Address Register Assignment for Reducing Code Size

M. Kandemir[1], M.J. Irwin[1], G. Chen[1], and J. Ramanujam[2]

[1] CSE Department
Pennsylvania State University
University Park, PA 16802
{kandemir,mji,guilchen}@cse.psu.edu
[2] ECE Department
Louisiana State University
Baton Rouge, LA 70803
jxr@ee.lsu.edu

Abstract. In DSP processors, minimizing the amount of address calculations is critical for reducing code size and improving performance since studies of programs have shown that instructions that manipulate address registers constitute a significant portion of the overall instruction count (up to 55%). This work presents a compiler-based optimization strategy to reduce the code size in embedded systems. Our strategy maximizes the use of indirect addressing modes with post-increment and post-decrement capabilities available in DSP processors. These modes can be exploited by ensuring that successive references to variables access consecutive memory locations. To achieve this spatial locality, our approach uses both access pattern modification (program code restructuring) and memory storage reordering (data layout restructuring).

1 Introduction

Address calculations play a key role in determining code quality in DSP processors since instructions that manipulate address registers constitute a significant portion of overall instruction count. For example, it was found that for a set of codes from MediaBench suite (a popular benchmark suite for embedded systems) running on Motorola's DSP56000 processor, nearly 55% of the instructions are used to manipulate address registers through explicit loads and stores [15]. Consequently, optimizing address code generation by eliminating as many explicit address register loads as possible can result in significant improvements in code size and performance. Note that code size improvements are very important not only because code size directly determines the capacity of the customized instruction memory (hence, its cost) in an embedded system, but also because a smaller instruction memory means lower power consumption.

Address calculations in modern DSPs such as NEC 7701, Motorola DSP56000, Analog Devices ADSP21xx, and Texas Instruments TMS320C5x are done in address generation units (AGUs). An AGU contains a number of

G. Hedin (Ed.): CC 2003, LNCS 2622, pp. 273–289, 2003.

address registers, the contents of which can be incremented or decremented in parallel with the ongoing activity in the main datapath. The instruction format for such processors allows one to encode a CPU activity and a post-increment/decrement of an address register in a single instruction. Thus, using post-increment/decrement operations instead of explicit address register loads enhances on-chip parallelism (performance) and reduces code size (as no separate instruction is necessary to update the address register). Cintra and Araujo [3] report that although some of the register increment/decrement operations can be accommodated in VLIW instruction slots, modern VLIW DSP architectures also have auto-increment and auto-decrement modes; this is because exploiting these modes effectively saves one instruction slot which might be used for some other operation.

An optimizing compiler can exploit these post-increment/decrement operations by performing computation and data transformations as well as by assigning variables to address registers optimally. Consider the following scenario where three scalar variables `c`, `a`, and `b` are to be accessed in the order `c,a,b` in a given DSP code. Also assume that the AGU in question has a single address register that can be post-incremented/decremented by 1 and that these three variables are stored in memory in the order `a`, `b`, `c`. The code for implementing this sequence of accesses uses three steps. The first step loads the address register with the address of `c` (the first variable in the access sequence). To access the variable `a` next, the second step loads the address of `a` into the address register. In accessing the variable `a`, a post-increment operation can be used to modify the content of the address register so that it points to `b` which will be accessed next. In the final step, the variable `b` is accessed. Overall, we need to perform two explicit address register loads. In addition to being a waste of machine cycles, this increases code size and thereby the instruction memory size, which is at a premium in many embedded designs.

We can reduce this overhead of explicitly updating the address register by using a better choice of the order in which the variables are stored in data memory. Instead of the storage order `a`, `b`, `c` in the previous scenario, we can eliminate one of the two address register loads if we use the storage order `c`, `a`, `b`. In this case, first, we load the address register with the address of `c` and post-increment the address register to make sure that, after the execution of the statement that accesses `c`, it will point to the next location (which contains `a`). Next, we access the variable `a`, and use again post-increment to make the address register point to the variable `b`. Finally, we access the variable `b`. This problem of determining the most suitable storage order of variables is called the *offset assignment problem* and has been partially addressed by Bartley [1], Liao et al. [10,11], and others (e.g., [9,15]). Basically, these solutions first determine a suitable storage order for variables and then assign address registers to these variables to minimize the number of address register loads. In essence, since we are determining the contents of the address register(s) before each variable access, this problem can also be defined as the address register assignment problem.

A major limitation of the techniques proposed so far for the address register assignment problem is that they either focus only on modifying the storage order of variables (e.g., [10,11]) or only on modifying the intra-statement access pattern using commutativity and associativity transformations (e.g., [13]). In this work, we present a framework that considers both computation-based (intra-statement and inter-statement) transformations and storage-based optimizations in a unified setting for "reducing the code size of a given application;" that is, our main objective is to save the code space. More specifically, this work makes the following contributions.

(1) It presents an algorithm based on access pattern modification that makes efficient use of post-increment/decrement addressing modes in DSPs. This algorithm assumes a fixed storage order for variables and restructures the code to exploit these addressing modes. This algorithm is more general than the one proposed in [13] as it considers both intra-statement and inter-statement transformations.

