Global Array Reference Allocation

GUIDO ARAUJO and GUILHERME OTTONI
IC-UNICAMP
and
MARCELO CINTRA
Mindspeed Technologies Inc.

Embedded systems executing specialized programs have been increasingly responsible for a large share of the computer systems manufactured every year. This trend has increased the demand for processors that can guarantee high-performance under stringent cost, power, and code size constraints. Indirect addressing is by far the most used addressing mode in programs running on these systems, since it enables the design of small and faster instructions. This paper proposes a solution to the problem of allocating registers to array references using auto-increment addressing modes. It extends previous work in the area by enabling efficient allocation in the presence of control-flow statements. The solution is based on an algorithm that merges address registers' live ranges pairwise. An optimizing DSP compiler, from Mindspeed Technologies Inc., is used to validate this idea. Experimental results reveal a substantial improvement in code performance, when comparing to a combination of local auto-increment detection and priority-based register coloring.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *code generation*; *optimization*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Address registers, auto-increment addressing modes, DSPs, register allocation

1. INTRODUCTION

Embedded programs, like those used in audio/video processing and telecommunications, are playing a crescent role in computing. Due to its performance and code size constraints, many embedded programs are written in assembly, and run on specialized processors and/or commercial CISC machines. The increase in the size of embedded applications has put a lot of pressure toward the

A preliminary version of this paper was presented at ACM SIGPLAN LCTES 2000, Vancouver, B.C., Canada, in June 2000. This work was partially supported by CNPq (Proc. 300156/97-9, Proc. 141958/2000-6), ProTem CNPq/NSF Collaborative Research Project (68.0059/99-7), and FAPESP (98/06225-2-R).
Authors' addresses: G. Araujo and G. Ottoni, IC-UNICAMP, Cx. Postal 6176, Campinas, SP, 13084-971, Brazil; emails: guido@ic.unicamp.br, ottoni@acm.org; M. Cintra, Mindspeed Technologies Inc., 4000 MacArthur Blvd, k02-230, Newport Beach, CA 92660; email: marcelo.cintra@mindspeed.com.

development of optimizing compilers for these architectures. Processors that run embedded programs range from commercial CISC machines (e.g., Motorola 68000) to specialized *digital signal processors* (DSPs) (e.g., ADSP 2100 [Analog Devices 1995]), and encompass a considerable share of the processors produced every year.

Address computation takes a large fraction of the execution time for most programs. Addressing can account for over 50% of all program bits and one out of every six instructions for a typical general-purpose program [Hitchcock III 1986]. In order to speed up address computation, most embedded processors offer specialized *addressing modes*. A typical example is the auto-increment (decrement) mode,[1] which enables the encoding of very short instructions. All commercial DSPs and most CISC processors' *instruction set architectures* (ISAs) have auto-increment (decrement) modes.

Register allocation is a well-studied problem in compilers. Many of the first problems in code generation involved finding good algorithms for register allocation [Bruno and Sethi 1976; Sethi 1975; Aho et al. 1977a]. Code generation for expression trees has many $O(n)$ solutions, where n is the number of vertices in the tree. These algorithms are used in code generation for stack machines [Bruno and Sethi 1975], register machines [Sethi and Ullman 1970; Aho and Johnson 1976; Appel and Supowit 1987] and machines with specialized instructions [Aho et al. 1977b]. Global register allocation is an important problem in code generation that has been extensively studied [Chaitin 1982; Briggs et al. 1989; Chow and Hennessy 1990; Gupta et al. 1994]. Other researchers have considered the interaction of register allocation and scheduling in code generation for RISC machines [Goodman and Hsu 1988; Bradlee et al. 1991], and interprocedural register allocation [Callahan and Koblenz 1991]. The allocation of local variables to the stack-frame, using auto-increment (decrement) mode, has been studied in Bartley [1992], Liao et al. [1995], Rao and Pande [1999], Leupers and David [1998], and Eckstein and Krall [1999].

Horwitz et al. [1996] proposed the first algorithm for optimal allocation of address registers in straight line code. Global register allocation for array references has been studied before by Bodik and Gupta [1996] and Callahan et al. [1990]. In Bodik and Gupta [1996] and Callahan et al. [1990] array elements are allocated to general-purpose registers. As the loop iteration progresses, references are moved among registers in a *pipelined* fashion. As a result, only a single load instruction is required per iteration, to load the register at the entrance of the register sequence, and this can be done by a single address register. Unfortunately, most DSPs are highly constrained architectures containing very few specialized registers, making the just described approach impractical for these architectures.

Local reference allocation (LRA) is the problem of allocating address registers to array references in a basic block such that the number of address registers and instructions required to update them are minimized. LRA has been studied before in Araujo et al. [1996], Gebotys [1997], and Leupers et al. [1998]. These

[1]In this paper we use postincrement only. A generalization to include preincrement is fairly straightforward.

```
(1)  for (i = 0; i < N-2; i++) {
(2)    if (i % 2) {
(3)      avg += a[i+1] << 2;
(4)      a[i+2] = avg * 3;
(5)    }
(6)    if (avg < error)
(7)      avg -= a[i+1] - error/2;
(8)    else
(9)      avg -= a[i+2] - error;
(10) }
(11)
(12)
(13)
```

(a)

```
p = &a[1];
for (i = 0; i < N-2; i++) {
  if (i % 2) {
    avg += *p++ << 2;
    *p-- = avg * 3;
  }
  if (avg < error)
    avg -= *p++ - error/2;
  else {
    p += 1;
    avg -= *p - error;
  }
}
```

(b)

Fig. 1. (a) Code fragment; (b) modified code that enables the allocation of one register to all references.

are efficient graph-based solutions, when references are restricted to basic block boundaries. In this paper, we propose a solution to the global form of this problem, when array references are spread across different basic blocks, a problem we call *global reference allocation* (GRA).

