edges.

Alg. III - Integrated Covering and Coalescing. There is an inherent interaction between coalescing decisions and path covering decisions. If path cover is computed first, as is the case in Alg. I, then the knowledge of this cover is exploited in carrying out node coalescing decisions. If coalescing is performed first, as is the case in Alg. II, then the knowledge of these decisions is exploited during path covering decisions. This algorithm interleaves covering and coalescing decisions so that both types of decisions have an opportunity to influence each other. This integrated algorithm strikes a balance between the two algorithms. We observe that Alg. I suffers when it performs too little coalescing and Alg. II suffers when it performs too much coalescing. Alg. III attempts to perform just the right amount of coalescing.

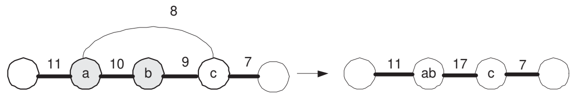
Next we discuss the three algorithms in complete detail.

3.1 Alg. I - Cover First, Coalesce Later

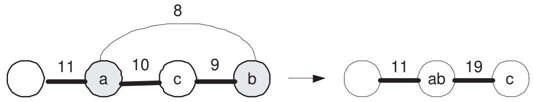
As the name implies, this algorithm first finds a path cover using the algorithm by Liao et al. [15] and then it carries out node coalescing to merge as many of the uncovered edges as possible into the path cover. This approach guarantees that coalescing only improves upon the solution provided by the initial path cover. However, in this approach, the path cover cannot be arbitrarily altered and thus coalescing decisions cannot be used to select a different and perhaps superior path cover.

The key issue we need to address is the identification of coalescing opportunities that eliminate an uncovered edge. We have identified four distinct situations in which coalescing is possible assuming that the nodes being coalesced satisfy the bitwidth precondition. These four cases are as follows:

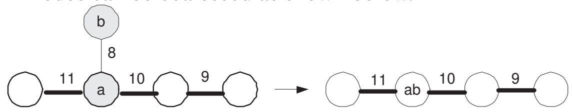
Case 1: In this case both of the nodes that are connected by an uncovered edge lie in the same covered path and there is only one node in between them. If any one of the nodes' from the uncovered edge can be coalesced with the intervening node, the uncovered edge will merge into the path cover and thus would be effectively covered. This situation is illustrated in the figure below. The shaded nodes in the figure represent nodes that can be coalesced.



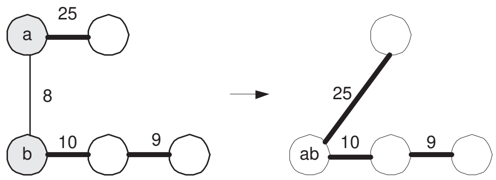
Case 2: In this case also both of the nodes that are connected by an uncovered edge belong to the same covered path and there is only one node in between them. However, in addition one of the endpoints of the uncovered edge is at the end of a path cover. In this case the two nodes connecting the uncovered edge may be coalesced together as shown in the figure below.



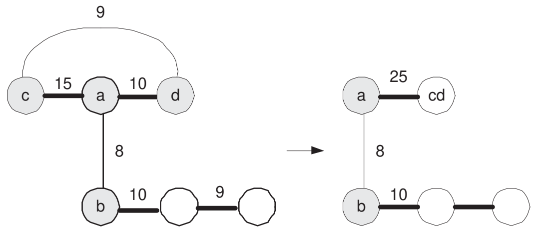
Case 3: In this case, for an uncovered edge, if one node is left by itself and the other node is part of a path cover, then the two nodes can be coalesced as shown below.



Case 4: In the fourth case, for an uncovered edge, if both of the nodes are the endpoints of a covered path, then we can coalesce the nodes. Actually this will connect the two covered subpaths. This situation is illustrated below.



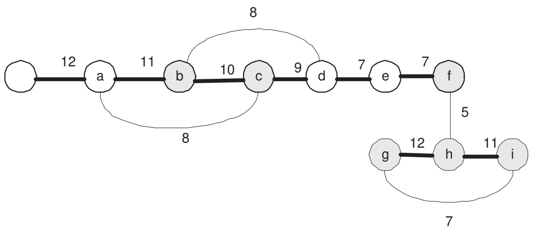
However, it should be noted that the above situation cannot occur when the initial path cover is examined. This is because Liao et al.'s algorithm would have simply included this uncovered edge into the path cover. But this situation can be created after application of other coalescing transformations. In the figure below, after coalescing of nodes c and d according to case 2, an opportunity for coalescing nodes a and b according to case 4 arises.



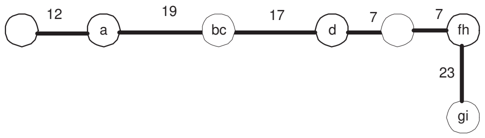
An example of the application of a sequence of above coalescing transformations is shown in Figure 5. Figure 5(a) gives the original access graph and the thick edges form the path cover identified using Liao's heuristic. The cost of this cover is 28. In this graph the shaded nodes correspond to 16 bit variables and unshaded nodes are assumed to correspond to 32 bit variables. Using case 1 nodes a and b are coalesced, by applying case 2 nodes c and d are coalesced, and by applying case 4 nodes e and f are coalesced. The resulting graph is shown in Figure 5(b). The cost of this cover is reduced to 0 as all edges have either disappeared due to coalescing or covered by the path cover.