(2) It gives an algorithm that modifies an access pattern (access sequences), given a partially-fixed storage order. A partially-fixed storage order is a storage order in which the memory locations of only a subset of the variables are fixed.

(3) It combines these two algorithms with the storage order-based optimization strategy (i.e., offset assignment) developed by Liao et al. [11], and presents a unified approach (which is demonstrated to be superior) to handle the offset assignment problem for a given control flow graph.

2 Review of Offset Assignment

The offset assignment problem [10] is one of assigning a frame-relative offset (i.e., storage location) to each variable in the code in order to minimize the number of address arithmetic instructions (that is, the instructions that load a new value to the address register) required to execute the code. The cost of an offset assignment is defined as the number of such instructions.

Given a code sequence, we can define a unique *access sequence* for it. In an operation `a = b op c`, where '`op`' is some binary operator, the access sequence is given by `b, c, a`. The access sequence for an ordered set of operations is simply the concatenated access sequences for each operation taken in order. For example, for the code fragment

```
a = c + d
d = d + c + b + c + a
```

the access sequence is `c, d, a, d, c, b, c, a, d`, assuming that addition is left-associative. Let us assume that the variables in this code fragment are stored in memory in the following order: `a, b, c, d`. The cost of a given storage sequence (offset assignment) is the number of consecutive accesses (in the access sequence) for which the accessed variables are *not* assigned to adjacent locations in memory. Therefore, the cost of the offset assignment given above is four as there are four transitions in the access sequence between non-adjacent variables.

The objective of the offset assignment problem is to determine a storage order for variables such that the cost will be minimum. Liao [10] showed that the offset assignment problem is equivalent to the Maximum Weighted Path Cover (MWPC) problem and proved that it is NP-complete. His heuristic solution was later improved by Leupers and Marwedel [9] who presented a tie-breaking strategy for achieving better storage assignments.

3 Computation Restructuring for a Fully Fixed Storage Sequence

Code size reduction using address register assignment is achieved by making the *access sequence* (i.e., the order in which the variables are accessed) and the *storage sequence* (i.e., the storage order of the variables in memory) compatible. In practice, it is possible to do either of the following: modify the access sequence for a fixed storage sequence, or modify the storage sequence for a given fixed access sequence. In this section, we discuss a strategy that adopts the former approach as opposed to Liao's scheme [10] which takes the latter approach. In this work, we apply code transformations to a high-level intermediate representation (IR) of the code where optimizations such as conventional (e.g., graph coloring-based) register allocation and common subexpression elimination have already been performed. This IR has statements very similar to high-level source statements. In the remainder of this presentation, when we mention statement, we actually refer to this IR-level statement. However, to make the presentation clear, we use source-level (C-like) statements. Consider, a statement of the following form

```
a = b + c
```

Let us assume that the machine has a single address register and that the storage sequence is `c, b, a`. The access sequence in this example is `b, c, a`, which is different from the storage sequence. As a result of this, going from variable `c` to variable `a` incurs an explicit address register load (since `c` and `a` are not consecutive in the storage sequence, so we cannot use post-increment/decrement mode). Liao's approach [10] fixes this problem by modifying the storage sequence from `c, b, a` to `b, c, a`. Changing the storage sequence is a viable option provided that the variables have not yet been assigned to storage locations, or (if they have already been assigned to locations) the cost of transforming the storage sequence from one form to another (which may require copying resulting in additional memory requirements) does not outweigh its benefits. An access pattern-oriented approach, on the other hand, can optimize this code by transforming this statement into

```
a = c + b
```

The new access sequence is `c, b, a` which is the same as the storage sequence. Note that, for this example, just applying commutativity transformation (an intra-statement transformation) was sufficient to obtain the desired result.

Let us consider the following code fragment with two statements.

```
a = c + e
b = c + f
```

We assume a single address register and a storage sequence of `a, b, c, d, e, f`. It should be noted that each variable access in this code fragment (under the assumed storage sequence) will require a load to the address register. A storage layout-oriented scheme would change the storage sequence of the variables, but this may be too costly if the variables have already been assigned to storage locations (for example, during the optimization of a different set of statements that manipulate the same variables.) On the other hand, a commutativity transformation would lead to

```
a = c + e
b = f + c
```

Note that this code fragment (which is obtained from the previous one by applying commutativity transformation to the right-hand side of the second assignment statement) eliminates one of the explicit loads to the address register. That is, in going from `c` to `b` in the second assignment statement, we can make use of the post-decrement mode (as these two variables are consecutive in memory). An inter-statement transformation, on the other hand, can generate the following program fragment

```
b = f + c
a = c + e
```

Note that this code fragment is obtained from the original one by interchanging the order of two statements and by applying commutativity transformation to one of the statements. In this case, two variable accesses (i.e., going from `c` to `b` in the first statement, and going from `b` in the first statement to `c` in the second statement) can be satisfied using post-increment/decrement modes. This is a simple example that illustrates the benefit of inter-statement optimization. However, there are some cases where it is not possible to interchange the order of statements due to data dependency constraints. For example, in the code fragment

```
a = a + c
c = c + 1
```

interchanging two statements would give a wrong result as the value used for `c` in `a = a + c` would be different than the one in the original case. Here, a storage-oriented approach (e.g., [10]), on the other hand, could store `a` and `c` in consecutive locations in memory, thereby leading to the effective use of post-increment and decrement addressing modes.