The basic concepts required to understand LRA and GRA are described in Section 3. In Section 4 we give a short description of the existing solutions to LRA. This is followed by a general description (Section 5) of our algorithm to GRA, named *live range growth* (LRG). The LRG algorithm is based on an operator that merges register live ranges pairwise. The merge operator is described in Section 6, and its formulation and complexity analysis in Section 7. Section 8 proposes a heuristic algorithm for GRA, and Section 9 evaluates the performance of the algorithm, using a real-world optimizing DSP compiler from Mindspeed Technologies Inc. Section 10 summarizes the main results and gives future directions.

2. PROBLEM DEFINITION

GRA is the problem of allocating address registers (`ar's`) to array references such that the number of simultaneously live address registers is kept below the maximum number of address registers in the processor, and the number of new instructions required to do that is minimized. In many compilers, this is done by assigning address registers to similar references in the loop. This usually results in very efficient code, when traditional optimizations like induction variable elimination and loop invariant removal [Aho et al. 1986; Muchnick 1997] are in place. Nevertheless, assigning the same address register to different instances of the same reference does not always result in the best code. For example, consider the code fragment of Figure 1(a). If a register is assigned to each distinct reference $a[i + 1], a[i + 2]$, two address registers are required for the loop. Now, assume that only one address register is available in the processor. If we rewrite the code from Figure 1(a) using a single pointer `p`, as

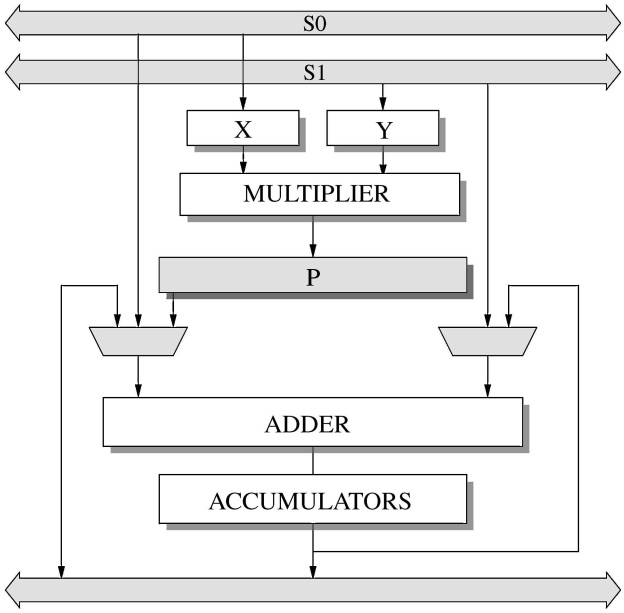

Fig. 2. The architecture `MAC` unit.

shown in Figure 1(b), the resulting loop uses only one address register that is allocated to `p`. The cost of this approach is the cost of a pointer *update instruction* (`p += 1`) introduced in one of the loop control paths, which is considerably smaller than the cost of spilling/reloading one register. The C code fragment of Figure 1(b) is a source level model of the *intermediate representation* resulting when optimal GRA allocation is performed on the code of Figure 1(a).

The target architecture used to validate the GRA approach described in this paper is an *in-house* DSP core from Mindspeed Technologies Inc. A production compiler for this processor has been used to implement our solution for GRA. Only some aspects of the architecture are described here, namely those that are related to the GRA problem. The target architecture is a typical DSP architecture featuring two `MAC` units (Figure 2) and an *address generation unit* (`AGU`) that performs memory address calculation in parallel with the other units. The `AGU` enables auto-increment operations on eight memory addressing registers, thus reducing the number of instructions required to access data in contiguous memory positions. Four *modify registers* (`mr's`) are available that can be used together with the addressing registers for memory access. From those, two `mr's` are used to handle stack and global variable addressing, one is assigned to a specialized processor feature, and the last `mr` stores -1 so as to enable auto-decrement addressing. Each `MAC` unit has two accumulators, and a typical instruction in the processor ISA takes one operand from *on-chip* memory[2] and the other from one of the accumulators. For example, instruction `add acc0, (*ar1++mr3)`, accumulates into `acc0` the data pointed to by the sum of address

[2]For some instructions, memory access is performed through pipeline operand registers `X` and `Y`.

register ar1 plus modify register mr3. The just described architecture is typical of most DSP architectures [Araujo 1997].

3. BASIC CONCEPTS

This section defines a set of basic facts that are required to formulate the GRA problem and to study its solutions. It defines the following concepts: (a) *indexing distance*, which is used to measure the cost of assigning two references to the same address registers; (b) *live range*, which is basically a set of references in the program that share the same address register.

First of all, assume for the rest of this paper that the order of the array references, with respect to each other, is known and that the array data type is a memory word (a typical characteristic of embedded programs). Moreover, references are kept atomic in the compiler intermediate representation, and array subscript expressions are not considered for *common subexpression elimination* (CSE).

A central issue in GRA is to bound the allocation of address registers to the number of address registers in the target processor. In this case sometimes it is desirable that two references share the same register. This is done by inserting an instruction between the references or by using the auto-increment (decrement) mode. The possibility of using auto-increment (decrement) can be measured by the indexing distance.