Figure 3.1 summarizes this algorithm. First, we compute path cover PC as a normal SOA solution of the access graph. The un-

Figure 5 An Example.



(a) Original graph and its cover.



(b) Coalesced graph and its cover.

covered edges are sorted in decreasing order of their weights and then processed one by one. In each step an attempt is made to handle the uncovered edge using coalescing operations defined by the four cases discussed. If coalescing is successful at any step, bookkeeping actions are performed to update the path cover and the access graph. The path cover PC and the log of coalescing actions NC together define the SWOA solution A.

3.2 Alg. II - Coalesce First, Cover Later

The drawback of Alg. I is that it sometimes does too little coalescing, i.e. it fails to carry out coalescing that can reduce the cost of SWOA further. Consider the access graph shown in Figure 7(a). As before the shaded nodes represent nodes that can be coalesced together. If we consider the four cases under which Alg. I performs coalescing, none of the four cases are applicable to nodes a and b in the Figure 7(a). Thus, Alg. I terminates with the path cover shown by thick edges in Figure 7(a). On the other hand lets assume that we coalesce the two nodes first giving the access graph shown in Figure 7(b). Now we find the path cover for this transformed graph, as shown by the thick edges. The cost of SWOA is reduced from 50 without coalescing to 45 with coalescing.

The above example shows the limitations of the Alg. I. The path cover is found first and only node coalescings that preserve the path cover are applied. In the example shown above the node coalescing performed does not preserve the path cover of Figure 7(a) and thus after coalescing a fresh path cover was computed. This path cover had a lower cost. This observation suggests another approach which forms the basis of our second algorithm. We carry out aggressive node coalescing first, and after all coalescings have been performed, the path cover is computed. Thus, during coalescing, there is no path cover implied restrictions on coalescing.

While it is clear that this time we will use the coalesce first and cover later policy, we still must devise a method for deciding when two nodes should be coalesced. We cannot simply exploit all opportunities for coalescing because as discussed in section 2, coalescing can sometimes be harmful as it may create high degree nodes. To evaluate a particular opportunity for coalescing, we formulate a notion of *locally beneficial* coalescing as follows. Given nodes u and v which are candidates for coalescing, we consider the *local subgraph* of the access sequence graph such that nodes in this subgraph are u, v, and all nodes connected by an edge to u and/or v while the edges in the subgraph are all edges emanating from nodes

Figure 6: Alg. I - Cover First, Coalesce Later.

input : A set N of nodes with bitwidth information.
An access sequence L for these nodes.
output : A subword offset assignment A.

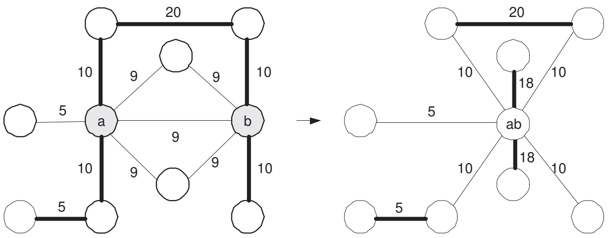
$G(V,E) \leftarrow$ `AccessGraph(`L,N`)`;
$NC \leftarrow \emptyset$;
$PC(V',E') \leftarrow$ `SolveSOA(`G`)`;
$F \leftarrow$ list of edges in $E - E'$ in descending order of weight;
while $F \neq \emptyset$ **do**
  $e(n_1, n_2) \leftarrow$ first edge in F;
  $F \leftarrow F - \{e\}$;
  $s \leftarrow$ `DetectCase(`e, G`)`;
  switch s **do**
    case *1*
      $n_3 \leftarrow$ the node between n_1 and n_2;
      if `Bitwidth(`n_1`)+Bitwidth(`n_3`)` $\leq$ *32 bits*
      then
        $G \leftarrow$ `Coalesce((`n_1, n_3`),`G`)`;
      else
        $G \leftarrow$ `Coalesce((`n_2, n_3`),`G`)`;
      end
      update PC;
      update F;
      record this coalescing in log NC;
      break;
    case *2,3 or 4*
      $G \leftarrow$ `Coalesce((`n_1, n_2`),`G`)`;
      update PC;
      update F;
      record this coalescing in log NC;
      break;
    otherwise
      break;
  end
end
$A \leftarrow$ build SWOA from PC and NC;
return A;

u and v. We find a cover for the local subgraph and then we coalesce nodes u and v in the local subgraph and find a cover for this transformed subgraph. If the cost of the cover of the transformed local subgraph is lower than the cost of the untransformed local subgraph, then we consider the coalescing to be *locally beneficial*. If a coalescing opportunity is found to be locally beneficial, we coalesce the two nodes involved; otherwise we do not take advantage of the coalescing opportunity.

It should be noted that the covers of the transformed and untransformed local subgraph that are found above are merely used to make the coalescing decision. We do not commit to any covering decisions. The above process is applied repeatedly to carry out as much coalescing as possible. Once no more opportunities of locally beneficial coalescing exist, we compute the path cover for the transformed graph using Liao et al.'s algorithm.

