# Reducing Code Size Through Address Register Assignment

G. CHEN, M. KANDEMIR, and M. J. IRWIN
Pennsylvania State University
and
J. RAMANUJAM
Louisiana State University

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 postincrement/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). Experimental results on a set of benchmark codes show the effectiveness of our solution and indicate that our approach outperforms the previous approaches to the problem. In addition to resulting in significant reductions in instruction memory (storage) requirements, the proposed technique improves execution time.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Main memory*; D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*

General Terms: Design, Performance

Additional Key Words and Phrases: Software compilation, address registers, DSP, register assignment

## 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

Authors' addresses: G. Chen, M. Kandemir and M. J. Irwin, CSE Department, Pennsylvania State University, Univerity Park, PA 16802; email: {guilchen,kandemir,mji}@cse.psu.edu; J. Ramanujam, ECE Department, Louisiana State University, Baton Rouge, LA 70803; email: {jxr}@ee.isu.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 1539-9087/06/0200-0225 $5.00

systems) running on Motorola's DSP56000 processor, nearly 55% of the instructions are used to manipulate address registers through explicit loads and stores [Udayanarayanan and Chakrabarti 2001]. 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 77110, Motorola DSP56000, Analog Devices ADSP21xx, and Texas Instruments TMS320C5x are done in address generation units (AGUs). An AGU contains a number of 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 postincrement/decrement of an address register in a single instruction. Thus, using postincrement/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 [2000] report that although some of the register increment/decrement operations can be accommodated in VLIW instruction slots, modern VLIW DSP architectures also have autoincrement and autodecrement 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 postincrement/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 postincremented/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 postincrement 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 postincrement 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 again use postincrement 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 [1992], Liao et al. [1995, 1996], and others [e.g., Leupers and Marwedel 1996; Udayanarayanan and Chakrabarti 2001]. 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., Liao 1996; Liao et al. 1995] or only on modifying the intrastatement access pattern using commutativity and associativity transformations [e.g., Rao and Pande 1999]. In this work, we present a framework that considers both computation-based (intra- and interstatement) 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 postincrement/ 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 Rao and Pande [1999] as it considers both intra- and inter-statement transformations (whereas the approach in Rao and Pande [1999] considers only intrastatement 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. [1995], and presents a unified approach (which is demonstrated to be superior) to handle the offset assignment problem for a given control flow graph (procedure-wide optimization).
4. It shows how our approach can be made to work in an interprocedural setting by making use of a call graph representation of the program being optimized.

As far as ISA is concerned, our technique does not assume anything on the underlying DSP architecture except for the existence of an AGU, which is available in many current DSP architectures (e.g., Motorola DSP56000 and TMS320C5x). Thus, our technique can optimize a broad set of DSP applications running on various DSP architectures. We implemented our technique using an experimental compiler and tested its effectiveness using seven embedded

benchmarks from the MediaBench suite [Lee et al. 1997]. Our results indicate significant reductions in code sizes over previous techniques.

Section 2 presents a brief review of the offset assignment problem. Section 3 presents a strategy that uses code transformations to improve the access sequence, assuming a fixed storage order; in Section 4, we discuss a more general technique that can work with a partially fixed storage sequence. Section 5 gives a unified procedure-level strategy that uses both storage and access sequence optimizations. Section 6 discusses how our approach can work in an interprocedural setting. Section 7 presents experimental data that demonstrate the efficacy of our approach. Finally, Section 8 concludes with a summary and an outline of the future work.

## 2. REVIEW OF OFFSET ASSIGNMENT

The offset assignment problem [Liao 1996] 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.

Note that offset assignment problem exists for both global and local variables. Local variables are accessed using the stack pointer and instructions for updating stack pointer will increase overall code size (like any other instructions). Consequently, our compiler-directed approach described in this paper utilizes AGU for reducing the number of stack pointer updates as well.

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 nonadjacent variables. The objective of the offset assignment problem is to determine a storage order for variables such that the cost will be minimum. Liao [1996] 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 [1996] who presented a tie-breaking strategy for achieving better storage assignments. A more detailed discussion of Liao's approach will be presented in Section 4.

Note that if the number of variables in the program being analyzed is very small, we can use an exhaustive approach to find the optimum solution. However, in general, this is not the case and we need a systematic approach to address this problem. Our approach mainly targets at such cases where the number of variables to be accessed from memory is large.

## 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 [1996] which takes the latter approach. In this work, we apply code transformations to a high-level intermediate representation (IR) of the code where commonly known low-level optimizations, such as conventional, such as convetional graph coloring-based register allocation and common subexpression elimination, have already been performed.[1] 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 postincrement/decrement mode). Liao's approach [Liao 1996] 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
```

[1]It needs to be noted that, register allocation can reduce the number of memory accesses in the application code and, consequently, reduce opportunities for utilizing AGU through our scheme. However, even the best register allocator (e.g., the powerful allocator used in our experiments) can remove only a small set of the total memory accesses in a data-intensive signal processing application. As a result, one still needs to apply some techniques for removing address calculations. The scheme proposed in this paper is one such technique.

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 intrastatement 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 postdecrement mode (as these two variables are consecutive in memory). An interstatement 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 postincrement/decrement modes. This is a simple example that illustrates the benefit of interstatement optimization. However, there are some cases where it is not possible to interchange the order of statements because of 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., Liao 1996], on the other hand, could store `a` and `c` in consecutive locations in memory, thereby leading to the effective use of postincrement and decrement addressing modes.

The preceding examples show that neither storage-based techniques nor access sequence (computation)-based techniques (intra- and interstatement 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 transferred 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 [Wolfe 1996].

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 occurrence in the basic block (i.e., we have a node for each variable use); 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 $j$th 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 $s_{iR}$.

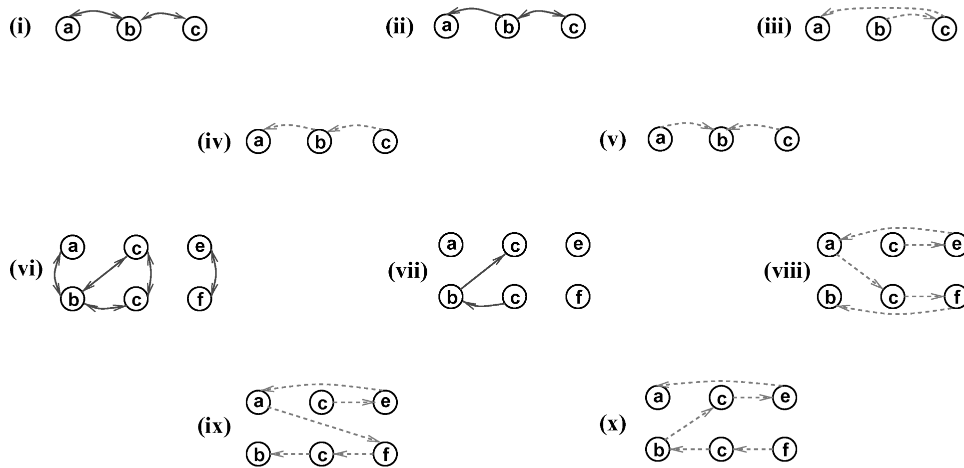


Fig. 1. (i–v) LTG, CLTG, and different traversals for an assignment statement under the storage sequence `c, b, a`. (vi–x) LTG, CLTG, and different traversals for a program fragment under the storage sequence `a, b, c, d, e, f`.

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 bidirectional edge between `c` and `b` (i.e., we have a directed edge from `c` to `b` and one from `b` to `c`), and another bidirectional edge between `b` and `a`. Labeling this statement by $s_1$, we have $s_{1L} = \{a\}$ and $s_{1R} = \{b, c\}$. 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 postincrement/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 the restriction 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 (this restriction is due to the fact that we do not want to interleave variable accesses from different statements since doing so can invalidate the impact of previous optimizations performed on the code).

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 intrastatement constraints (i.e., evaluation rules that need to be obeyed when processing an RHS expression) and the interstatement 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 because of a new restriction (in the access order) resulting from the said transition between $v_i$ and $v_j$. Therefore, our CLTG traversal strategy (which is detailed in the following subsection) is geared toward coming up with a legal (semantic-preserving) order in which the nodes are visited.

### 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. We will later discuss how to extend our approach to handle multiple address registers.

3.3.1 *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.

3.3.2 *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 postincrement or a postdecrement 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 reexpressed 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; however, we omit the proof because of 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
```