Definition 3.1. Let a and b be array references and s the increment of the loop containing these references. Let $index(a)$ be a function that returns the subscript expression of reference a. The indexing distance between a and b is the positive integer

$$d(a,b) = \begin{cases} |index(b) - index(a)|, & \text{if } a < b, \\ |index(b) - index(a) + s|, & \text{if } a > b, \end{cases}$$

where $a < b$ ($a > b$) if a (b) precedes b (a) in the schedule order.

Example 3.1. Consider, for example, the array references from lines (9) and (3) of Figure 1(a) (i.e., $a[i+2]$ and $a[i+1]$), where $s = 1$. The indexing distance $d(a[i+1], a[i+2]) = |(i+1) - (i+2) + 1| = 0 \leq 1$. Since the indexing distance is smaller/equal to 1, no update instruction is needed to update the register allocated to the address register assigned to $a[i+2]$ such that it ends up pointing to $a[i+1]$ in the next iteration.

Definition 3.1 for indexing distance can only be applied to unidimensional arrays. When nested loops or multidimensional arrays are present, the indexing distance should take into consideration the layout of the array in memory and the size of the dimensions. Similarly as in the unidimensional case, only references for which the index function at each dimension is an affine function must be considered.

In the following, assume that the size of each array element is a memory word, that arrays are stored in *row major*, and that $r[i_1] \cdots [i_n]$ an n-dimensional array reference within a loop. Each subscript index of reference r can be represented by a triple $i_k = (a_k, j_k, b_k)$, a_k and b_k are integers, and j_k is an

```
(1)   for (i=0;i<N;i++){
(2)     for (j=0;j<N;j++){
(3)       for (k=0;k<N;k++)
(4)         c[i][j] += a[i][k]*b[k][j];
(5)     }
(6)   }
```

Fig. 3. Matrix multiplication algorithm.

induction variable. Hence, reference r can be represented by a set of triples $\{(a_1, j_1, b_1), \ldots, (a_n, j_n, b_n)\}$.

Let j be an induction variable of a given loop L with step $s = 1$. Assume that the subscript index expression i_k is a linear function of induction variable j, that is, $i_k = (a_k, j, b_k)$. If $k = n$, then consecutive iterations of L result in a contiguous memory access pattern, where any two consecutive memory references to elements of r are in adjacent memory positions. In this case, auto-increment mode can be used to update the address register pointing to r. On the other hand, if $k \neq n$ then any consecutive references to r result in a sequence of accesses to memory positions that are separated from each other by an amount that depends on the size of the array dimensions that are greater than k. We call this amount the *dimensional shift* (D_k) of dimension k, and compute it (below) as the product of the size of all dimensions of r that are greater than k.

$$D_k = \begin{cases} 1, & \text{if } k = n, \\ \prod_{j=k+1}^{n} size(j), & \text{otherwise.} \end{cases}$$

where $size(j)$ is the size of dimension j. Based on the dimensional shift concept, we can now generalize the indexing distance for n-dimensional array references.

Definition 3.2. Let r_1 and r_2 be two n-dimensional array references represented, respectively, by the set of triples $\{(a_1, j_1, b_1), \ldots, (a_n, j_n, b_n)\}$ and $\{(a_1, j_1, c_1), \ldots, (a_n, j_n, c_n)\}$. The indexing distance between r_1 and r_2 is the integer quantity:

$$d(r_1, r_2) = \begin{cases} |\sum_{k=1}^{n} (c_k - b_k) * D_k|, & \text{if } r_1 < r_2, \\ |\sum_{k=1}^{n} (c_k - b_k + a_k * s) * D_k|, & \text{if } r_1 > r_2. \end{cases}$$

Example 3.2. Consider, for example, the algorithm of Figure 3, which computes the product of two matrices `a` and `b` and stores the result into matrix `c`. Assume that all matrices are bidimensional arrays where $size(d) = 50$, for any dimension $d \leq 2$. The subscript indices of all array references in Figure 3 are linear functions of the induction variables i, j, or k. Consider references `a[i][k]` and `b[k][j]` in line (4) of the inner loop. For the sake of simplicity, call these references by the name of the arrays they refer to, that is, a and b. Assume that register `ar1` (`ar2`) has been allocated to reference a (b). The indexing distance between reference a, in the current iteration, and the same reference in the next loop iteration is, according to Definition 3.2, $d(a, a) = 1$. Hence, auto-increment mode can be used to update register `ar1` in the current

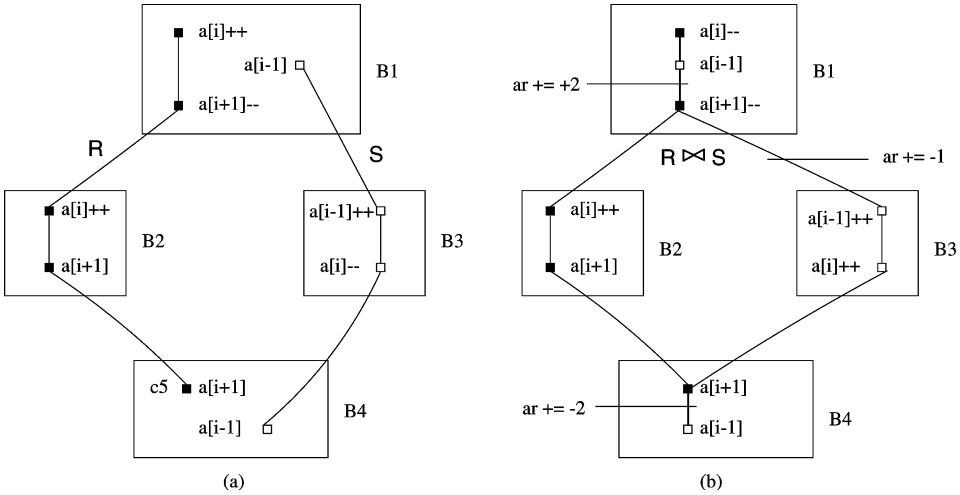

Fig. 4. (a) Two live ranges R and S; (b) a single live range is formed after joining $R \bowtie S$.