In Figure 8(a) the local subgraph corresponding to nodes a and b of Figure 7(a) is shown. The cost of the cover for this local subgraph is 50. After coalescing nodes a and b the graph that results is the one in Figure 8(b) and the cost of the cover for this subgraph is 45. Therefore, this coalescing opportunity is locally beneficial and we take advantage of it. Following coalescing the access graph obtained is the one in Figure 7(b) and by applying Liao's algorithm we find its cover which is also shown in Figure 7(b).

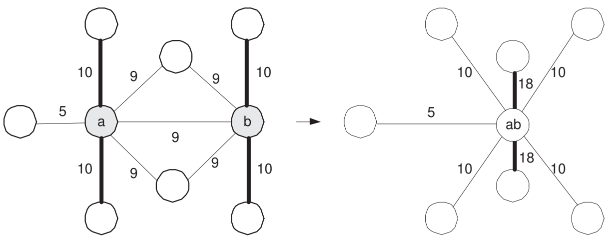
Figure 7 Motivating example.



(a) Original Path Cover (Cost = 50)

(b) Path Cover on Coalesced Graph (Cost = 45)

Figure 8 Local subgraph: path cover before and after coalescing.



(a) Local Subgraph (Cost = 50)

(b) Coalesced Local Subgraph (Cost = 45)

In the remainder of this section we formally define the notion of locally beneficial and present the pseudo code for this algorithm.

Definition 2. (Local Subgraph) Given coalescing candidate nodes u and v in access sequence graph $G\langle V, E\rangle$. The *local subgraph* for u and v, denoted as $LG(u,v) = \langle V', E'\rangle$, is defined as:
$V' = \{u, v\} \cup \{n : \ n \in V \ and \ n - u \ or \ n - v \ \in \ E\}$ and $E' = \{a - b : \ a - b \in E \ and \ \{a, b\} \cap \{u, v\} \neq \phi\}$.

Definition 3. (Locally Beneficial Coalescing) Given the local subgraph $LG(u, v)$, if the cost of path cover for LG(u,v) (also referred to as the *precost*), is higher than the cost of path cover for local subgraph obtained after coalescing u and v (also referred to as the *postcost*), then coalescing of u and v is said to be *locally beneficial*.

Figure 9 is the pseudo-code of this algorithm. First, we identify all those edges that connect nodes that can be coalesced. Then we go through these edges one by one and apply the *locally beneficial* test to them. In this process some edges may be collapsed through coalescing while other may be left as is. Once coalescing is over, we apply Liao's algorithm to find the path cover for the resulting graph giving us the SWOA solution.

3.3 Alg. III - Integrated Covering and Coalescing

In preceding sections we have shown that while conservative coalescing performed by Alg. I can sometimes lead to inferior solutions, Alg. II performs coalescing more aggressively to achieve better performance. Next we show that in some situations Alg. II may perform too much coalescing leading to a worse solution than Alg. I. Consider the example shown in Figure 10. Solution generated by Liao et al.'s algorithm, which does not coalesce at all, leads to the path cover with a cost of 68 as shown in Figure 10(a). There are four coalescing opportunities in this graph. Alg. I exploits one of these opportunities leading to the solution in Figure 10(b) which has a cost of 48. Alg. II exploits all four coalescing opportunities leading to the solution in Figure 10(c) which has a cost of 51. In other words, Alg. II performs too much coalescing in this case.

So we can see that in some situations Alg. I performs better than Alg. II while in other situations Alg. II performs better than Alg. I. In fact, looking further into the above example, we can see that there is another solution that is better than the solutions produced by Alg. I and Alg. II. This solution takes advantage of two of the four coalescing opportunities as shown in Figure 10(d) leading to a solution with a cost of only 40. In other words, sometimes Alg. I performs too little coalescing while Alg. II performs too much coalescing. To get the best result we need to strike a balance between the two algorithms.

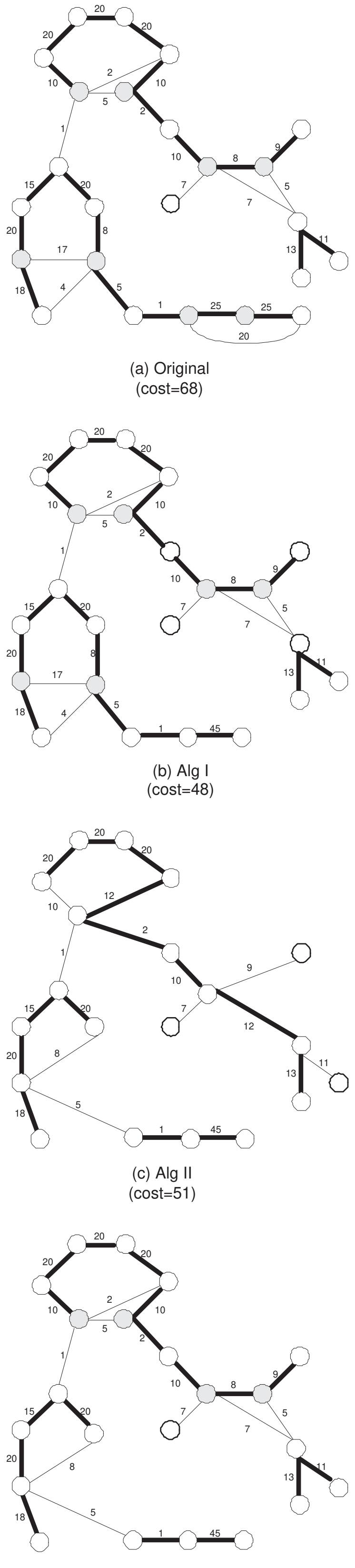
The real reason for the observed behaviors of Alg. I and Alg. II is as follows. There is an inherent interaction between coalescing decisions and path covering decisions. If node coalescing is performed after the path cover has been computed, as is the case in Alg. I, then the knowledge of coalescing decisions cannot influence the path covering decisions. If path covering is carried out after coalescing has been performed, as is the case in Alg. II, then the knowledge of the path covering decisions cannot influence coalescing decisions. If we want node coalescing and path covering decisions to influence each other, we need an integrated algorithm that interleaves covering and coalescing decisions. Our Alg. III is one such algorithm which produces the result of Figure 10(d).