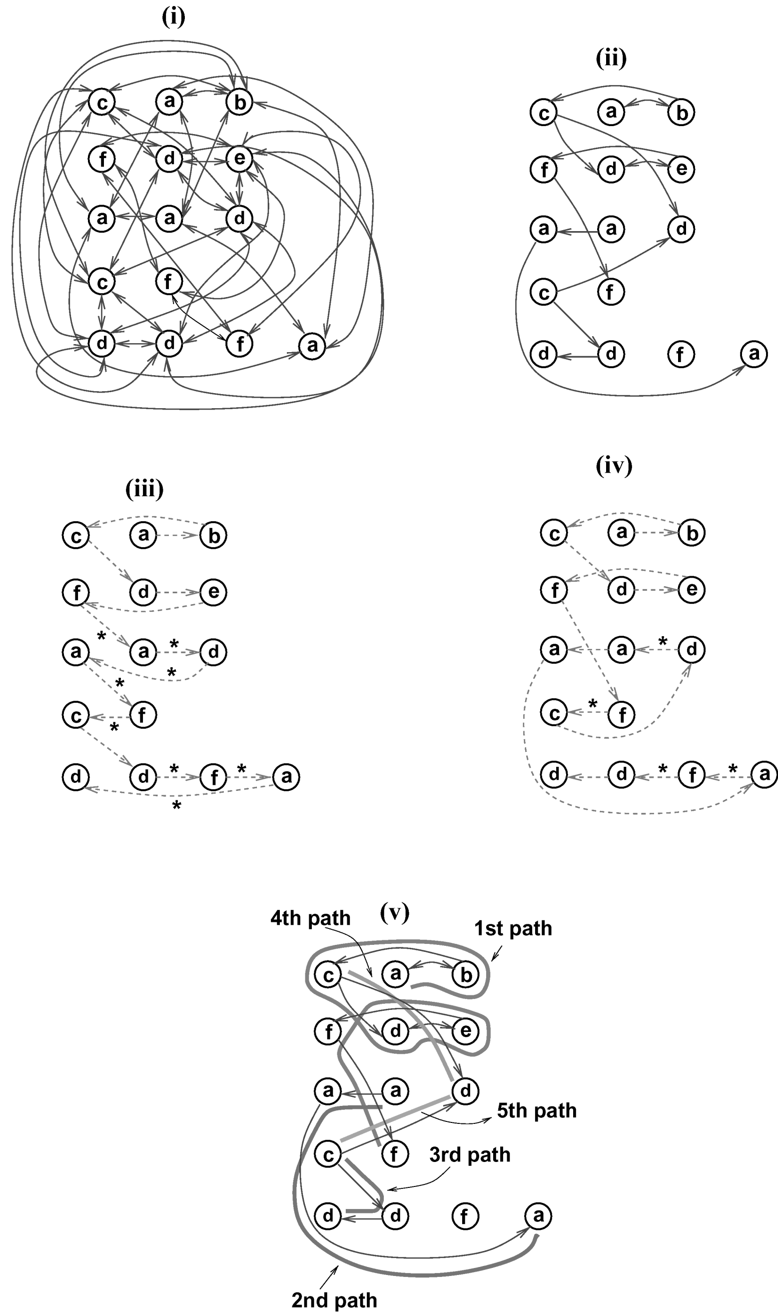


Fig. 2. (i) LTG and (ii) CLTG for a given basic block. (iii) Default access sequence. (iv) Optimized access sequence. (v) Example paths in the CLTG.

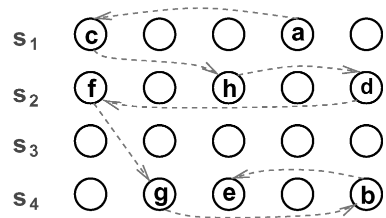


Fig. 3. An abstract CLTG and the longest path.

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

3.3.3 *Overview of the Algorithm.* 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- and intrastatement) 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).[2] It then transforms the portion of the CLTG (which contains a subset of the statements in the original basic block) in accordance with this longest path. Finding the longest path in a given directed graph takes $O(N^3)$ time, where $N$ is the number of nodes in the graph [Cormen et al. 1990]. After the longest path has been determined and the portion of the CLTG that contains the longest path (that is, a subset of the statement 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. Special attention is paid to ensure that we do not modify any parts of the basic block 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. Our experiments show that this tie-breaking strategy performs well in practice.

3.3.4 *Transformations Imposed by a Path.* Transforming the program code in accordance with a path is 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]This can be done by first assigning a unit negative ($-1$) weight to each edge, and then solving the shortest path problem [Cormen et al. 1990].

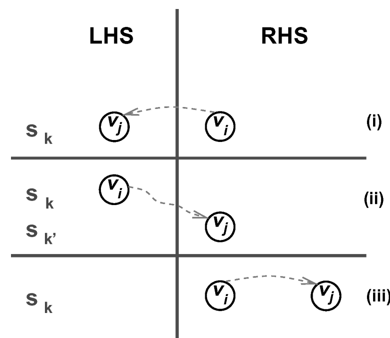


Fig. 4. Different situations that might require code transformations.

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 $s_2$;
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 because of inconsistencies between the requirements given above, or owing 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.

Figure 4 summarizes the situations that *may* demand code (access sequence) transformations during the optimization process. The first situation corresponds to the case where there is an edge from a node $v_i$ in $s_{kR}$ to the node $v_j$ in $s_{kL}$. In this case, the RHS node in question should be made the last node

accessed on the RHS (if it is not already the last node on the RHS). The second situation is the one in which we have an edge (in the CLTG) from the LHS node $v_i$ of a statement $s_k$ to a node $v_j$ in $s_{k'R}$ of statement $s_{k'}$ where $k \neq k'$. In this case, we may need two types of transformations: first, if $s_{k'}$ and $s_k$ are not consecutive, that should be made consecutive (an interstatement transformation); second, if $v_j$ is not the first variable accessed in the $s_{k'R}$, it should be made so (an intrastatement transformation). The third situation corresponds to the case in which we have an edge from a node $v_i$ in $s_{kR}$ to another node $v_j$ in $s_{kR}$. In this case, we need an intrastatement transformation that should bring these two nodes together. In cases where a variable $v_j$ is needed to be both the first variable accessed in $s_{k'}$ (because of the edge from $s_k$ to $s_{k'}$) and be the last variable accessed (because of the intrastatement edge in $s_{k'}$), we need a conflict-resolution scheme.

3.3.5 *Example Transformation.* 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 1st `path` in Figure 2,v. Based on this path, it builds an access subsequence `a, b, c, d, e, f, f`. This subsequence 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 interstatement transformation. Next, it finds the path `a, a, a` (marked as the 2nd `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`; however, the transformation implied by this is not possible. The last path is the one between `c` and `d` (marked as the 5th `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 yet been fixed. 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.

### 3.4 Multiple Address Registers

Thus for we have focused on the address register assignment problem under the assumption of a single address register in the AGU. This subsection describes a heuristic approach that makes use of multiple address registers when the AGU architecture supports it (e.g., as in the case of Analog Devices ADSP21xx digital signal processor line). The idea is to reduce the number of transitions on the CLTG (using multiple address registers) that do not have a corresponding

edge in the CLTG beyond the reduction obtained when we have a single address register.