The preceding examples show that neither storage based techniques nor access sequence (computation) based techniques (intra and inter statement transformations) dominate the other, and a unified framework that uses both the techniques may be needed for better results. In the rest of this section, we formulate the computation oriented transformations using a graph-based representation.

3.1 Terminology

We represent a program using a control flow graph (CFG) which is a directed graph in which each node denotes a basic block and an edge between two basic blocks indicates that there is a possibility that the flow of control (during execution) may be transfered from one of these basic blocks to the other. A basic block can be defined informally as a straight-line sequence of statements that can be entered only at the beginning and exited only at the end [16].

Consider a graph $G = (V, E)$ where V is the set of nodes (vertices) and E is the set of edges. A path cover (or cover) C of a given graph $G(V, E)$ is a set of paths such that every node in V is incident at some edge belonging to the chosen set of paths. In other words, we can think of a cover $C(V', E')$ as a subgraph of $G(V, E)$ where $V' = V$ and $E' \subseteq E$. The length of a path is the number of edges in the path, and the length of a cover is the sum of the number of edges of each constituent path. A path that has the maximum length (among all paths in the cover) is referred to as the longest path.

3.2 Layout Transition Graph

Given a basic block, we use a *layout transition graph* (LTG) to show the connections between elements that are stored consecutively in memory. The layout transition graph of a basic block is a directed graph $LTG(V, E)$, where each node v_i represents a variable that occurs in the basic block; and a directed edge $e = (v_i, v_j)$ from a node v_i to a node v_j indicates that the variable represented by v_i is stored (in memory) next to the variable represented by v_j. Whether v_i comes before v_j in the storage order or after v_j is not important for the purposes of this work (as long as they are consecutive in memory). An LTG also contains an edge from v_i to v_j if these two nodes represent the occurrences of the same variable. Note that the variable access pattern of a program touches all the nodes of the corresponding LTG.

For ease of exposition, we divide a given LTG into layers, each layer corresponding to a statement in the basic block. If the basic block contains K statements, each variable v_i in the jth statement from top (denoted s_j where $1 \leq j \leq K$) is assumed to belong to the variable set of s_j; we express this as $v_i \in s_j$. We will use s_j to denote both the statement and its variable set, where there is no confusion.

A given variable set s_i can also be divided into two logical subsets: one that contains the variable on the left hand side (LHS), and one that contains the variables on the right hand side (RHS). For a variable set s_i, the first subset is denoted by s_{iL} and the second subset is denoted by s_{iR}.

To illustrate these concepts, consider the LTG shown in Figure 1(i) for the statement `a = b + c`, assuming that the storage sequence is `c, b, a`. There is a bi-directional edge between `c` and `b` (i.e., we have a directed edge from `c` to `b` and one from `b` to `c`), and another bi-directional edge between `b` and `a`. Labeling this statement by s_1, we have $s_{1L} = \{\texttt{a}\}$ and $s_{1R} = \{\texttt{b, c}\}$. Note that the access sequence for this statement is `b, c, a` as shown in Figure 1(iii)

using dashed arrows. It should also be noted that a new access sequence can be obtained by traversing the edges in the LTG in a different manner. If we start from the variable `c`, we can first traverse the edge (`c,b`) and then the edge (`b,a`), as depicted in Figure 1(iv). Note that this new traversal corresponds to transforming the statement from `a = b + c` to `a = c + b` (i.e., a commutativity transformation).

We need to emphasize that it may not always be possible to transform a statement based on its LTG. Further, not every traversal of the edges in the LTG is legal. For example, going from `a` to `b` using the edge (`a,b`) is not acceptable (see Figure 1(v)) as all the right-hand side references should be accessed before the left hand side reference. We can prevent some of the transitions such as this by eliminating edges from the LTG that would lead to unacceptable or infeasible transformations. For example, in order to prevent a transformation from `a` to `b`, we eliminate the directed edge from `a` to `b` as shown in Figure 1(ii). Obviously, given the two legal traversals in Figures 1(iii) and (iv), we prefer the one in Figure 1(iv) as all transitions between variables in this figure are between consecutive memory locations, meaning that we can use post-increment/decrement mode for these transitions. Another way of expressing this is that both the edges visited during the traversal in Figure 1(iv) belong to the LTG given in Figure 1(ii). On the other hand, one of the transitions taken during the traversal in Figure 1(iii) (the transition from `c` to `a`) does not have any corresponding edge in the LTG. Therefore, the objective of a traversal must be minimizing the number of transitions that do not correspond to an edge in the LTG. We will formalize this concept later.