The integrated algorithm we propose prepares a sorted list of edges in the decreasing order of the edge weights. Then it runs through the edges one by one, where in each step, the algorithm de-

Figure 9: Alg. II - Coalesce First, Cover Later.

input : A set N of nodes with bitwidth information.
An access sequence L for these nodes.
output : A subword offset assignment A.
$G(V, E) \leftarrow$ `AccessGraph(`L,N`)` ;
$F \leftarrow \emptyset$;
$NC \leftarrow \emptyset$;
for *all edges* $e(n_1, n_2) \in V$ **do**
    if `Bitwidth(`n_1`)+Bitwidth(`n_2`)` $\leq$ *32 bits* **then**
        $F \leftarrow F \cup \{e\}$;
    end
end
sort the edges in F in descending order of weight;
while $F \neq \emptyset$ **do**
    $e(n_1, n_2) \leftarrow$ first edge from F ;
    $G'(V', E') \leftarrow$ `LocalSubgraph(`e,G`)` ;
    $precost \leftarrow$ `ComputePreCost(`e,G'`)` ;
    $G''(V'', E'') \leftarrow$ `Coalesce(`e,G'`)`;
    $postcost \leftarrow$ `ComputePostCost(`G''`)` ;
    if $precost - postcost > 0$ **then**
        $G \leftarrow$ `Coalesce(`e, G`)`;
        record the coalescing information in the log NC;
        update F;
    else
        $F \leftarrow F - \{e\}$;
    end
end
$PC \leftarrow$ `SolveSOA(`G`)`;
$A \leftarrow$ build SWOA from PC and NC;
return A;

Figure 10 Right amount of coalescing.



(a) Original (cost=68)

(b) Alg I (cost=48)

(c) Alg II (cost=51)

(d) Alg III (cost=40)

termines whether coalescing is possible and desirable. Coalescing is possible if the combined bitwidth condition is met and coalescing will maintain the integrity of the partial cover. When coalescing is possible, it is also desirable if it satisfies the *globally beneficial* criteria described later. If the edge is not handled through coalescing, then an attempt is made to include it into the path cover. Note that when an attempt is made to include the current edge into the partial cover, we are not guaranteed of success. This is essentially due to the same reason that sometimes Liao et al.'s greedy algorithm is not able to include an edge in the partial cover.

From the above discussion it is clear that at any point in time a partial cover exists. A coalescing decision is made based upon prior path covering decisions and a path covering decision exploits the knowledge of prior coalescing decisions. In this way we are able to tightly integrate coalescing and covering.

The notion of *globally beneficial* coalescing is an extension of *locally beneficial* criteria. Given a pair of nodes that can be coalesced while maintaining the current partial cover, we identify the *global subgraph* as the union of the *local subgraph* and the *partial cover* which is the cover that has already been built. We find the complete path cover of this subgraph by extending the current partial cover that is a part of it and then determine the cost of the cover and refer to the cost as the *precost*. Next we coalesce the nodes and find a complete cover for the transformed global subgraph by extending the current partial cover that is a part of it. We determine the cost of this cover and refer to it as the *postcost*. If the *postcost* is less than the *precost* we proceed with coalescing.

Definition 4. (Global Subgraph) Given a pair of coalescing candidate nodes u and v in access sequence graph $G\langle V, E\rangle$. The *global subgraph* for u and v, denoted as $GG(u,v) = \langle V', E'\rangle$, is the union of the *local subgraph* $LG(u,v)$ and the current *partial cover*.

Definition 5. (Globally Beneficial Coalescing) Given the global subgraph $GG(u,v)$, if the cost of path cover for GG(u,v) (also referred to as the *precost*), is higher than the cost of path cover for the global subgraph obtained after coalescing u and v (also referred to as the *postcost*), then coalescing of u and v is said to be *globally beneficial*. The path covers considered in above analysis must be consistent with the current partial cover.

Let us revisit the example of Figure 10 and show that the integrated algorithm, i.e. Alg. III, does produce the result shown in Figure 10(d). As we have seen before, there are four possible places in the graph where coalescing could be performed. The handling of the four cases is shown in Figure 11. In Figure 11 only the relevant parts of the global subgraphs are shown both before and after coalescing and their corresponding cover costs are given. The graphs contain three types of edges: thick dark edges that are part of the current partial cover; thick dashed edges which were included to form the complete cover for the global subgraph; and thin edges that are not part of the complete cover of the global subgraph. The thin edges contribute to the precost and postcost corresponding to the two covers. In the first two cases, we can see that coalescing is found to be globally beneficial. In the third case coalescing cannot be carried out because it would require a change in the current partial cover. In the final case coalescing does not yield any benefit. Therefore the first two coalescing opportunities are exploited while the other two opportunities are ignored. Thus, the results of the Alg. III correspond to the solution shown earlier in Figure 10(d).