Our solution starts by building the CLTG and finding all paths on it using the technique explained for the single address register case. Then, starting with the longest path, it improves the cost of the traversal (associated with the path being considered) by making use of extra address registers available. That is, it traverses the paths one-by-one and attempts to eliminate the number of explicit address register loads using a minimum number of address registers. Suppose that we have $R$ address registers and $Q$ paths. First, we try to assign a register per path. The idea here is to eliminate the extra loads (to address registers) that would occur if we use just one register and frequently transition between two paths (i.e., touch variables from different paths) during execution. When we use a register per path, we do not have to worry about path transitions. (Note, however, that using more address registers means more register initializations and, thus, larger basic block size.) If $R \geq Q$, it is always possible to allocate one address register per path. On the other hand, if $R < Q$, we allocate address registers to paths starting with the longest paths. In this case, the paths without private registers reuse registers of other paths, as in the case when we have a single address register. If $R > Q$, the remaining address registers (after assigning one per path) are then distributed across paths and utilized as explained in the following.

Let us focus on a specific path. Our approach, which is a greedy heuristic is, to start with, one address register; we try to access as many variables as possible using this register. In case we need to perform a transition that needs explicit load, the approach allocates a new address register. In each step, it checks whether the next variable can be addressed using one of the address registers already in use by this path and, if so, it uses the appropriate register; otherwise, if there is an available (unused) register, it uses that; if not, it performs an explicit load to one of the address registers in use.

As an example, consider the traversal shown in Figure 5,i. Suppose that we are in node `a` (during the traversal of a path), and need to visit node `b` next (that is, we need to take the edge marked using "*"). Also, suppose that `a` and `b` are *not* consecutive in memory and we have another address register at our disposal. Assuming that the next node to be visited following `b` is `c`. Our approach considers four possibilities:

1. `b` and `c` are consecutive in memory, but `a` and `c` are not consecutive. In this case, we load the second address register with the address of `b`. After that, during accessing `b`, we postincrement/decrement the *second* address register to point `c`. This situation is depicted in Figure 5,ii, where `AR1` and `AR2` denote the first and the second address registers, respectively.
2. `b` and `c` are not consecutive in memory, but `a` and `c` are consecutive. In this case, we load the second address register with the address of `b`. Also, before that, during accessing `a`, we postincrement and decrement the *first* address register to point `c`. This situation is illustrated in Figure 5,iii.
3. Both `b` and `c` and `a` and `c` are consecutive in memory. This case is shown in Figures 5,iv and v. In this case, to decide whether to use the first or the

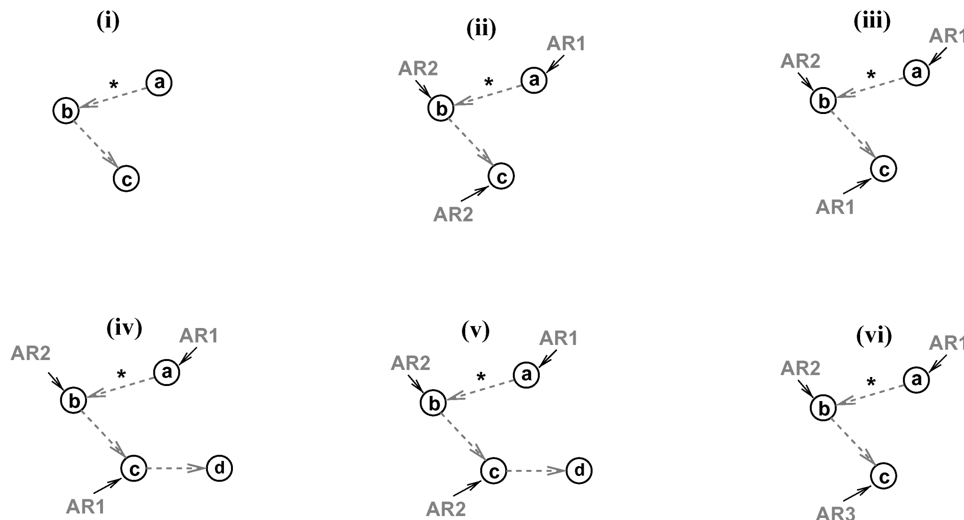


Fig. 5. (i) An example access sequence. (ii–vi) Different address register assignments.

second address register to point to the variable c, we consider the location of the next variable to be accessed following c. If this variable (labeled d in Figures 5,iv and v) and b are consecutive in memory, it is better to use the first address register to access c (Figure 5,iv); otherwise, if d and a are consecutive, we use the second address register for c (Figure 5,v).

4. None of the cases above apply. In this case, if available, we use a different address register for pointing to c ( AR3 in Figure 5,vi). Otherwise, we use either the first or the second address register.

We have observed from our experiments and experience that, in most cases, the number of paths in a basic block is close to the number of address registers (when there are multiple registers in the architecture). Thus, each path can use its own address register.

### 3.5 Addressing Using Larger Increments/Decrements

In some embedded architectures, it is possible to use postincrement/decrement addressing modes with an increment/decrement value of $l > 1$. One way of implementing this is to embed the increment/decrement value $l$ in the instruction format. Our approach can easily be made to work with those architectures as well. The idea is to relax the constraints adopted during the construction of an LTG. Recall that when building an LTG with $l = 1$, we have inserted an edge from a node $v_i$ to a node $v_j$ if, and only if, the variables represented by $v_i$ and $v_j$ are consecutive in the storage sequence. If $l > 1$, we can have an edge between nodes $v_i$ and $v_j$ as long as the variables represented by these nodes are $r$ locations apart from each other, where $r < l$. Two nodes $v_i$ and $v_j$ are $r$ locations apart from each other in a given storage sequence if there are exactly $r$ variables between them. They are called 0 locations apart if they are consecutive in memory.

As an example, assume a storage sequence of `a, b, c, d` and an assignment statement `a = d + c`. Assuming $l = 1$ and we have a single address register, the only edges in the LTG are those between `d` and `c`. In this case, the two possible access sequences, `d, c, a` and `c, d, a`, have the same quality and incur an explicit load to the address register before accessing variable `a`. However, if $l = 2$, we will have an edge from `c` to `a`, in addition to the edges in the previous case (when $l = 1$). In this case, the access sequence `d, c, a` is clearly preferable over the sequence `c, d, a`, as (in the former) going from `c` to `a` can be done using the postdecrement addressing mode with a decrement value of 2.

## 4. COMPUTATION RESTRUCTURING: PARTIALLY FIXED-STORAGE SEQUENCE CASE

Thus, 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;
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;
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 6,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)
```

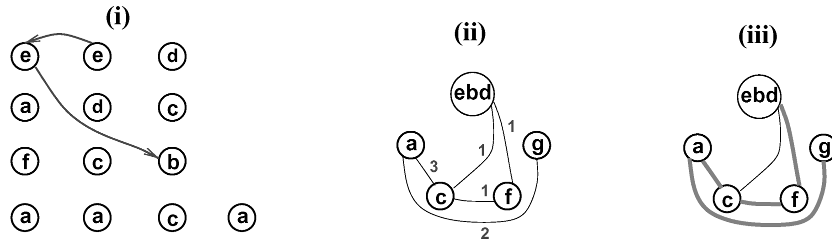


Fig. 6. (i) An example CLTG. (ii) An access graph for partially fixed storage sequence. (iii) Selected maximum weight cover.

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 [1996]. Liao summarizes the access sequence using what he 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 because of (incident on) $b_{\mathcal{L}}$ or $t_{\mathcal{L}}$.

Figure 6,ii shows this modified access graph for our example. Note that this access graph is constructed by taking into account the transformations (both inter- and intrastatement) done in the previous step. Next, we run Liao's heuristic [1996] on this access graph. Figure 6,iii show the maximum weight cover detected by the heuristic. Afterward, 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

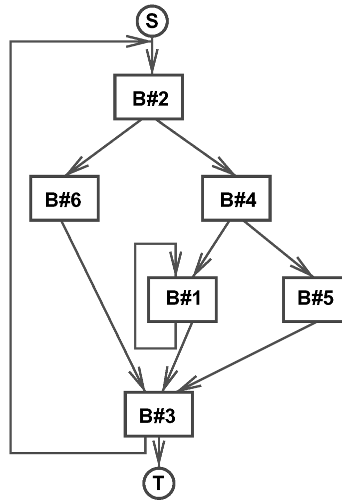


Fig. 7. An example weighted control-flow graph (WCFG). S and T denote the start and terminal nodes, respectively.

when it is used in conjunction with techniques that increase basic block sizes (e.g., superblocks/hyperblocks).

## 5. INTRAPROCEDURAL 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. Eckstein and Krall [1999] discuss the importance of procedure level address register assignment.

Let us start by considering the WCFG shown in Figure 7, which corresponds to the code fragment below.