iteration such that it points to reference a in the next iteration. On the other hand, the indexing distance between b in the current iteration and b in the next iteration is $d(b, b) = |1 - 1 + 0| * 1 + |1 - 1 + 1| * 50 = 50$. Since the indexing distance between two consecutive references to b is greater than 1, auto-increment mode cannot be used for `ar2`. Therefore, an update instruction `ar2 += 50` is required, at the end of the inner loop, to redirect `ar2` such that it points to $b[k][j]$ in the next iteration.

Definition 3.3. A live range $L = \{r_1, r_2, \ldots, r_n\}$ is a set of array references that share the same address register.

The concept of live range used in this paper is a straightforward extension of idea of *variable liveness* adopted in the compiler literature [Aho et al. 1986]. The set of array references of a program can be divided into a number of subsets, each of them defining a live range. For example, in Figure 4(a) references are divided into live ranges R and S. In our notation for live range, little squares represent array references, and squares with the same color are in the same range. Moreover, an edge between two references of a live range indicates that the references are glued together through auto-increment mode or an update instruction. In Figure 4(a), symbol "++" ("−−") following a reference assigns a postincrement (decrement) mode to that reference. Notice that a live range can include many references across distinct basic blocks. The problem of dividing references in a basic block into ranges is known as the *local reference allocation* (LRA) problem and is discussed in Section 4 below.

4. LOCAL REFERENCE ALLOCATION

The goal of LRA is to allocate an address register to each array reference in a basic block, by dividing them into live ranges and assigning an address register to each range. Therefore, the final number of ranges should not exceed

the total number of address registers of the processor. Moreover, the number of instructions required to redirect registers through references should be minimum. Solutions to the LRA have been studied before in the literature.

Araujo et al. [1996] proposed an approach to the LRA problem based on the solution of a bipartite matching problem. In Araujo et al. [1996], references are associated to vertices of a graph, called the *indexing graph* (IG). There exists an edge in the IG between two references, if auto-increment/decrement can be used to redirect the address register from one reference to another. The LRA problem is then solved by determining the minimum disjoint path covering of the IG, using a bipartite matching algorithm [Boesch and Gimpel 1977]. Each path in the resulting cover corresponds to a live range that is assigned an addressing register. Araujo's approach presents a drawback when the number of ranges is larger than the number of address registers available in the processor. The ranges are merged pairwise, using a heuristic, until its number is reduced to the number of available address registers.

A way to tackle this drawback is to bound the number of live ranges in the block to the number of address registers in the processor. Gebotys [1997] proposed a formulation that does exactly this. Her technique is based on the solution of a minimum network-flow circulation problem [Tarjan 1983]. Every array reference in the basic block is associated to a vertex in the network, and there exists an edge from any two references if one follows the other in the schedule order. Every edge is assigned an unitary *edge capacity* and a *cost*. The cost is zero (one) if no (one) instruction is needed to redirect the address register pointing to the reference at the source of the edge, such that it points to the reference at its destination. A *circulation edge*[3] is added to the network graph with a capacity equal to the number of address registers available in the processor. The goal of Gebotys's algorithm is to find the maximum circulation flow that minimizes the number of update instructions (i.e., flow cost). A live range in this formulation is a network path (from source to drain) formed exclusively by edges for which the flow is nonzero. The drawback of this technique stems from the fact that it seeks to determine the minimum cost of the maximum flow (number of allocated registers), thus increasing register pressure.

The approaches to the LRA problem proposed above do not address the allocation of address registers across loop iterations. In both cases, the allocation across iterations is considered only after the covering is performed, and works as follows. For each live range resulting from the cover, take the head and tail of each range, and check if auto-increment/decrement is enough to redirect the reference at the tail (in the current iteration) to the reference at the head (in the next iteration). If this is not possible an update instruction is required to redirect the address register from tail to head. After tail and head of each range are connected (through an auto-increment/decrement mode or update instruction), the resulting range is a cycle and an address register is assigned to it. The drawback of this strategy is that cycles are considered after path covering, and there are chances that the final ranges require more update instructions than the optimal solution [Araujo 1997].

[3]A circulation edge in a network graph is an edge from the destination to the source vertex.

```
(1)  procedure LRG (R, nars)
(2)      while |R| > nars do
(3)          mincost ← +∞
(4)          for each range R ∈ R do
(5)              for each range S ∈ R, S ≠ R do
(6)                  if cost⋈(R,S) < mincost then
(7)                      mincost ← cost⋈(R,S)
(8)                      minpair ← {R, S}
(9)          R ← (R - minpair) ∪ (R ⋈ S)
```

Fig. 5. LRG algorithm.