Figure 11 Integrated coalescing.

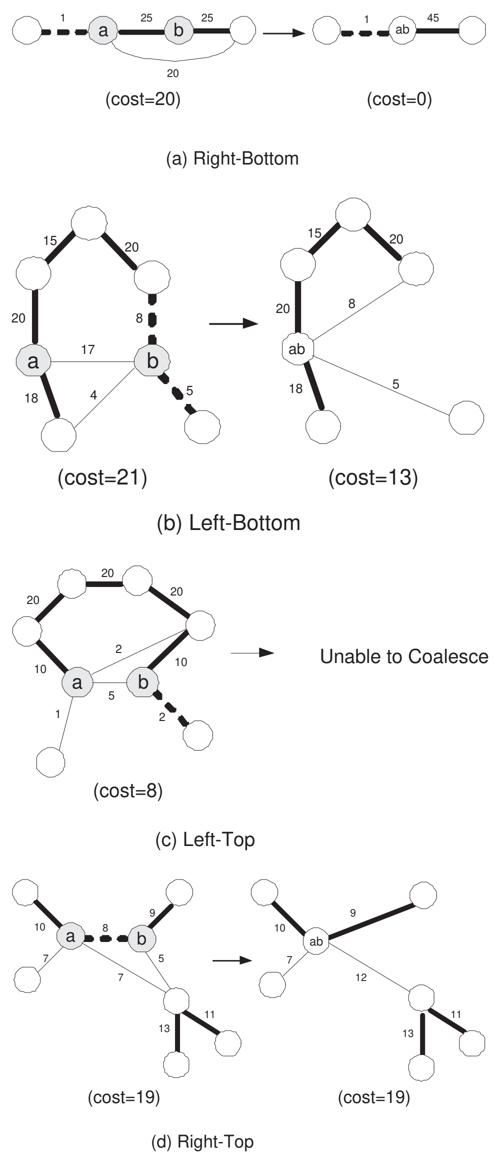


Figure 12: Alg. III - Integrated Covering and Coalescing.

input : A set N of nodes with bitwidth information.
An access sequence L for these nodes.
output : A subword offset assignment A.

$G(V, E) \leftarrow$ `AccessGraph(`L,N`)`;
$NC \leftarrow \emptyset$;
$PC(V', E') : V' \leftarrow V, E' \leftarrow \emptyset$;
$F \leftarrow$ list of edges in E in descending order of weight;
while $F \neq \emptyset$ **do**
    $e(n_1, n_2) \leftarrow$ first edge in F;
    $F \leftarrow F - \{e\}$;
    if `Bitwidth(`n_1`)+Bitwidth(`n_2`)` $\leq$ *32 bits* **then**
        if n_1 *and* n_2 *can be coalesced* **then**
            $G_1(V_1, E_1) \leftarrow$ `LocalSubgraph(`e,G`)`;
            $G_2(V_2, E_2) : V_2 \leftarrow V', E_2 \leftarrow E' \cup E_1$;
            $precost \leftarrow$ `ComputePreCost(`e,G_2`)`;
            $G_3(V_3, E_3) \leftarrow$ `Coalesce(`e,G_2`)`;
            $postcost \leftarrow$ `ComputePostCost(`G_3`)`;
            if $precost - postcost > 0$ **then**
                $G \leftarrow$ `Coalesce(`e, G`)`;
                update PC; update F;
                Record this coalescing information in log NC;
            else
                if e *can be covered* **then**
                    $E' \leftarrow E' \cup \{e\}$;
                end
            end
        end
    else
        if e *can be covered* **then**
            $E' \leftarrow E' \cup \{e\}$;
        end
    end
end
$A \leftarrow$ build SWOA from PC and NC;
return A;

Figure 12 is the pseudo-code of this algorithm. First, the partial cover is set to be empty. Then we run through the edges in the access graph in weight-descending order. First we attempt coalescing to handle the edge. In order for coalescing to occur three conditions must be true: the bitwidths must not exceed 32 bits; the current partial cover should be maintained; and finally coalescing should be globally beneficial. If any one of these conditions is not satisfied, the edge cannot be coalesced. Thus, an attempt is made to incorporate it into the partial cover by checking the same conditions that were formulated by Liao et al., i.e. no loop is formed in the partial cover after including the edge and degree of no node exceeds 2 after the edge is included in the partial cover. If neither coalescing nor covering succeeds, the edge is left uncovered.

3.4 Further Extensions

While the focus of our work has been on simple offset assignment problem, our approach can be extended to consider the global offset assignment (GOA) problem. Our SWOA can be easily used in conjunction with the variable partitioning strategy described in [15] where two variables with largest cost are selected and placed in separate partitions. However, we observe that given the bitwidth information of each variable, a better variable partitioning strategy can be devised by taking the bitwidth information into consideration. For example, if there is a pair of nodes in the access graph that can possibly be coalesced, we should avoid placing them into separate partitions. This is because only by placing them in the same partition can we leave open the possibility of coalescing. In other words a partitioning strategy should also consider interactions with coalescing and covering.