```
B#2     a8 = a10 + a4
        a3 = a3 + 5
        a16 = a15 - 4
        if (...) go to B#4
B#6     a1 = a3 + a4
        a2 = a2 + a1
        go to B#3
B#4     a8 = a8 - 5
        a14 = a12 + a13
        if (...) go to B#5
B#1     while (...)
          a1 = a4 + a3
          a4 = a3 + 1
```

```
          a8 = a10
        go to B#3
B#5     a12 = a12 + a20 * a20
        a20 = a12 - 3
B#3     a10 = a3 + a4
        a17 = a20 + a8
        if (...) go to B#2
```

For simplicity, in this graph, the weights are abstracted out. Instead, the basic blocks are labeled such that a basic block with label `B#i` is more frequently executed than a basic block with label `B#j` if `i<j`; that is, we assign the basic block labels to indicate the relative execution frequencies (for ease of presentation). In our current implementation, we run the code with sample inputs and determine the execution frequency of each basic block.

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. We then 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 have 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. Afterward, 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 postprocessing pass.

Let us consider our current example given above. We start the optimization process with `B#1` (as we assumed that this is the most frequently executed basic block). An application of Liao's heuristic [1996] to this basic block generates a maximum weight cover which gives the first storage subsequence as `a1, a3, a4, a10, a8` (see Figure 8,i). Subsequently, we move to `B#2`. Considering the storage pattern just found (when optimizing `B#1`), we interchange the order of two statements and apply commutativity transformation to one of the

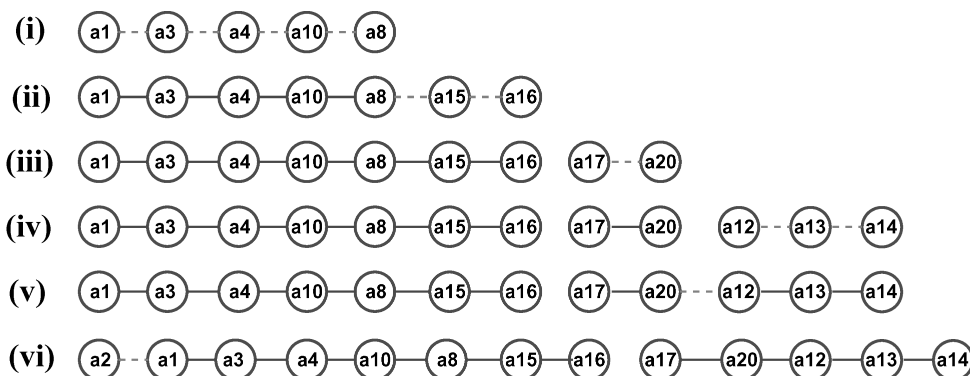


Fig. 8. Intermediate (storage) subsequences from an optimization. The incremental changes are marked using dashed edges whereas the solid edges denote the storage subsequence(s) inherited from optimizing previous basic blocks.

assignment statements. The resulting basic block is as follows:

**B#2**
```
a3 = a3 + 5
a8 = a4 + a10
a16 = a15 - 4
if (...) go to B#4
```

After that, we determine a storage subsequence involving a16, a15, and a8, considering this transformed basic block. Simply, a15 should be stored immediately after a8 and a16 should be stored immediately after a15. Figure 8,ii shows the updated storage sequence. Next, we move to optimize B#3. Again, taking into account the storage subsequences determined so far, the compiler first attempts to optimize the access sequence of this basic block. In this specific case, it uses only commutativity transformation. The transformed basic block is:

**B#3**
```
a10 = a3 + a4
a17 = a8 + a20
if (...) go to B#2
```

and the latest form of the storage sequence is depicted in Figure 8,iii. It should be noted that, although from the perspective of basic block B#3, it would be best to store a8 and a20 consecutively in memory, since during the optimization of B#1 and B#2 we decided to make a10 and a15 neighbors of a8, this requirement of B#3 cannot be satisfied. Consequently, the only incremental addition in Figure 8,iii with respect to Figure 8,ii is the storage subsequence of a20, a17. A similar problem occurs when we move to optimize the basic block B#4. For this basic block, the best alternative would be storing a12 or a13 next to a8, but both of a8's neighbors have already been fixed earlier in the optimization process. The entire storage sequence following the optimization of B#4 is illustrated in Figure 8,iv. The code of this basic block remains unmodified. In optimizing B#5, we determine that a12 and a20 should be brought together in memory (see Figure 8,v). Finally, when we move to optimize B#6, we apply commutativity transformation to the first assignment statement and make a1 and a2 consecutively accessed. The updated storage sequence is depicted in Figure 8,vi. The

transformed basic block is as follows:

**B#6**
```
a1 = a4 + a3
a2 = a2 + a1
go to B#3
```

At this point, we have visited and optimized all basic blocks. From Figure 8,vi, we can distinguish two disjoint clusters (storage subsequences). The variables within each cluster should be stored in consecutive memory locations, as dictated by the edges between them. The positions of clusters in memory with respect to each other may not be as important (they can even be stored in storage locations that are far apart). In our example, one possibility is connecting the two clusters by storing `a16` and `a17` consecutively in memory. Our current implementation, however, applies a postprocessing pass and determines the best way of connecting the clusters. The idea is to select a cluster first and then consider which one of the remaining clusters is the most suitable candidate to be attached to this cluster. This is accomplished by traversing the code and checking the number of times that the end points of other clusters are accessed immediately after or immediately before the endpoints of the cluster in question. After choosing a candidate and connecting the two clusters, we apply the same strategy recursively to other clusters. It should be emphasized that our cluster-combining strategy is just one possible alternative; it is possible to develop more sophisticated strategies. However, our experience shows that, as far as the memory layout is concerned, the most important issue is the placement of the variables within a cluster; relative placements of clusters with respect to each other are of secondary importance.

An advantage of our approach is that it first optimizes the most frequently executed basic blocks (when we have no storage constraint) from which we expect the most benefits. For example, in an application that contains several nested loops, the proposed technique first optimizes the inner loop bodies where a large portion of the execution time is expected to be spent. The overall complexity of our approach is $O(K(ElogE+L)+KL^3)$, where $K$ is the number of basic blocks, $L$ is the maximum length of any access sequence in any basic block, and $E$ is the maximum number of edges in access graph of any basic block. Here, the term $ElogE+L$ is because of storage pattern detection, and $L^3$ is because of access pattern optimization. Assuming that $K \gg L$ and $K \gg E$, we can express the overall complexity as $O(K^4)$.