Basu et al. [1999] proposed a solution to the LRA problem that addresses this issue. In Basu et al.'s work, a branch-and-bound algorithm is used to cover the references using cycles in the graph, while a merge operator keeps the number of paths limited to the maximum number of address registers. The path covering approach from Araujo et al. [1996] is used as a lower bound and a heuristic algorithm as an upper bound to determine the minimum number of address registers required to achieve the optimal cycle covering. Basu et al.'s approach is the most complete optimal solution known to the LRA problem.

As described above, the LRA problem has been extensively studied. On the other hand, not much research work has been done toward finding solutions for the allocation of address registers for a whole procedure (GRA). Good address register allocation for basic blocks can improve the performance of programs, but it is not enough due to the following reasons. First, the majority of the real-world programs have loop bodies with more than one basic block. The separate allocation of address registers to individual blocks decreases the possibility of register sharing among blocks, considerably increasing register pressure and spilling. Second, even for loop bodies that contain only a single block, local allocation does not handle the allocation of registers across different loop nesting levels.

5. GLOBAL REFERENCE ALLOCATION

To solve the GRA problem we developed a technique that repeatedly *merges* pairs of live ranges R and S, such that all references in the new range (called $R \bowtie S$) are allocated to the same register. The algorithm starts by partitioning all references across a set of initial live ranges. The initial set of ranges can be formed in many different ways. In this paper we studied two cases: (a) when each individual reference is assigned a separate range; and (b) when the initial ranges are the solution of solving the LRA problem to all loop blocks. In this case any of the algorithms in Section 4 could be applied.

Ranges are merged pairwise until the number is smaller than or equal to the number of address registers available in the processor. We call this technique *live range growth* (LRG). The cost of merging two ranges R and S ($cost_{\bowtie}(R, S)$) is the total number of cycles of the update instructions required by the merge. The pseudo-code for the LRG algorithm is shown in Figure 5.

In LRG, inner loops are treated first, hierarchically followed by outer loops. The algorithm is greedy and its runtime complexity is $O(n^2)$, where n is the number of references in the loop.

Despite its complexity, the experimental results (Section 9) reveal that LRG is fast (no substantial increase in runtime was detected) and considerably improves address register allocation. For example, after ranges R and S in Figure 4(a) are merged a single range $R \bowtie S$ results, shown in Figure 4(b).

In order to enable references to share the same address registers, we have to convert them to a new representation that satisfies the following requirement: any reference b should be reached by only one reference a, given that we want to compute the indexing distance $d(a, b)$ at compile time. This allows us to decide, at compile time, between using auto-increment (decrement) mode for a, or inserting an update instruction on the path from a to b. We have realized that this requirement can be satisfied if the references in the CFG are in *static single assignment (SSA) Form* [Cytron et al. 1989]. We call this variation of SSA a *single reference form* (SRF) and describe it below.

5.1 The Single Reference Form

Let G be some program *control-flow graph* (CFG) and $\mathcal{R}$ the set of array references in G. G is in SRF, with respect to $\mathcal{R}$, if each reference is reached by a single reference. Translating a CFG to SRF can be done using the so-called ϕ-functions [Cytron et al. 1989] at the join nodes of G that are relevant to $\mathcal{R}$. In order to merge ranges that converge to the same block B, we first insert, at the entry of B, a dumb ϕ-function $\phi(a, a, \ldots, a)$, with the number of argument positions equal to the number of control-flow predecessors and successors[4] of B. This function is assigned to a new reference w, resulting in the *ϕ-equation* $w = \phi(a, a, \ldots, a)$. The goal of the ϕ-equations is to sort out which references reach a given block B (see Figure 6). Notice that the result of a ϕ-equation (i.e., w) is a reference generated by an update instruction, or by the auto-increment (decrement) mode assigned to any reference that is an argument of the ϕ-function. In order to simplify the notation, from now on, we call references in the original program *real references* and references that are the result of ϕ-equations *virtual references*.[5]

Similarly as in the case of SSA translation, ϕ-equations are required only at certain basic blocks. These blocks are given by the *iterated dominance frontier* [Cytron et al. 1989] of the set that contains all references. Our work uses the algorithm for iterated dominance frontier described in Cytron et al. [1989]. For the sake of space, we do not describe here any further details on how to convert a program into its SSA form. The interested reader should look that up Cytron et al. [1989] or to modern compiler optimization books [Muchnick 1997].

The next step in our approach is to substitute ϕ-equations for update instructions whenever this is required. In order to do that, we need to compute two sets: (a) the set of references that reach ϕ-equations; and (b) the set of

[4]Successors are not considered in the SSA form.

[5]Virtual references are only a solution artifact and do not perform any access to memory.