Access sequence reordering optimization, either by code scheduling or algebraic transformation on expression trees, can also be combined with subword offset assignment. We must consider that different access sequences may lead to different access graphs with different number of possible node coalescings. Intuitively, reordering should arrange variables that can be coalesced such that they are accessed in sequence so that the cost of edges corresponding to them can be removed by coalescing.

4. EXPERIMENTAL RESULTS

The tool we used for experimentation is the Motorola's gcc-based DSP56000 C compiler with source code available in public domain. We have implemented our three algorithms and Liao's greedy algorithm in order to evaluate this work. We use the latter algorithm as the baseline algorithm to measure the effectiveness of the proposed algorithms. Our implementations focus on local variables and optimize stack frame memory accesses. For implementing our algorithms we obtained the bitwidth of each variable by the bitwidth analysis techniques developed in our prior work [22]. The applications considered were taken from the Mediabench [11] (`adpcm`, `g721`, and `mpeg2`) and Bitwise Project at MIT (`softfloat`) [20].

In general, a good storage layout has two positive effects: it reduces code size and it also improves performance. We carried out two experiments. In the first experiment we give higher priority to reducing code size and lower priority to improving performance while in the second experiment we reversed the priorities. In the first experiment access graphs as described in our discussion were used, i.e. the weight of an edge between two variables was the number of times two variables were observed as being accessed in sequence in the static code. This approach gives priority to reducing the static number of explicit address arithmetic instructions. In the second experiment we used profile information to determine weights as being the dynamic number of times two variables are accessed in sequence. Clearly this approach gives greater priority to reducing dynamic count of explicit address arithmetic instructions.

We also report the reduction of the stack frame size due to variable packing in the experiments. Node coalescing causes multiple subword variables to be packed into one storage location. Since the candidates for our optimization are the local variables and temporaries located in stack frame, stack frame size can be reduced. Smaller stack frame size directly leads to smaller memory footprint at runtime. This, together with static code size reduction, further reduces the memory needs of an application.

Static Code Size

In Table 1 we report the percentage reduction in the number of the explicit address arithmetic instructions that our algorithms achieve over the Liao et al. algorithm. When code size reduction is given higher priority, our algorithms achieve on average 26%, 26.9% and 32% reduction in the number of explicit address arithmetic instructions. As the results show the first two algorithms can outperform each other in different situations. However, the third algorithm significantly outperforms the other two algorithms in many cases and does at least as good as the best of the other two algorithms in other cases. Thus, we conclude that interleaving covering and coalescing decisions is the best approach. When the reduction in dynamic instruction count of explicit address arithmetic instructions is given a higher priority, we observe an average reduction of 28.5%, 26.5%, and 31.4% in the explicit address arithmetic instructions.

Stack Frame Size

In Table 2 we report the reduction of the size of stack frame. In the experiments when code size reduction has higher priority, our algorithms achieve on average 14.5%, 22.1% and 22.7% reduction in stack frame size. In the experiments when higher priority is given to dynamic instruction count reduction, our algorithms achieve on average 16.4%, 19.5% and 21.7% reduction. The effects of this reduction can be combined with the reduction of static code size as to the memory requirement of an application. From the results, we can see that the second and third algorithms achieve more reduction in the stack frame size. This is because of the first algorithm takes a path cover first policy and reduces the number of possible node coalescing.

Dynamic Instruction Count

In Table 3 we report the percentage reduction in the number of dynamically executed explicit address arithmetic instructions. When reducing dynamic instruction count is given a higher priority, our algorithms achieve on average 24.3%, 24.7%, and 30.2% reduction in executed explicit address arithmetic instructions over Liao's algorithm. As expected, the third algorithm performs the best overall. When the reduction in static code size of explicit address arithmetic instructions is given a higher priority, we observe an average reduction of 21.6%, 26.6%, and 29.5% in the dynamic instruction count of explicit address arithmetic instructions executed by our three algorithms.

From the above results it is clear that significant reductions in static and dynamic instruction count are achieved over Liao et al.'s algorithm by all three of our proposed algorithms. Our algorithms also achieve significant reductions in stack frame size. The integrated algorithm, Alg. III, gives the best performance.

5. CONCLUSION

In this paper we demonstrated that the combination of subword referencing and indirect addressing with autoincrement and autodecrement is effective in reducing code size and stack frame size. By allowing packing of multiple subword variables into a single memory location, storage layouts are obtained which both reduce the need for address arithmetic and increase the use of autoincrement and autodecrement when address arithmetic is needed. We showed how variable packing can be expressed in terms of node coalescing operations on access graph. The interactions between node coalescing and path cover selection were studied and three algorithms were developed to solve these problems together. Our experimental results show that these three algorithms provide significant reductions in code size over Liao et al.'s algorithm. The integrated algorithm that interleaves coalescing and covering decisions performs the best.

Acknowledgements

This work is supported by National Science Foundation grants CCR-0324969, CCR-0220334, CCR-0208756, CCR-0105355 and EIA-0080123, and grants from Intel and IBM to the University of Arizona.