Now, let us consider the LTG given in Figure 1(vi) for the following program fragment.

```
a = c + e
b = c + f
```

It is assumed here that the storage sequence is `a, b, c, d, e, f`. As before, a traversal of the nodes of this LTG corresponds to a specific access sequence. The default access sequence is `c, e, a, c, f, b` as shown in Figure 1(viii). Note that a different traversal of the nodes corresponds to a transformation of the code sequence. Here, an important point should be noted. In traversing the nodes (or edges), we have a restriction in the sense that once we are in a statement we need to finish all the nodes in the statement before moving to a node in another statement. That is, we are not allowed to go from a node in s_{kR} to a node in $s_{k'R}$ if $k \neq k'$, assuming that each statement has a left hand side variable.

The preceding discussion indicates that we need some restrictions on the traversal order of the nodes in the LTG. For this purpose, we use a modified form of the LTG called *constrained layout transition graph* (CLTG), and perform our traversal on this graph. Simply, in those cases where the compiler can detect that variable v_i in statement s_k cannot be accessed immediately after the variable v_j in statement $s_{k'}$ (s_k and $s_{k'}$ are not necessarily distinct here), the corresponding edge (if any) from v_j to v_i in the LTG should be removed when constructing

the CLTG (Instead of deleting edges from the LTG to construct the CLTG, it is possible to directly construct the CLTG using the necessary edges, albeit using somewhat more complicated rules. The correctness of the algorithms is not affected by the choice of either method to construct the CLTG).

A constrained layout transition graph, written $CLTG(V', E')$, is a subgraph of the $LTG(V, E)$ such that $V' = V$ and E' contains all the edges in E *except* those that can lead to an incorrect or infeasible code transformation. The construction of the CLTG subsumes both the intra-statement constraints (i.e., evaluation rules that need to be obeyed when processing an RHS expression) and the inter-statement constraints (i.e., dependence and other constraints between statements). For example, a CLTG cannot contain an edge between the variable occurrences of the right hand sides of two different assignment statements. In mathematical terms, an edge $e = (v_i, v_j) \in E$ does not belong to E' if $v_i \in s_{kR}$ and $v_i \in s_{k'R}$, where $k \neq k'$. Figure 1(vii) depicts the CLTG for the LTG in Figure 1(vi). Note that the default traversal (access sequence) given in Figure 1(viii) does not use any of the edges in the underlying CLTG. Consequently, an explicit address register load is necessary prior to each variable access. Now consider the traversal given in Figure 1(ix). In this case, the new access sequence corresponds to a transformation in which the right hand side of the second statement is transformed using commutativity. Note that one of the transitions in this traversal (i.e., the one from `c` to `b`) has a corresponding edge in the CLTG given in Figure 1(vii). Finally, let us focus on the traversal given in Figure 1(x). The transformation corresponding to this traversal is one of interchanging the order of the two statements and applying the commutativity transformation to one of the statements. In this traversal, two transitions, one going from `c` to `b` and the other going from `b` to `c` have corresponding edges in the CLTG. These two examples in Figure 1 show that the preferred traversal must maximize the number of transitions that have corresponding edges in the underlying CLTG. In other words, it should minimize the number of transitions that do not have corresponding edges in the CLTG.

It should be noted, however, that although a given CLTG shows possible legal transitions between nodes, it is still possible to generate an illegal traversal (access sequence) on the CLTG. For example, by itself, accessing two nodes v_i and v_j consecutively may not break any dependence; however, after this modified access sequence, it may not be possible to generate legal code due to a new restriction (in the access order) resulting from the said transition between v_i and v_j.

3.3 Traversing the CLTG

We formulate the problem of modifying a given basic block code for effective use of the address register(s) as one of determining a path cover and a traversal order in the CLTG. We assume for now that the AGU has only a single address register.

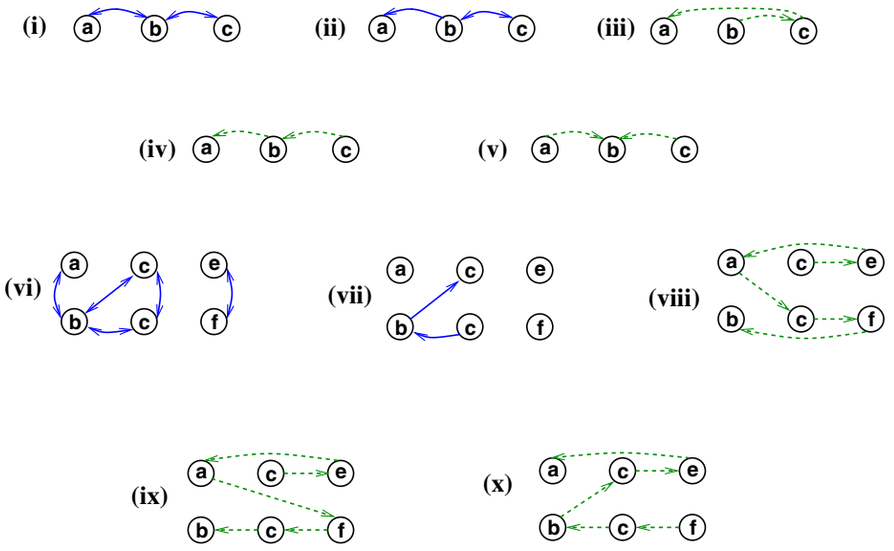

Fig. 1. (i-v) LTG, CLTG, and different traversals for an assignment statement under the storage sequence `c, b, a`. (vi-x) LTG, CLTG, and different traversals for a program fragment under the storage sequence `a, b, c, d, e, f`.