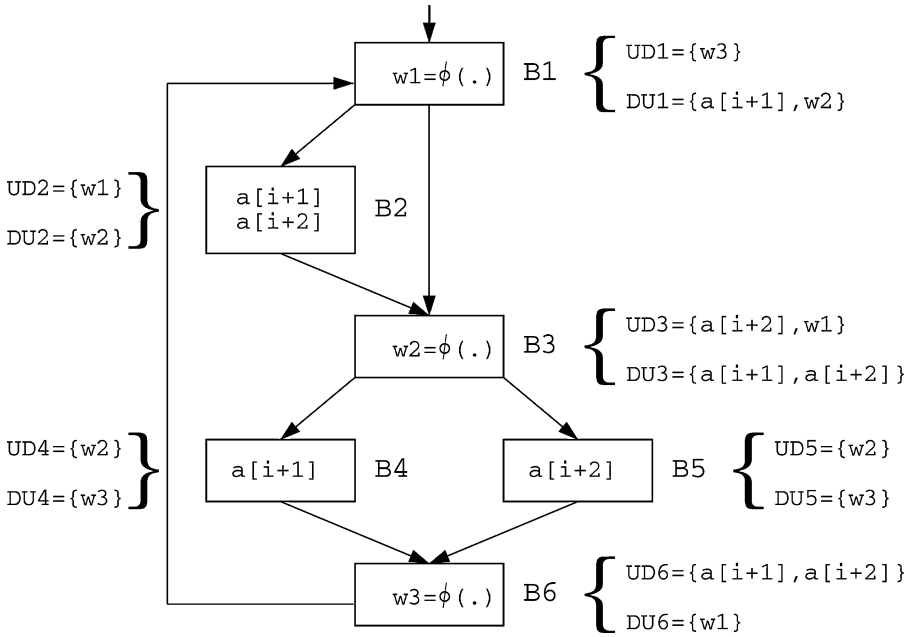

Fig. 6. CFG fragment after ϕ-instruction insertion and reference analysis.

references that are reached by the result of ϕ-equations. We call the technique used to compute these sets *reference analysis*, and base it on dataflow analysis of the program CFG [Aho et al. 1986; Muchnick 1997].

5.2 Reference Analysis

The data item used during reference analysis is the array reference. We say that statement s *generates* reference a if a is used in s, and a is a reference from the live ranges R or S that we want to merge (i.e., $a \in R \cup S$). Statement s *kills* a if it uses some other reference $b \in R \cup S$ and $b \neq a$. Based on this notation, we can determine, for each basic block B, the following dataflow sets: (a) $r\_gen[B]$, the set of references generated in B; (b) $r\_kill[B]$, the set of references killed in B. *Reference analysis* is the problem of computing the array references from $R \cup S$ that reach any given statement. It can be formulated as a *reaching definitions* problem for which there are well-known solutions [Aho et al. 1986; Muchnick 1997].

After reference analysis is performed we have, at each program point, the set of references reachable at that point. We use this to compute, for each statement s, the following sets: (a) UD_s, the *ud-chain* of the references that reach s; (b) DU_s, the set of references that use a reference defined at s. When s is a ϕ-equation, references in UD_s are used to rename the ϕ-function arguments to $\phi(a_1, a_2, \ldots, a_n, b_1, \ldots, b_j, \ldots, b_m)$, $a_i \in UD_s$ and $b_j \in DU_s$. For the sake of simplicity, we represent this ϕ-function as $\phi(UD_s, DU_s)$. For example, for block B_3 in Figure 6 $UD_3 = \{a[i+2], w_1\}$ and $DU_3 = \{a[i+1], a[i+2]\}$.

Example 5.1. Consider, for example, the CFG fragment shown in Figure 6, after ϕ-equations are inserted into blocks B_1, B_3, and B_6. The UD and DU sets for all basic blocks that perform array references are shown in Figure 6. We assume here that the program begins (ends) before (after) block B_1 (B_6). Notice that, if ϕ-equations had not been inserted, the instruction associated to reference $a[i+1]$ in B_2 would be reached (across one loop iteration) by two references, that is, $UD_2 = \{a[i+1], a[i+2]\}$. In this case, it would not be possible to determine, at compile time, which of the these two references reach $a[i+1]$, and hence if auto-increment mode could be used by their instructions to point to $a[i+1]$. After ϕ-equations are inserted, the ud-chain at B_2 becomes $UD_2 = \{w_1\}$, only one reference reaches $a[i+1]$, and thus the decision can be made once the value for w_1 is computed.

6. THE MERGE OPERATOR ($R \bowtie S$)

After a program is in SRF, any loop basic block to which array references converge will have a ϕ-equation, including the header and the tail of the loop. The LRG algorithm is a greedy algorithm that takes the initial set of references and merges them pairwise. At each step of the algorithm two live ranges are merged. The cost $cost_{\bowtie}(R, S)$ of merging any two live ranges R and S into a single range $R \bowtie S$ is used to determine which pair of ranges to merge. The pair of ranges that leads to the smallest value of $cost_{\bowtie}$ is always selected for merge. The algorithm ends when the number of live ranges is smaller/equal to the number of address registers in the processor.

Before merging two ranges R and S, we have to estimate the cost of the update instructions required to do that, so as to decide if the cost of this merge is smaller than the current minimum merge cost. Consider, for example, two live ranges R and S that join at basic block B, as shown in Figure 7(a). By definition, after the program is in SRF, each reference $b \in R \cup S$ has associated to it a set UD that contains a single reference a also from $R \cup S$. During merge, a and b are glued together by using auto-increment (decrement) mode at a or by inserting an update instruction between a and b. The cost of an update instruction is measured by the number of cycles it takes to execute, which we assume to be 1. Let DU_a be the def-use chain of the references that are reachable from a. The cost of adding an update instruction from reference a to $b \in DU_a$ can be defined as

$$cost(a, b) = \begin{cases} 0, & \text{if } |d(a, b)| \leq 1 \text{ and } a \text{ is real, or} \\ & \text{if } |d(a, b)| = 0 \text{ and } a \text{ is virtual,} \\ 1, & \text{otherwise.} \end{cases} \quad (1)$$

The first condition for the zero cost, in Equation (1), aims at detecting the possibility of assigning an auto-increment (decrement) to a real reference a. The second condition for the zero cost measures if the virtual reference (resulting from a ϕ-equation) is equal to the reference b that it reaches. In all other cases, an update instruction is required and the cost is 1.