6. REFERENCES

[1] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson, "The next generation of Intel IXP network processors," *Intel Technology Journal*, Vol. 6, No. 3, pages 6-18, August 2002.

[2] D.H. Bartley, "Optimizing stack frame accesses for processors with restricted addressing modes," *Software Practice and Experience* (SP&E), Vol. 22, No. 2, pages 101-110, February. 1992.

[3] W-K. Chen, B. Li, and R. Gupta, "Code compaction of matching single-entry multiple-exit regions," *10th Annual International Static Analysis Symposium* (SAS), *LNCS 2694*, pages 401-417, San Diego, CA, June 2003.

[4] Y. Choi and T. Kim, "Address assignment combined with scheduling in DSP code generation," *ACM/IEEE Design Automation Conference* (DAC), pages 225-230, New Orleans, LA, June 2002.

Table 1: Reduction in Static Code Size.

Benchmark Function	Priority – Code Size Priority – Dynamic Instruction Count						
	SOA	Alg I		Alg II		Alg III	
	#cost	#cost	[%]	#cost	[%]	#cost	[%]
adpcm							
coder	27	27	0.0	23	14.8	21	22.2
	28	24	14.3	24	14.3	22	21.4
decoder	21	16	23.8	14	33.3	14	33.3
	23	16	30.4	14	40.9	14	40.9
g721							
encoder	7	5	28.6	5	28.6	5	28.6
	7	5	28.6	5	28.6	5	28.6
decoder	8	3	62.5	5	37.5	3	62.5
	9	5	55.6	6	33.3	5	55.6
fmult	3	1	66.7	1	66.7	1	66.7
	5	2	66.7	1	66.7	1	66.7
quantize	0	0	0.0	0	0.0	0	0.0
	0	0	0.0	0	0.0	0	0.0
update	15	8	46.7	5	66.7	5	66.7
	15	6	60.0	6	60.0	6	60.0
mpeg2							
frametotc	1	1	0.0	1	0.0	1	0.0
	1	1	0.0	1	0.0	1	0.0
formpred	10	9	10.0	9	10.0	9	10.0
	10	9	10.0	9	10.0	9	10.0
softfloat							
m32i_64	12	8	33.3	10	16.7	8	33.3
	12	8	33.3	10	16.7	8	33.3
f32_i32	14	12	14.3	11	21.4	10	28.6
	14	12	14.3	11	21.4	10	28.6
Average			26.0		26.9		32.0
			28.5		26.5		31.4

Table 2: Reduction in Stack Frame Size

Benchmark Function	Priority – Code Size Priority – Dynamic Instruction Count						
	SOA	Alg I		Alg II		Alg III	
	#size	#size	[%]	#size	[%]	#size	[%]
adpcm							
coder	12	12	0.0	10	16.7	10	16.7
	12	11	8.3	11	8.3	10	16.7
decoder	10	8	20.0	7	30.0	7	30.0
	10	8	20.0	7	30.0	7	30.0
g721							
encoder	9	7	22.2	6	33.3	5	44.4
	9	7	22.2	6	33.3	6	33.3
decoder	8	7	12.5	6	25.0	6	25.0
	8	6	25.0	7	12.5	6	25.0
fmult	6	4	33.3	3	50.0	4	33.3
	6	4	33.3	3	50.0	4	33.3
quantize	6	6	0.0	6	0.0	6	0.0
	6	6	0.0	6	0.0	6	0.0
update	14	11	21.4	9	35.7	9	35.7
	14	11	21.4	10	28.6	9	35.7
mpeg2							
frametotc	6	6	0.0	6	0.0	6	0.0
	6	6	0.0	6	0.0	6	0.0
formpred	9	8	11.1	8	11.1	8	11.1
	9	8	11.1	8	11.1	8	11.1
softfloat							
m32_64	8	6	25.0	7	12.5	6	25.0
	8	6	25.0	7	12.5	6	25.0
f32_i32	7	6	14.3	5	28.6	5	28.6
	7	6	14.3	5	28.6	5	28.6
Average			14.5		22.1		22.7
			16.4		19.5		21.7

Table 3: Reduction in Dynamic Instruction Count

Benchmark Function	Priority – Dynamic Instruction Count Priority – Code Size						
	SOA	Alg I		Alg II		Alg III	
	#cost	#cost	[%]	#cost	[%]	#cost	[%]
adpcm							
coder	1677709	1489644	11.2	1489644	11.2	1363256	18.7
	2169286	2169286	0.0	1874394	13.6	1710628	21.1
decoder	1702288	1702288	0.0	1185524	30.4	1185524	30.4
	1719386	1497810	12.9	1185524	31.0	1185524	31.0
g721							
encoder	1610	1150	28.6	1150	28.6	1150	28.6
	1610	1150	28.6	1150	28.6	1150	28.6
decoder	885120	442560	50.0	590080	33.3	442560	50.0
	885120	442560	50.0	442560	50.0	442560	50.0
fmult	5528	1840	66.7	1840	66.7	1840	66.7
	3688	1840	50.1	1840	50.1	1840	50.1
quantize	0	0	0.0	0	0.0	0	0.0
	0	0	0.0	0	0.0	0	0.0
update	1770	920	48.0	920	48.0	690	61.0
	2760	1840	33.3	920	66.7	920	66.7
mpeg2							
frametotc	1	1	0.0	1	0.0	1	0.0
	1	1	0.0	1	0.0	1	0.0
formpred	54534	46296	15.1	46296	15.1	46296	15.1
	54534	46296	15.1	46296	15.1	46296	15.1
softfloat							
m32_64	12	8	33.3	10	16.7	8	33.3
	12	8	33.3	10	16.7	8	33.3
f32_i32	14	12	14.3	11	21.4	10	28.6
	14	12	14.3	11	21.4	10	28.6
Average			24.3		24.7		30.2
			21.6		26.6		29.5