## 6. INTERPROCEDURAL OPTIMIZATION STRATEGY

In this subsection, we show how our strategy can be made to work in an interprocedural setting. This is important because real embedded applications usually contain multiple procedures. To employ our approach in an interprocedural setting, we need a strategy to resolve the cases where different procedures demand different storage sequences for the same set of variables. We attempt to achieve this using a two-pass optimization strategy that operates on the call graph representation of the application. A call graph [Muchnick 1997] is a directed graph, $G = (V, E)$. The finite set of nodes, $V$, consists of the procedures that may be called in the program. For any two procedures (nodes) $f$ and $g$

in $V$, if there is a potential call to $g$ by $f$ then the (directed) edge $(f, g)$ appears on the graph. The complete collection of edges is denoted by $E$. A root node of a call graph $G = (V, E)$ is a procedure, which is not called by any other procedure; it corresponds to the main procedure in a C program. While, if desired, the edges of the call graph can be marked using the information about parameters passed/results returned, in this work, it is sufficient to work with a plain (unmarked) call graph. Since call graph traversal is a standard compiler technique, we give only a high-level view of our interprocedural strategy here.

Our optimization strategy operates on the call graph representation of the program and makes two passes over it: a bottom-up pass and a top-down pass. The bottom-up pass proceeds from the leaves of the call graph to its root, whereas, the top-down pass traverses the call graph from the root to the leaves.

The bottom-up pass works as follows. First, each procedure is analyzed and a local access graph is built for it using Liao's approach. This CLTG is, in fact, a combination of the individual access graphs (one per basic block). While it is possible to operate on this local access graph directly and determine the storage assignments (i.e., variable layouts), such an approach may not be very suitable as other procedures can also manipulate the same variables and different procedures might demand different storage assignments. Therefore, instead of operating on local access graphs right away, our approach propagates them up in the call graph. That is, after building a local access graph, the procedure passes it to its parent(s). However, during this propagation, the compiler also performs necessary variable mappings (between actual parameters and formal parameters). It is to be noted that, in general, such mappings require merging the nodes. As an example, suppose that a procedure (called `P1`) has two formal parameters `z1` and `z2`. Assume further that this procedure is called using the actual parameters `z3` and `z4` by another procedure (named `P2`). In processing `P1`, our approach builds a local access graph that uses `z1` and `z2`. When this graph is passed to `P2`, the nodes are renamed as `z3` and `z4` and are merged with the `z3` and `z4` nodes that are already in the local graph of `P2`. In this way, each procedure (once it gets the local access graphs from its children), it combines them with its own local access graph, and passes the resulting combined access graph to its parents. Thus, at the end of this bottom-up pass, we have at the root (that is, the main procedure) a large combined access graph. It should be noted, however, that the local variables in a given procedure are not considered in the local access graph; instead, the local access graph involves only the global variables and formal parameters. The reason for this is that the best storage assignment decisions for local variables can be done only within the procedure that declares them, not at the root; thus, there is no need to propagate them up.

In the next step, we use Liao's heuristic and solve this combined graph at the root and determine the storage assignments for all variables that reach to the root as well as its (the root's) local variables. Following this, our top-down pass (over the call graph) begins. In the top-down pass, first, the root passes the storage assignments it has determined to its children. In obtaining these storage assignments, we make use of the strategy discussed in Section 5 to determine the best code transformations (that satisfy the storage sequences found by the

root) and the storage sequence for its local variables. This is repeated by each procedure until we reach and process the leaves of the call graph after which the algorithm terminates.

It should be noted that our approach does not employ optimizations such as procedure inlining [Waddell and Dybvig 1997; Muchnick 1997] or cloning [Muchnick 1997]. This is because, while such optimizations might be very effective from the performance angle, they also increase code size (in some cases very significantly). Therefore, they should be applied with care in embedded systems. As a part of our future work, we will focus on integrating our approach with inlining/cloning taking into account memory space limitations.

## 7. EXPERIMENTAL RESULTS

### 7.1 Simulation Environment

In this section, we present experimental data showing the efficacy of our approach. We report experimental data at the basic block level, procedure level, and whole program level for illustrating the effectiveness of our algorithms at the basic block (Section 3), procedure (Section 5), and whole program (Section 6) levels.

Our experimental environment includes a simulator for Texas Instrument's SM320C6201, which is a high-performance fixed-point digital signal processor (DSP) with a 6.7 ns instruction cycle time. It has eight functional units, a 512K-Bit Internal Program Cache, and a 512K-Bit Dual-Access Internal Data Cache. The architecture we simulated has 32 general-purpose registers (later we change this value to conduct a sensitivity analysis). To obtain the optimized versions of the codes in our experimental suite, the SUIF back-end [Robert P. Wilson et al. 1994] is modified to generate code for the SM320C6201, which is then fed to the simulator. Before our optimization pass is activated, the input code has been optimized using high-level (e.g., loop and data transformations for cache locality [Wolfe 1996; Kandemir 2001]) as well as low-level (e.g., instruction scheduling, dead code elimination, common subexpression elimination, and global register allocation [Briggs 1992]) optimizations. In particular, we have performed all low-level optimizations that would be performed by a commercial compiler for SM320C6201. More specifically, we have paid attention to include VLIW architecture specific optimizations, such as software pipelining, data path partitioning, instruction packing, unrolling and other low-level loop optimizations (e.g., loop-invariant code hoisting), and a limited form of predication. These optimizations try to make sure that there are no superfluous variables in the code fed to our optimization. Note that all optimizations other than address register assignment have been used for original (the base versions) and optimized codes in exactly the same way. The only difference between the original and optimized codes is that the latter use the address register assignment strategy discussed in this work.