Legality. In order to generate correct code (that is, to preserve the original semantics of the basic block), we impose the following conditions on the traversal order:

(1) Each node in the LTG (i.e., a variable occurrence in the basic block) should be visited.

(2) For a given layer in the LTG corresponding to the statement s_k, all nodes in s_{kR} should be visited before any node in s_{kL}.

(3) Once the traversal reaches the layer corresponding to the statement s_k, it should finish all the variables in that layer (i.e., the set $s_{kL} \cup s_{kR}$) before moving to another layer.

(4) All the data dependences and other restrictions such as latency constraints or expression evaluation constraints should be observed.

Condition (1) indicates that each variable should be touched (by any legal execution of the code). We enforce Condition (4) by ensuring that we do not make a transition from a $v_i \in s_k$ to a $v_j \in s_{k'}$ (even if v_i and v_j are consecutive in memory) when there is a data dependence from $s_{k'}$ to s_k. To enforce Condition (2), we do not allow a transition from the node $v_i \in s_{kL}$ to a node $v_j \in s_{kR}$. To enforce Condition (3), we disallow transitions between node $v_i \in s_{kR}$ and any node $v_j \in s_{k'R}$ for $k \neq k'$. A transition from a node $v_i \in s_{kL}$ to a node $v_j \in s_{k'L}$ (where $k \neq k'$) is allowed only if $s_{k'}$ has no variables on the right hand side (i.e., $s_{k'R} = \emptyset$). Also, there cannot be a transition from a node $v_i \in s_{kR}$ to a node

$v_j \in s_{k'L}$ (where $k \neq k'$) unless $s_{k'}$ has no variable on the right hand side (i.e., $s_{k'R} = \emptyset$) and s_k has no LHS variable, which cannot occur in our framework.

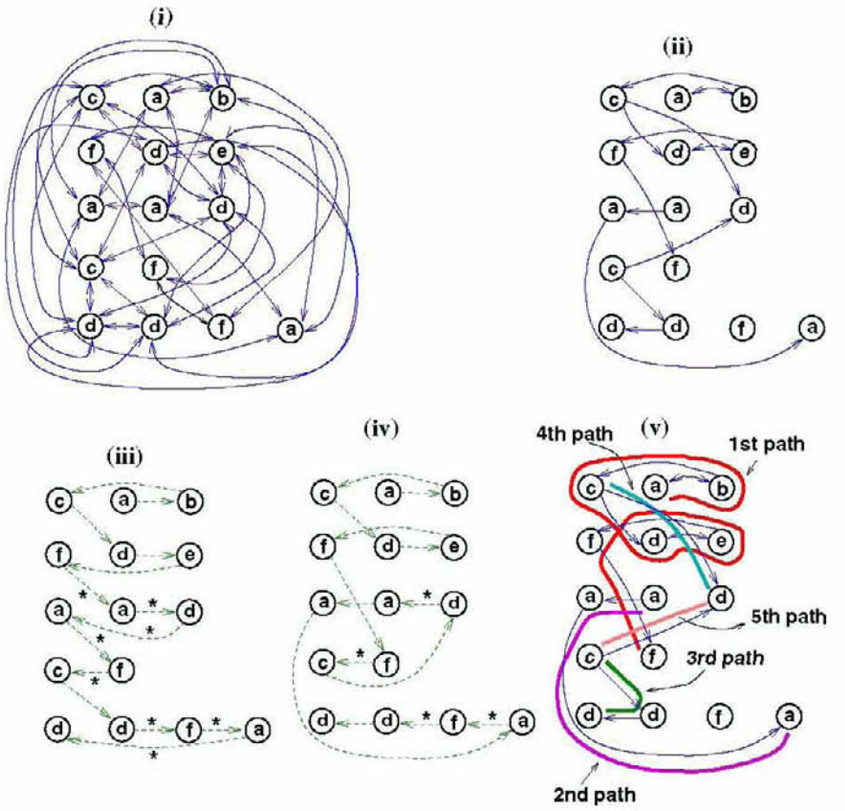

Fig. 2. (i) LTG and (ii) CLTG for a given basic block. (iii) Default access sequence. (iv) Optimized access sequence. (v) Example paths in the CLTG.

Profitability. The objective of the traversal of the nodes in the CLTG is to minimize the *cost of the traversal*, which is defined as the number of transitions from a node v_i to a node v_j such that v_i and v_j are not consecutive in the storage sequence (i.e., there is no edge (v_i, v_j) in the CLTG) for all i and j. It should be noted that a storage sequence imposes constraints on the CLTG. If a transition from v_i to v_j does not use an edge in the CLTG, this means that a post-increment or a post-decrement cannot be used for this transition; thus, new value should be loaded in the address register (using an explicit load instruction), thereby increasing the code size. As a result, the cost of a traversal can be viewed as

the number of transitions in the access sequence that do *not* use an edge in the CLTG. Thus, the address register assignment problem can be re-expressed as

> *determining a traversal of the nodes in the CLTG—subject to the four legality conditions listed above—that minimizes the number of transitions that do not correspond to an edge in the CLTG.*

It can be shown that this problem is NP-complete; but, we omit the proof due to lack of space.