[5] C. Fraser, E. Myers, and A. Wendt, “Analyzing and compressing assembly code,” *ACM SIGPLAN Symposium on Compiler Construction* (SCC), pages 117-121, Montreal, Canada, June 1984.

[6] R. Gupta, E. Mehofer, and Y. Zhang, “A representation for bit section based analysis and optimization,” *International Conference on Compiler Construction* (CC), LNCS 2304, Springer Verlag, pages 62-77, Grenoble, France, April 2002.

[7] M. Kandemir, M.J. Irwin, G. Chen, and J. Ramanujam, “Address register assignment for reducing code size,” *International Conference on Compiler Construction* (CC), LNCS 2622, Springer Verlag, pages 273-289, Warsaw , Poland, April 2003.

[8] A. Krishnaswamy and R. Gupta, “Profile guided selection of ARM and Thumb instructions,” *ACM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems* (LCTES-SCOPES), pages 55-63, Berlin, Germany, June 2002.

[9] A. Krishnaswamy and R. Gupta, “Enhancing the performance of 16-bit code using augmenting instructions,” *ACM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems* (LCTES), pages 254-264, San Diego, CA, June 2003.

[10] A. Krishnaswamy and R. Gupta, “Mixed width instruction sets,” *Communications of the ACM* (CACM), in special section on Program Compaction, Vol. 46, No. 8, pages 47-52, August 2003.

[11] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, “Mediabench: A tool for evaluating and synthesizing multimedia and communications systems,” *IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 330-335, Research Triangle Park, NC, December 1997.

[12] R. Leupers and P. Marwedel, “Algorithms for address assignment in DSP code generation,” *International Conference on Computer-Aided Design* (ICCAD), pages 109-112, San Jose, CA, November 1996.

[13] R. Leupers and F. David, “A uniform optimization technique for offset assignment problems,” *International Symposium on Systems Synthesis* (ISSS), pages 3-8, Hsinchu, Taiwan, December 1998.

[14] B. Li and R. Gupta, “Bit section instruction set extension of ARM for embedded applications,” *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (CASES), pages 69-78, Grenoble, France, October 2002.

[15] S. Liao, S. Devadas, K. Keutzer, S. Tijiang, and A. Wang, “Storage assignment to decrease code size,” *ACM Transactions on Programming Languages and Systems* (TOPLAS), Vol. 18, No. 3, pages 235-253, May 1996.

[16] S. Liao, S. Devadas, K. Keutzer, S. Tijiang, and A. Wang, “Storage assignment to decrease code size,” *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 186-195, La Jolla, CA, June 1995.

[17] G. Memik, B. Mangione-Smith, and W. Hu, “NetBench: A benchmarking suite for network processors,” *IEEE International Conference Computer-Aided Design* (ICCAD), pages 39-42, San Jose, CA, November 2001.

[18] X. Nie, L. Gazsi, F. Engel, and G. Fettweis, “A new network processor architecture for high speed communications,” *IEEE Workshop on Signal Processing Systems* (SiPS), pages 548-557, Taipei, Taiwan, October 1999.

[19] A. Rao and S. Pande, “Storage assignment optimizations to generate compact and efficient code on embedded DSPs,” *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 128-138, Atlanta, GA, May 1999.

[20] M. Stephenson, J. Babb, and S. Amarasinghe, “Bitwidth analysis with application to silicon compilation,” *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 108-120, Vancouver B.C., Canada, June 2000.

[21] A. Sudarsanam, S. Liao, and S. Devadas, “Analysis and evaluation of address arithmetic capabilities in custom DSP architectures,” *ACM/IEEE Design Automation Conference* (DAC), pages 287-292, Anaheim, CA, June 1997.

[22] S. Tallam and R. Gupta, “Bitwidth aware global register allocation,” *30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL), pages 85-96, New Orleans, LA, January 2003.

[23] S. Udayanarayanan and C. Chakrabarti, “Address code generation for digital signal processors,” *ACM/IEEE Design Automation Conference* (DAC), pages 353-358, Las Vegas, NV, June 2001.

[24] J. Wagner and R. Leupers, “C compiler design for an industrial network processor,” *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (LCTES), pages 155-164, Snowbird, UT, June 2001.

[25] T. Wolf and M. Franklin, “Commbench - A telecommunications benchmark for network processors,” *IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS), pages 154-162, Austin, TX, April 2000.

[26] X. Zhuang, C. Lau, and S. Pande, “Storage assignment optimizations through variable coalescence for embedded processors,” *ACM SIGPLAN Conference on Languages, Compiler, and Tools for Embedded Systems* (LCTES), pages 220-231, San Diego, CA, June 2003