We consider seven embedded benchmark codes from MediaBench [Lee et al. 1997]: djpeg (decompression for still images), cjpeg (compression for still images), adpcm (adaptive audio coding), mpeg2decode (decompression for video),

| Benchmark | Code Size | Number of Basic Blocks | Execution Cycles | Procedures Name | Size | Basic Blocks Id | Size |
|---|---|---|---|---|---|---|---|
| djpeg | 492356 | 1751 | 6258317 | keymatch | 1176 | 3 | 48 |
| | | | | parse_switches | 2816 | 4 | 48 |
| cjpeg | 456692 | 1997 | 16526121 | set_sample_factors | 1914 | 2 | 52 |
| | | | | parse_switches | 2810 | 6 | 48 |
| adpcm | 645720 | 119 | 18336124 | rawdaudio | 9847 | 5 | 32 |
| | | | | rawcaudio | 2928 | 2 | 56 |
| mpeg2decode | 422480 | 1379 | 205958574 | Spatial_Prediction | 17420 | 3 | 64 |
| | | | | Spatial_Prediction | 17420 | 9 | 60 |
| mpeg2encode | 472108 | 3420 | 1362069719 | transform | 4016 | 5 | 52 |
| | | | | motion_estimation | 17122 | 6 | 48 |
| rasta | 883924 | 2031 | 42233738 | do_mapping | 2573 | 4 | 68 |
| | | | | comp_Jah | 2038 | 8 | 56 |
| g.721 | 693648 | 428 | 288220693 | g.721_encoder | 27868 | 8 | 56 |
| | | | | g.721_decoder | 27860 | 13 | 52 |

Fig. 9. Benchmark codes used in the experiments and their important characteristics (all size values are in bytes).

mpeg2encode (compression for video), rasta (speech recognition algorithm), and g.721 (CCITT voice compression). MediaBench is a suite of embedded media benchmarks. Figure 9 gives some useful information about these benchmarks. The second column gives the code size for each benchmark in bytes. The third column and fourth column show, respectively, the number of basic blocks in the code and execution cycles. The next two columns present information about the procedures for which we report separate experimental data. The fifth column gives the procedure name and the next column gives its size in bytes. The last two columns present information about the basic blocks for which we present separate data. These columns give the basic block number (its Id in the control-flow graph) and its size in bytes.

## 7.2 Code Size Improvement

We start by presenting the basic block results in Figure 10. The graph in this figure shows code sizes for fourteen different basic blocks randomly selected from our benchmarks (each benchmark contributes two basic blocks). We assume a default storage sequence (order) in which the variables are stored in memory in the order of their first accesses within the basic block. Specifically, if the first access to variable $a$ occurs in the code earlier than the first access to variable $b$, $a$'s location in memory has a lower address than $b$'s location. All bars in Figure 10 represent basic block sizes *normalized* with respect to that of the original basic blocks (given in the last column of Figure 9). For each basic block, we experimented with five different optimized versions: `L` gives the normalized code size resulting from Liao's approach. `L + P` is the same as `L` except that in breaking ties (when there exist multiple edges to choose from) it uses the heuristic proposed by Leupers [Leupers and Marwedel 1996]. `L + RP` gives the result when Liao's approach is followed by Rao and Pande's [1999] heuristic, which uses only intrastatement transformations, such as commutativity and associativity. `CT` is our strategy, based on computation transformation (Section 3). Finally, `L + CT` is Liao's approach followed by our method in Section 3. From the results in Figure 10, we observe that the average (normalized) basic block sizes because of `L`, `L + P`, `L + RP`, `CT`, and `L + CT` are 76.10,

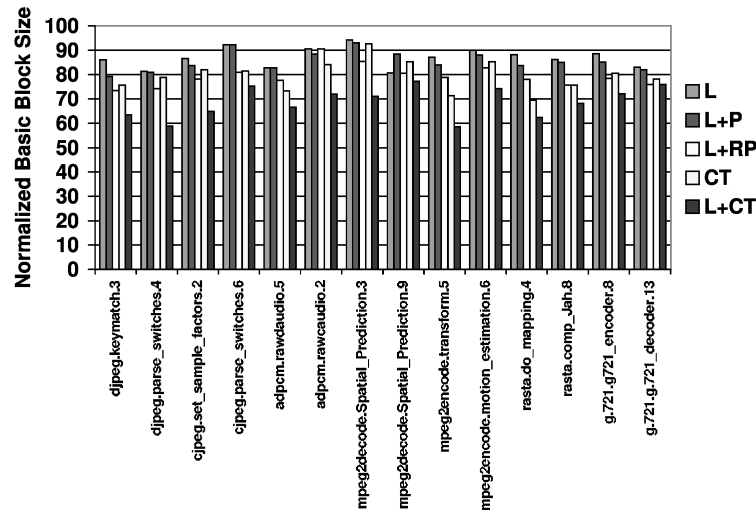


Fig. 10. Normalized basic block sizes. In the x-axis, $x.y.z$ denotes the $z$th basic block of procedure $y$ in benchmark $x$. All values are normalized with respect to the size of the original basic blocks.

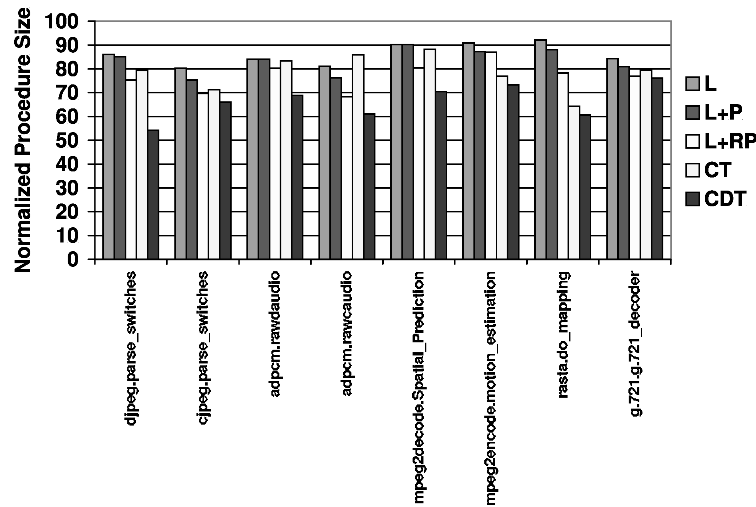


Fig. 11. Normalized procedure sizes. In the x-axis, $x.y$ denotes procedure $y$ in benchmark $x$. All values are normalized with respect to the size of the original procedures.

74.77, 69.43, 69.91, and 60.01%, respectively. One can also see that while there are cases where CT does not perform very well (as it does not optimize variable storage sequence), when it is combined with Liao's approach (that is, L + CT) it generates very good results. These results clearly emphasize the importance of optimizing both memory layout and access patterns (even at the basic block level).

We next move to the procedural level optimization and present in Figure 11 the procedure sizes for a total of eight procedures selected from our seven

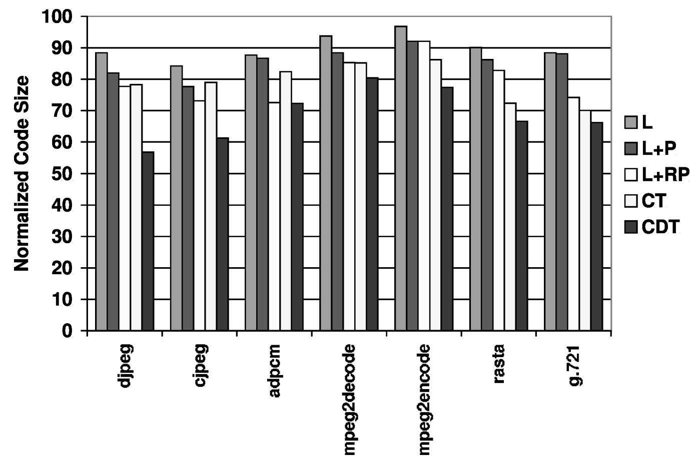


Fig. 12. Normalized benchmark code sizes. All values are normalized with respect to the size of the original benchmarks.

benchmarks. As before, these values are *normalized* with respect to those of the original codes (procedures) without any size optimization (see the sixth column in Figure 9). We conducted experiments with five different versions. L, L + P, L + RP, and CT are the same as have been discussed above except that L is the result of Liao's procedure-wide offset assignment approach, as explained in his Ph.D. thesis [Liao 1996]. CDT is the result of our intraprocedural strategy detailed in Section 5. Note that while CT only uses computation transformations, CDT uses both computation and layout optimizations. The normalized procedure sizes because of L, L + P, L + RP, CT, and CDT are 86.11, 83.38, 77.02, 78.61, and 66.32%, respectively. We see that combining computation and data transformations is very important at the procedure level.