Let us now concentrate on the larger basic block given below assuming a storage sequence of `a, b, c, d, e, f`.

```
c = a + b
f = d - e - 2
a = a + 3d
c = 2f + 4
d = d + f + a
```

Figures 2(i) and (ii) show the LTG and CLTG, respectively, for this code fragment under the assumed storage sequence. Note that, in going from the LTG to the CLTG, many edges are dropped as they are not possible for any legal traversal. Figure 2(iii) shows the default access sequence (i.e., without any optimization). This access sequence has a cost of eight, and the transitions that contribute to this cost are marked using the symbol '*'. Our approach, on the other hand, results in the access sequence (traversal) given in Figure 2(iv). We see that the cost of this access sequence is four (again, the transitions that contribute to the cost are marked using the symbol '*'). In other words, we are able to eliminate four address register loads in the code. This traversal corresponds to the following transformed program:

```
c = a + b
f = d - e - 2
c = 2f + 4
a = 3d + a
d = a + f + d
```

Note that this optimized code is obtained from the original one through one statement reordering (inter-statement transformation) and a number of intra-statement transformations.

The Algorithm and Transformations. We now present an algorithm that takes as input a CLTG and generates as output a traversal (an access sequence) and all the necessary (inter-statement and intra-statement) transformations to obtain this access sequence. Given a CLTG, the algorithm first detects the longest directed path (i.e., the path that contains the maximum number of edges in the same direction).[1] It then transforms the portion of the CLTG (which contains a subset of the statements in the original basic block) in accordance with

[1] Note that the longest path detection problem is a hard problem in general. Here, we are employing a heuristic.

this longest path. Finding the longest path in a given directed graph is straightforward, and takes $O(N^3)$ time, where N is the number of nodes in the graph [5]. Transforming the program code in accordance with the longest path is more challenging. Consider the abstract CLTG in Figure 3 and the longest path shown. Note that each layer in the CLTG is labeled with a different statement id. The desired access sequence here is `a, c, h, d, f, g, b, e`. To achieve this access sequence, the following transformations need to be performed:

(1) The variable `a` should be made the last variable accessed on the RHS of the statement s_1;

(2) In statement s_2: (i) the variable `h` should be made the first variable accessed on the RHS; (ii) the variable `h` should be made to immediately precede the variable `d`;

(3) Statement s_4 should be made to immediately follow the statement s_2; and

(4) In Statement s_4: (i) the access of variable `b` should be made to immediately follow the variable `g`; (ii) the variable `e` should be made to immediately follow the variable `b`.

In addition to these transformations, the transformed program should not modify the following properties of the input code (CLTG):

(1') Statement s_2 immediately follows statement s_1.

(2') `d` is the last variable accessed on the RHS of Statement s_2.

(3') `g` is the first variable accessed on the RHS in Statement s_4.

If the compiler can find a series of transformations to satisfy all these constraints, we achieve the best possible access sequence (for this path). In many cases, however, this may not be possible due to inconsistencies between the requirements given above, or due to a situation that does not involve the variables on the longest path. An example of the former is the inconsistency between conditions (2.i), (2'), and (2.ii) above. That is, if we make the variable `h` the first variable on the RHS of the statement s_2 and insist on keeping the variable `d` as the last variable on the RHS, it is not possible to access `h` and `d` successively as there are two more variables on the RHS. We assume that these other variables are different from those labeled in the figure. An example of the second type of difficulty is the possibility that it may not be legal to access the statement s_4 immediately after the statement s_2 (as required by the condition(3)). This may occur for example if the statement s_3 writes a variable `x` (assumed to be a different variable from the ones shown in the figure) that is subsequently read by the statement s_4. Although it may not always be possible to achieve all of the desired transformations, our approach attempts to achieve as many of the desired transformations as possible. Note that this strategy helps to use as many edges in the CLTG as possible.

After the longest path has been determined and the portion of the CLTG that contains the longest path (that is, a subset of the statements in the original basic block) has been transformed, our approach continues by selecting the second longest path and transforming the relevant parts of the CLTG. A special attention is paid to ensure that we do not modify any parts of the basic block

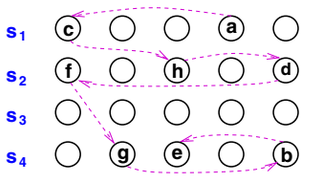

Fig. 3. An abstract CLTG and the longest path.

that have already been transformed in accordance with a longer path considered earlier. In this way, our approach selects the next longest path in each step and transforms the relevant portions of the basic block. The process stops when it is not possible to transform the basic block any further (without distorting the previous transformations). In case we have two paths of the same length, the current implementation favors the one that leads to minimal modification to the original code.