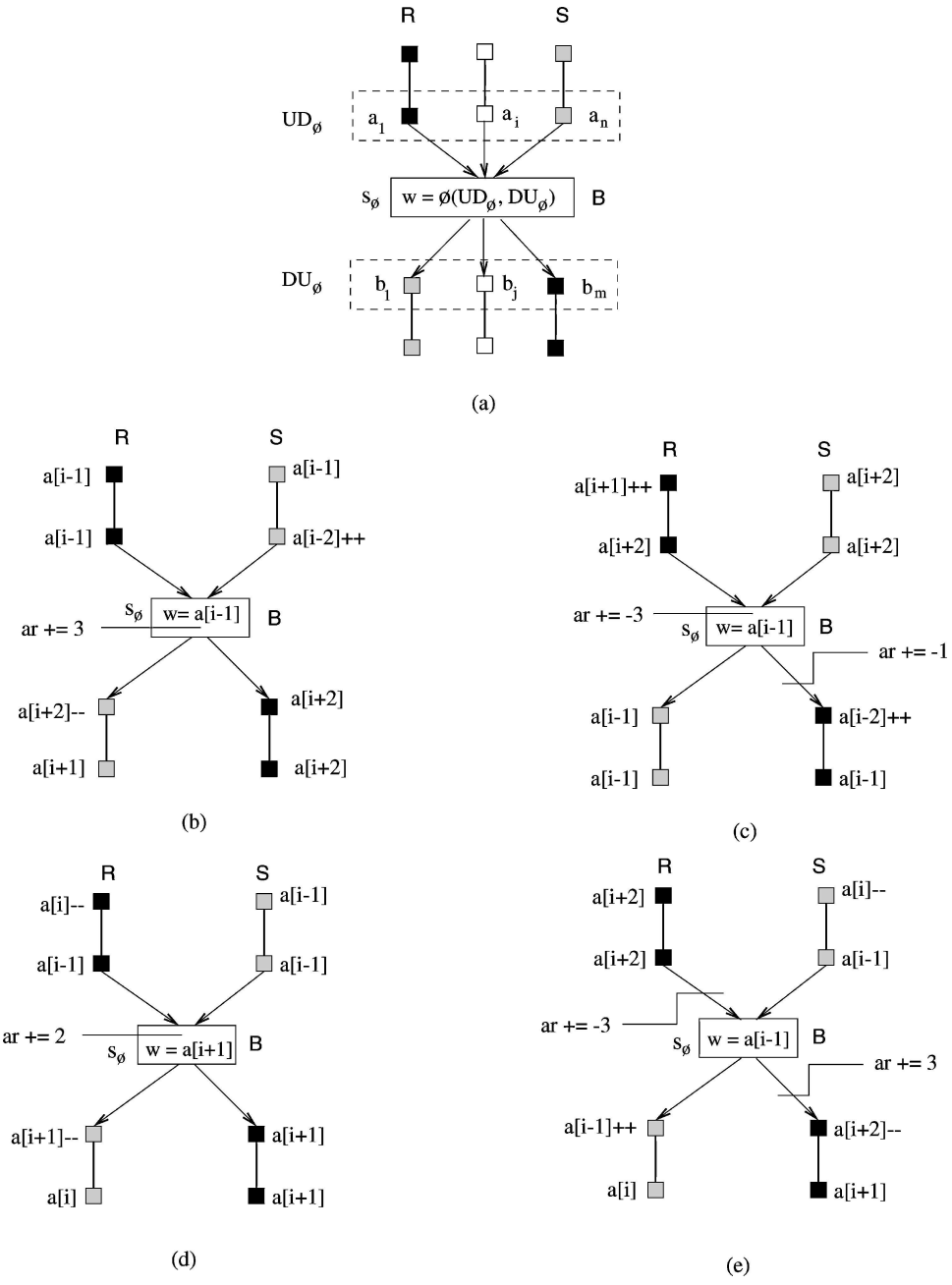

Fig. 7. Examples of update instruction insertion.

Let $Pred(B)$ ($Succ(B)$) be the set of predecessors (successors) of block B that contains ϕ-equation s_ϕ, and ar the address register assigned to $R \bowtie S$. Consider all statements s_ϕ that have at least one reference from R and S as arguments. Notice that the other arguments of ϕ are irrelevant given that we want to merge only ranges R and S. After reference analysis, any s_ϕ has associated to it sets UD_ϕ and DU_ϕ, with elements $a_i \in UD_\phi$ and $b_j \in DU_\phi$. The value of w depends on how distant the elements in UD_ϕ and DU_ϕ are. We want to select a reference w that reduces the number of update instructions required to merge all a_i to all b_j. This can be done using the following algorithm that is illustrated in Figure 7(b)–(e).

(1) If the elements $b \in DU_\phi$ are the same, but elements of UD_ϕ are different. In this case, w is the array reference that minimizes the cost $\sum_{i=1}^{|UD_\phi|} cost(a_i, w) + cost(w, b)$. Remove s_ϕ and insert into B an update instruction $ar += d(w, b)$, if $cost(w, b) = 1$. Insert, at the exit of the block $P_i \in Pred(B)$ that contains a_i, an update instruction $ar+ = d(a_i, w)$, whenever $cost(a_i, w) = 1$; otherwise, use the adequate auto-increment (decrement) mode for a_i.