Next is the presentation of the results when we target an entire benchmark. Figure 12 gives the normalized code sizes at the whole application level for our seven benchmarks. We use the same five versions discussed in the previous paragraph. However, CDT here corresponds to the interprocedural strategy discussed in Section 6. All bars in this figure are *normalized* with respect to values in the second column of Figure 9. These results show that using both access sequence and storage sequence transformations generates much better results than pure storage sequence based or pure access sequence based optimizations (the average percentage reductions in normalized code sizes are 21.33, 30.80, and 39.86%, for L, CT, and CDT, respectively). Note that CT performs better than L. This is because determining suitable storage orders for variables when the entire application is considered is difficult, as different parts of the application can demand different storage orders for the same set of variables. In contrast, each part of the application can be optimized independently using computation transformations (CT). CDT attempts to capture the interaction among different procedures (in the same application) by propagating access graphs over the call graph. One might also consider an alternative (and easier to implement) strategy which can be described as follows. We can first run the intraprocedural

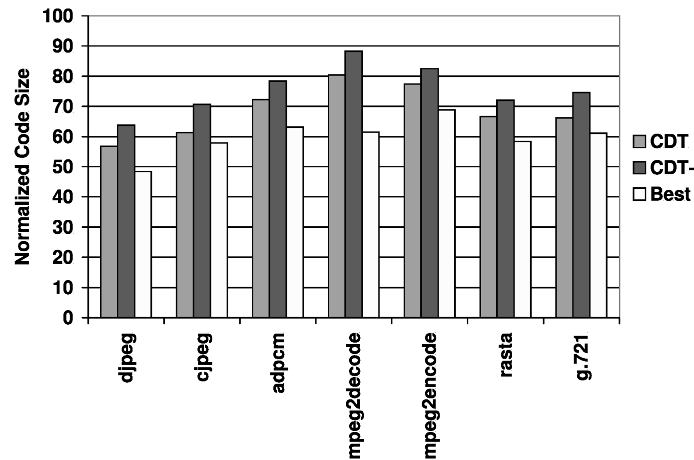


Fig. 13. Influence of interprocedural analysis and the theoretical best.

optimization strategy of Section 5 for each procedure. This gives us a storage sequence for each procedure (note that these storage sequences are potentially different from each other). Then, whenever we move from one procedure to another (i.e., visit a call graph edge), we can reshuffle the storage sequence (i.e., change the layout of variables). The downside of this strategy is that these reshufflings increase both code size (as the compiler needs to insert code to implement them) and execution cycles. The graph in Figure 13 compares this strategy (denoted by `CDT-`) with `CDT`. As before, each bar is *normalized* with respect to the code size of the original benchmark (given in the second column of Figure 9). The normalized code sizes of `CDT` and `CDT-` are 60.14 and 66.28%, respectively, indicating that performing interprocedural analysis is important. In fact, all seven benchmarks benefit from interprocedural, analysis, to some extent.

Thus for all results have been obtained using a single address register. To quantify the impact of using multiple address registers, we conducted experiments using 2, 4, 8, and 16 address registers. Figure 14 gives the impact of using different number of address registers with `CDT`. Each point in this graph corresponds to the average (across all benchmarks) code size *normalized* with respect to that of the original codes. We see that, in general, the best reductions are obtained when the number of address registers is two (an exception is mpeg2encode, where using four address registers generates the best result). Increasing the number of address registers further does not help, because the initializations of these registers start to take a large percentage of code space for a given basic block. However, we can also observe from Figure 14 that, except for one benchmark (mpeg2decode), we achieve at least 15% reduction in code size even when 16 address registers are employed, which indicates that our strategy is effective across different numbers of address registers.

While it is encouraging to see that our approach brings large savings in code size, it is also important to quantify how close it comes to the best possible optimized code. Unfortunately, it is very difficult to generate a code by hand or

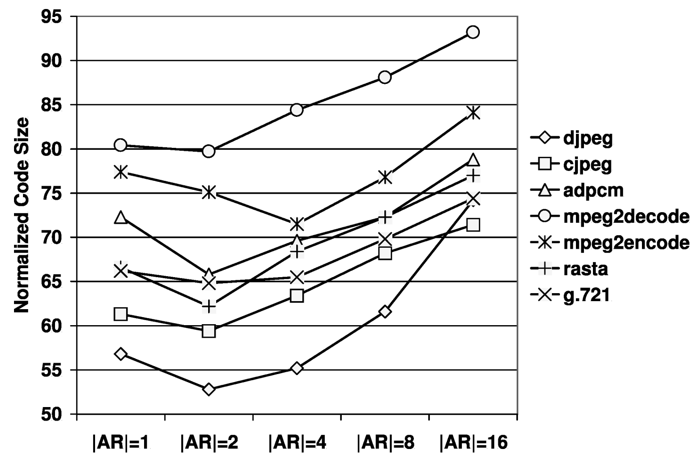


Fig. 14. Normalized code sizes with different number of address registers when our approach is used ($|AR|$ means the number of address registers).

compiler, and claim that it is optimal, as there are many different ways that explicit assignments to address registers can be eliminated. Therefore, the last bar (for each benchmark) in Figure 13 (denoted `Best`) gives the theoretical best, a result obtained by eliminating all explicit address register assignments from the code generated by `CDT`. Obviously, it is not possible to create such a legal code (i.e., a code without any address register assignments); however, it gives us a lower bound to compare. We see that, as compared to the `CDT` version, `Best` brings 8% more improvement in code size.

In our next set of experiments, we measure our code size savings with different numbers of general-purpose registers. Recall that the default numbers of general-purpose registers used in our experiments so far was 32. The graphs in Figure 15 plot the normalized code sizes for the entire benchmarks with varying number of general-purpose registers for the `CT` (top) and `CDT` (bottom) schemes. In all these experiments, we use only a single address register. We see from these results that as the number of general purpose registers is increased, we observe a drop in the effectiveness of our code restructuring-based approach. This is because, as we increase the number of registers, more variables are captured within the register file, and this reduces the opportunities for our approach, as has been discussed earlier. However, it is encouraging to see that, even with 128 registers, the average code size savings are 14.04 and 25.16% with `CT` and `CDT`, respectively. These results clearly show that the proposed approach is successful even with large register files.

## 7.3 Execution Time Impact

Our main focus in this work is on reducing code sizes of applications. However, as we discussed earlier, reducing the number of explicit load operations to address registers might also reduce the execution time as there are fewer instructions to execute. However, by reducing the number of address register instructions, we may also adversely affect the effectiveness of some low-level

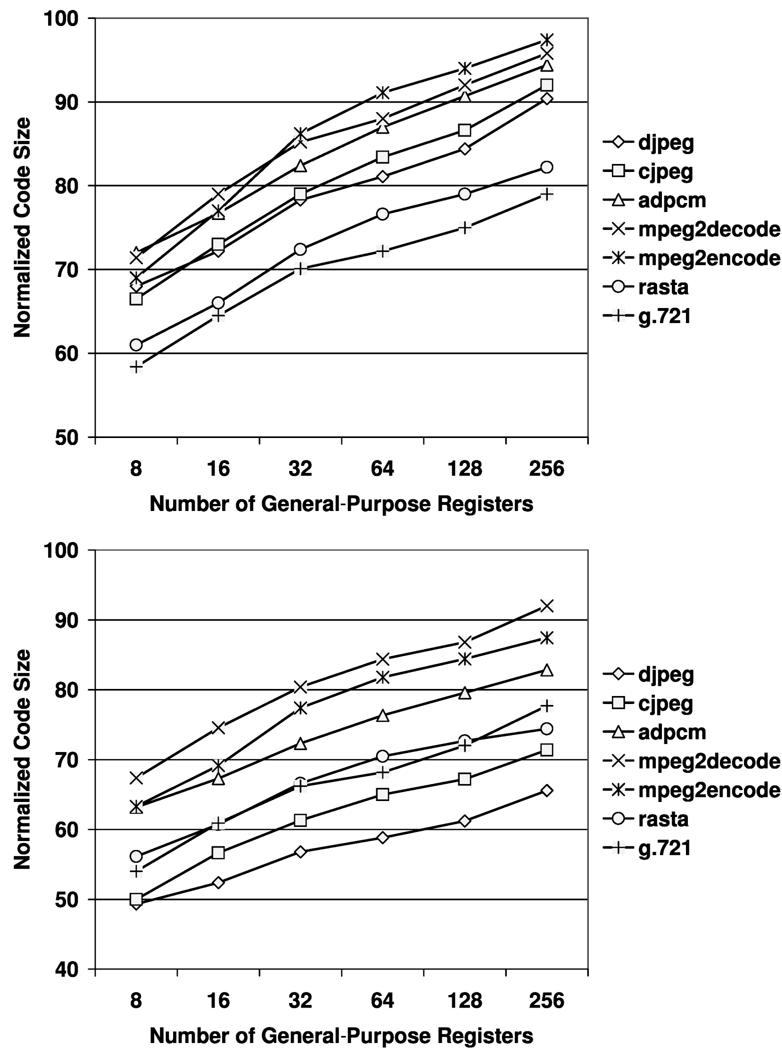


Fig. 15. Normalized code sizes with different number of general-purpose registers. *Top*: CT scheme; *bottom*: CDT scheme.

optimizations. To evaluate this tradeoff, we performed another set of experiments where we measured the execution cycles of our benchmark codes with and without the use of our optimization (CDT) as well as other strategies evaluated in this work. Table I presents the execution cycles (for the single address register case) for different versions and all benchmark codes in our suite. Each value in this figure is *normalized* with respect to the corresponding value in the fourth column of Figure 9. The average reductions (over all benchmarks) in execution cycles with L, L + P, L + RP, CT, and CDT are 4.06%, 5.54, 4.93, 6.57, and 10.77%, respectively. Note that these values are not as good as the savings in code space as our target architecture is a VLIW machine and can accommodate some of the explicit loads in unused execution slots.