In the example in Figure 2, following the construction of the CLTG shown in Figure 2(ii), our approach determines the longest path marked as the 1^{st} `path` in Figure 2(v). Based on this path, it builds an access sub-sequence `a`, `b`, `c`, `d`, `e`, `f`, `f`. This sub-sequence completely specifies the transformations required for three of the five statements in the code (i.e., the first, second, and fourth statements in the original code). Note also that the transformations performed along this path include an inter-statement transformation. Next, it finds the path `a`, `a`, `a` (marked as the 2^{nd} `path`). Note that this path fixes the access sequence for the third statement in the original code completely as `d`, `a`, `a`. It also specifies that the variable `a` should be the first variable accessed in fifth statement. After that, the approach selects the path `c`, `d`, `d`. The (`c`,`d`) part of this path says that the fifth statement should follow the fourth statement in the transformed program, but this is not possible as the fourth statement has already been transformed, and it now (in the transformed code) comes before the third statement (in the original program). The (`d`,`d`) part of the path, on the other hand, is feasible, and indicates that `d` should be the last variable accessed in the fifth statement. The next path is `c`, `d`; but, the transformation implied by this is not possible. The last path is the one between `c` and `d` (marked as the 5^{th} `path` in the figure). It implies that `d` should be the first variable accessed in the third statement, and the third and fourth statement should be interchanged. At this point, the algorithm has traversed all the paths. It next visits each statement, and fixes the access order for the variable whose order has not been fixed yet. It visits the fifth statement (in the original code) and makes `f` the second variable accessed on the RHS. The final access sequence is shown in Figure 2(iv).

4 Computation Restructuring: Partially Fixed Storage Sequence Case

So far, we have assumed that the storage sequence (storage pattern) of variables is fixed completely. That is, a storage location is assigned to each program variable. In this section, we describe how to optimize an access sequence when only a subset of the variables have fixed memory locations. This is called the partially fixed storage. Specifically, given a partially fixed storage pattern of a basic block, we address two subproblems:

(1) Determining the best access sequence for all variables in the basic block, and

(2) Determining the storage sequence for the variables in the basic block whose memory locations are yet to be determined.

This problem is important because the compiler employs it during procedure-wide optimization (as will be discussed in the next section). Our approach to the problem involves the following three steps:

(1) Determine the best access (possibly partial) pattern for the partial storage order given,

(2) Determine the storage sequence for the variables whose memory locations are yet to be determined, and

(3) If there is further flexibility, then determine the best access pattern for the portions of the basic block that involves the variables whose storage sequence was determined in Step (2).

Consider the following program fragment assuming a single address register and a partially fixed storage sequence of `e, b, d`.

```
e = e + d
a = d + c
f = 3c + b
a = (a * c) + (a * g)
```

Figure 4(i) shows the CLTG for this basic block, under the given partial storage sequence. Clearly, there is just one path in this case. Transforming the code in accordance with this path gives us:

```
e = d + e
f = b + 3c
a = d + c
a = (a * c) + (a * g)
```

Note that this transformation (which corresponds to Step (1) above) involves one statement interchange and one commutativity transformation. In the next step (which is Step (2) above), the compiler attempts to determine a storage sequence for the variables whose storage locations are yet to be determined. We achieve this using a *modified version* of Liao's heuristic [10]. Liao summarizes the access sequence using a graph called the *access graph*. In this graph, each variable is represented by a node and a weighted edge between two variables corresponds to the number of transitions between them. Liao then runs an algorithm on this

graph to select a path cover, with no node having more than two selected edges incident on it.

The variables represented by the nodes connected by a selected edge are assigned to consecutive memory locations. The objective is to maximize the total weight of the edges selected (which corresponds to capturing the most frequent transitions). We modify this heuristic as follows. Let $\mathcal{L} = \{v_i\}$ be the set of all variables v_i that have already been assigned to consecutive storage locations. Let us assume for now that there is only a single such set. We use $b_{\mathcal{L}}$ to denote the first (start) node of $\mathcal{L}$, and $t_{\mathcal{L}}$ to denote the last (terminal) node. Each node in the modified access graph corresponds to either a single node v_j such that $v_j \notin \mathcal{L}$ or a block node $v_{\mathcal{L}}$ that represents $\mathcal{L}$. There exists an edge between v_j ($\notin \mathcal{L}$) and $v_{\mathcal{L}}$ if and only if there is an edge between v_j and $b_{\mathcal{L}}$ or an edge between v_j and $t_{\mathcal{L}}$. We also keep track of whether the edge between v_j and $v_{\mathcal{L}}$ is due to (incident on) $b_{\mathcal{L}}$ or $t_{\mathcal{L}}$.

Figure 4(ii) shows this modified access graph for our example. Note that this access graph is constructed by taking into account the transformations (both inter-statement and intra-statement) done in the previous step. Next, we run Liao's heuristic [10] on this access graph. Figure 4(iii) show the maximum weight cover detected by the heuristic. Afterwards, we determine the complete storage order (sequence) for the variables. In our example, this sequence is `e, b, d, f, c, a, g`. Although it does not occur in this example, in some cases, the compiler may have additional scope, and may apply Step (3) above to further modify the access pattern to accommodate the needs of the variables whose storage locations have been determined in Step (2). Note that although we explain this strategy assuming that there is a single block node ($\mathcal{L}$), it is straightforward to extend the approach to multiple block nodes. Note also that since our approach is essentially basic block oriented, we can expect its effectiveness to increase when it is used in conjunction with techniques that increase basic block sizes (e.g., superblocks/hyperblocks).