Table I. Normalized Execution Cycles for Different Optimized Versions Compared to Original Codes (see Fourth Column of Figure 9)

| Benchmark | Versions | | | | |
|---|---|---|---|---|---|
| | L | L + P | L + RP | CT | CDT |
| djpeg | 96.62 | 94.17 | 93.28 | 93.96 | 88.16 |
| cjpeg | 92.08 | 91.18 | 98.27 | 91.95 | 86.14 |
| adpcm | 95.21 | 95.05 | 93.16 | 94.98 | 93.27 |
| mpeg2decode | 97.07 | 95.28 | 94.88 | 94.51 | 91.20 |
| mpeg2encode | 98.63 | 94.79 | 99.52 | 94.60 | 90.36 |
| rasta | 96.51 | 95.67 | 94.37 | 92.19 | 87.34 |
| g.721 | 95.47 | 95.11 | 92.00 | 91.80 | 88.11 |

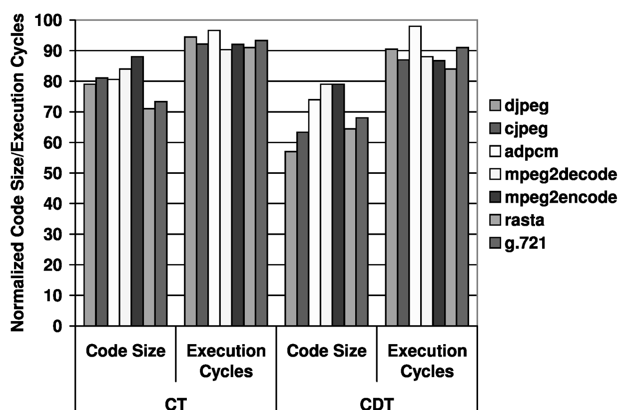


Fig. 16. Comparison with the commercial compiler.

## 7.4 Comparison with Commercial Compiler

Recall that all the energy and performance savings reported so far were normalized with respect to a base version explained in Section 7.1. While this base version implements almost all low-level and high-level optimizations known to be effective for array-intensive applications, it is also important to compare our approach against the commercial compiler. Figure 16 presents our code size and execution cycle improvements, normalized with respect to the corresponding values achieved by the commercial compiler for SM320C6201. The commercial compiler is run with the -msl1 and -O1 options (we have not used the -O2 option since it increased the code size dramatically for all benchmark codes in our experimental suite). With the -O1 option, the compiler performs local common subexpression elimination, copy/constant propagation, loop rotation, and removal of unused code. With the -ms1 option, it tries to reduce code size (through address register assignment and other optimizations). Since the commercial compiler can use only a single address register, we report the results with 1 address register and 32 general-purpose registers. We see from the results in Figure 16 that our savings with respect to the commercial compiler are similar to those against the base version used so far. This is because both the base version and the commercial compiler employ a similar set of code/data

Table II. Percentage Increase in Compilation Times

| | Versions | | | | |
|---|---|---|---|---|---|
| Benchmark | L | L + P | L + RP | CT | CDT |
| djpeg | 41.81 | 44.37 | 70.16 | 38.27 | 88.14 |
| cjpeg | 67.70 | 71.10 | 98.43 | 72.03 | 127.63 |
| adpcm | 53.56 | 58.04 | 78.87 | 61.27 | 146.04 |
| mpeg2decode | 58.05 | 61.15 | 70.20 | 70.93 | 167.13 |
| mpeg2encode | 77.82 | 80.02 | 113.55 | 75.24 | 154.77 |
| rasta | 49.52 | 52.88 | 72.84 | 68.49 | 122.98 |
| g.721 | 64.17 | 69.14 | 95.11 | 80.18 | 172.06 |

optimizations. Consequently, while the actual values may change, the trends we observe in Figures 12, and 16, and Table I are similar, emphasizing the impact of code restructuring for signal processors.

### 7.5 Increase in Compilation Time

Table II presents the percentage increase in compilation times as a result of code size optimization. The percentage increases are with respect to the original compilation times (i.e., the compilation times of the codes without any code space optimization, but with all other high-level and low-level optimizations). We see that the average increases in compilation times because of `L`, `L + P`, `L + RP`, `CT`, and `CDT` are 58.94, 62.38, 85.59, 66.63, and 139.82%, respectively. It should be emphasized that long compilation times in embedded computing are not as problematic as they are in general-purpose computing. This is because a given embedded system typically runs a single (or a small set of) application(s), and long compilation times can be compensated for by improved code and implementation quality of high-volume products.

## 8. SUMMARY AND FUTURE WORK

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. Our experimental results on basic blocks, procedures, and whole benchmarks from the MediaBench suite have shown the effectiveness of our solution in reducing code sizes. We have also found that our approach is beneficial from the performance angle as well. 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 perform experiments with different architectures as different instruction set architectures (ISA) can lead to different code sizes [Davidson and Vaughan 1987].

## REFERENCES

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

BRIGGS, P. 1992. Register allocation via graph coloring. Tech. Rep. (Apr.) Ph.D. Thesis, Computer Science Department, Rice University, Houston, TX.

CINTRA, M. AND ARAUJO, G. 2000. Array reference allocation using ssa-form and live range growth. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

DAVIDSON, J. W. AND VAUGHAN, R. A. 1987. The effect of instruction set complexity on program size and memory performance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 60–64.

ECKSTEIN, E. AND KRALL, A. 1999. Minimizing cost of local variables access for dsp-processors. In *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems*. 20–27.

ROBERT P. WILSON, ROBERT S. FRENCH, CHRISTOPHER S. WILSON, SAMAN P. AMARASINGHE, JENNIFER M. ANDERSON, STEVE W. K. TJIANG, SHIH-WEI LIAO, CHAU-WEN TSEENG, MARY W. HALL, MONICA S. LAM, AND JOHN L. HENNESSY 1994. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices 29,* 12 (Dec.), 31–37.

KANDEMIR, M. 2001. A compiler technique for improving whole program locality. In *Proceedings of the 28th Annual ACM Symposium on Principles of Programming Languages*.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th International Symposium on Microarchitecture*. 330–335.

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

LIAO, S. 1996. Code generation and optimization for embedded digital signal processors. Tech. Rep. Ph.D. Thesis, MIT, Cambridge, MA.

LIAO, S. Y., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. 1995. Storage assignment to decrease code size optimization. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 186–195.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

RAO, A. AND PANDE, S. 1999. Storage assignment optimizations to generate compact and efficient code on embedded dsps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

UDAYANARAYANAN, S. AND CHAKRABARTI, C. 2001. Address code generation for dsps. In *Proceedings of the 38th Design Automation Conference*.

WADDELL, O. AND DYBVIG, R. K. 1997. Fast and effective procedure inlining. In *Proceedings of the 4th International Symposium on Static Analysis*. Springer-Verlag Lecture Notes in Computer Science, vol. 1302. 35–52.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison Wesley, Reading, MA.

Received April 2003; revised March 2005; accepted July 2005