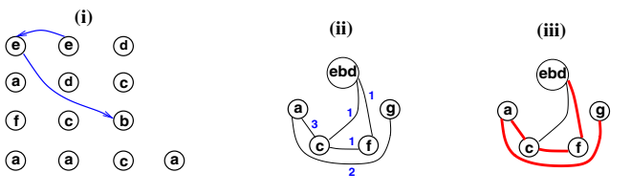

Fig. 4. (i) An example CLTG. (ii) An access graph for partially fixed storage sequence. (iii) Selected maximum weight cover.

5 Intra-procedural Optimization Strategy

We now present a unified strategy that employs both access sequence and storage sequence transformations to make effective use of address registers. The approach

works on a representation called *weighted control flow graph* (WCFG), which is a CFG with weighted nodes (basic blocks). A node weight specifies the number of times the corresponding basic block is entered (dynamic execution frequency). This is typically calculated by considering the execution frequencies of edges and branch probabilities.

Our approach to this global (procedure-wide) optimization problem is as follows. After determining the execution frequencies of basic blocks and labeling them, we visit basic blocks one-by-one, and optimize a basic block completely before moving to the next one. The optimization order is determined by the weights (i.e., basic block labels).

The first (most frequently executed) basic block is optimized using Liao's heuristic (explained in Section 2). After optimizing this basic block, we determine a storage sequence for all the variables accessed by this basic block. Note that this step determines only a partial storage sequence (called the *storage subsequence*) as the variables accessed by this block form, in general, a subset of all the variables declared in the program. Then, we move to the next most frequently executed basic block, and optimize it using the approach explained in Section 3 or Section 4 depending on whether all the variables manipulated by this basic block has already fixed memory (storage) locations or not. After optimizing this basic block, new storage subsequences (for the variables accessed by this second most frequently executed basic block, but not accessed by the most frequently executed basic block) are determined. Afterwards, we move to the third most frequently executed basic block and, in optimizing it (using the techniques given in Section 3 and Section 4), we take into account all the storage sequences determined so far. In this way, our approach handles the basic blocks one-by-one, and in optimizing each of them, it considers the storage sequences found so far. If at a given point, the storage location for each variable in the code is fixed (i.e., a complete storage sequence is determined), the remaining basic blocks are optimized using the technique discussed in Section 3. At the end of the process, if the storage sequences found do not form a single connected component, they are made so using a post-processing pass.

6 Summary

In this work, we have presented a compilation framework that employs both program restructuring and storage order optimizations to reduce the size of the generated code for embedded processors by eliminating as many explicit address register loads as possible. Reducing code size is extremely important as in many embedded systems a reduction in code size means a reduction in memory size. Work in progress includes the investigation of different ways of combining storage layout and code restructuring transformations, incorporating partitioning of variables among different address registers, and studying the impact of SSA transformation on code size. We also plan to make experiments with different architectures as different instruction set architectures (ISA) can lead to different code sizes [6].

References

1. D. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. Software – Practice and Experience, 22(2):101–110, February 1992.
2. P. Briggs. Register Allocation via Graph Coloring, Ph.D. Thesis, Computer Science Department, Rice University, Houston, TX, April 1992.
3. M. Cintra and G. Araujo. Array reference allocation using SSA-form and live range growth. In Proc. ACM SIGPLAN 2000 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), June 2000, Vancouver B. C., Canada.
4. K. Cooper and P. Schielke. Non-local instruction scheduling with limited code growth. In Proc. Workshop on Languages, Compilers, and Tools for Embedded Systems (LCPC), pp. 193–207, June 1998.
5. T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms, MIT Press, Cambridge, Massachusetts, 1990.
6. J. W. Davidson and R. A. Vaughan. The effect of instruction set complexity on program size and memory performance. In Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1987, pp. 60–64.
7. M. Kandemir. A compiler technique for improving whole program locality. In Proc. 28^{th} Annual ACM Symposium on Principles of Programming Languages (POPL), London, UK, January, 2001.
8. C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In Proc. the 30th International Symposium on Microarchitecture (MICRO), pp. 330–335, 1997.
9. R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In Proc. the International Conference on Computer Aided Design (ICCAD), pp. 109–112, November 1996.
10. S. Liao. Code Generation and Optimization for Embedded Digital Signal Processors, Ph.D. Thesis. MIT, June 1996.
11. S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. ACM Transactions on Programming Languages and Systems (TOPLAS), 18(3):235–253, 1996.
12. S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1st edition, July 1997.
13. A. Rao and S. Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), May 1999.
14. R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Notices, 29(12):31–37, December 1994.
15. S. Udayanarayanan and C. Chakrabarti. Address code generation for DSPs. In Proc. the 38th Design Automation Conference (DAC), June 2001.
16. M. Wolfe. High Performance Compilers for Parallel Computing, Addison Wesley Publishing Company, 1996.