Example 6.1. Consider, for example, the live ranges in Figure 7(b). Notice that $w = a[i-1]$ and all references reachable from s_ϕ are the same, that is, $b_j = a[i+2], \forall b_j \in DU_\phi$. Since $cost(a[i-1], a[i+2]) = 1$ we substitute s_ϕ by an instruction $ar+ = 3$ that redirects ar from $w = a[i-1]$ to $a[i+2]$. Reference $a[i-2]$ is assigned auto-increment mode, and thus it is rewritten to $a[i-1]$ for the future steps of the algorithm.

(2) If the elements $a \in UD_\phi$ are the same, but elements of DU_ϕ are different. In this case w is the array reference that minimizes the cost $cost(a, w) + \sum_{j=1}^{|DU_\phi|} cost(w, b_j)$. Remove s_ϕ and insert into B update instruction $ar+ = d(a_i, w)$, if $cost(a_i, w) = 1$; otherwise use the adequate auto-increment (decrement) mode for a_i. Insert, at the entry of block $S_j \in Succ(B)$ that contains b_j an update instruction $ar+ = d(w, b_j)$, if $cost(w, b_j) = 1$.

Example 6.2. Consider, for example, the live ranges in Figure 7(c). Here $w = a[i-1]$ and all references that reach s_ϕ are the same, that is, $a_i = a[i+2], \forall a_i \in UD_\phi$. Since $cost(a[i+2], a[i-1]) = 1$, we substitute s_ϕ by an instruction $ar+ = -3$ that redirects ar from $a[i+2]$ to $w = a[i-1]$. Moreover, since $cost(a[i-1], a[i-2]) = 1$, we insert $ar+ = -1$ on the path from $w = a[i-1]$ to $a[i-2]$.

(3) If the elements of DU_ϕ (UD_ϕ) are the same.
In this case, w is the reference that minimizes $cost(a, w) + cost(w, b)$. Remove s_ϕ and insert update instructions or use auto-increment (decrement) mode according to steps (1) and (2) above.

Example 6.3. Consider, for example, the live ranges in Figure 7(d). Notice that $w = a[i+1]$ and all references in UD_ϕ (DU_ϕ) are the same, that is, $a[i-1]$ ($a[i+1]$). Since $cost(a[i+1], a[i+1]) = 0$ no update instructions are required from $w = a[i+1]$ to $a[i+1]$. On the other hand, $cost(a[i-1], a[i+1]) = 1$ and thus an instruction $ar+ = 2$ is required in block B.

(4) If the elements in UD_ϕ (DU_ϕ) are not the same.
In this case, w is the array reference that minimizes $\sum_{i=1}^{|UD_\phi|} cost(a_i, w) + \sum_{j=1}^{|DU_\phi|} cost(w, b_j)$. Remove s_ϕ and insert, at the exit of the block $P_i \in Pred(B)$ that contains a_i, an update instruction $ar+ = d(a_i, w)$, if $cost(a_i, w) = 1$; otherwise, use auto-increment (decrement) mode for a_i. Insert an update instruction $ar+ = d(w, b_j)$ at the entry of $S_j \in Succ(B)$, if $cost(w, b_j) = 1$.

Example 6.4. Consider, for example, the ranges in Figure 7(e). In this case $w = a[i-1]$, and the elements of sets UD_ϕ and DU_ϕ are distinct within each set. Since $cost(a[i+2], a[i-1]) = 1$ we insert at the end of the block that contains $a[i+2]$ an update instruction $ar+ = -3$ to redirect ar from $a[i+2]$ to $w = a[i-1]$. Similarly, instruction $ar+ = 3$ is inserted at the entry of the block that contains $a[i+2]$.

7. THE MINIMUM COST $R \bowtie S$ PROBLEM

Given that the merge operation is central to the LRG algorithm, it is important to structure the problem and analyze its computational complexity. Assume that during some intermediate step in the LRG algorithm two ranges R and S are considered for merging. Consider, for the following analysis, only those references that are in $R \cup S$. The set of ϕ-equations associated to these references forms a system of equation that evaluates according to the rules discussed in Section 6. The unknown variables in this system are virtual references w_k. These equations have circular dependencies, caused basically by the following reasons: (a) they originate from a loop; (b) each ϕ-function depends on its sets UD_ϕ and DU_ϕ. Therefore, any optimal solution for variables w_k cannot always be determined exactly. Consider, for example, the CFG of Figure 8 corresponding to the loop in Figure 1(a), after ϕ-equations are inserted and reference analysis is performed. Three ϕ-equations result in blocks B_1, B_3, and B_6, namely:

$$s1_\phi : w_1 = \phi(w_3, a[i+1], w_2), \tag{2}$$

$$s2_\phi : w_2 = \phi(a[i+2], w_1, a[i+1], a[i+2]), \tag{3}$$

$$s3_\phi : w_3 = \phi(a[i+1], a[i+2], w_1). \tag{4}$$

We call the problem of solving the set of equations above such that the minimum number of update instructions is required the *minimum merge problem* (MMP). MMP can be formulated as a graph theoretical problem and proved to be NP-hard. The complexity analysis of MMP will not be discussed further, but the details can be found in Ottoni et al. [2001]. In Ottoni et al. [2001] we proved that the merge operation (i.e., finding the merge of minimal cost) is NP-hard through a reduction from the minimal vertex-covering problem. Moreover, we showed that for some special cases the merge operation admits an optimal $O(n)$ solution based on a dynamic programming algorithm.

8. IMPLEMENTATION OF $R \bowtie S$

As the merge operation is NP-hard, no polynomial time algorithm to find the merge of minimum cost is known. As usual, two approaches are possible in this

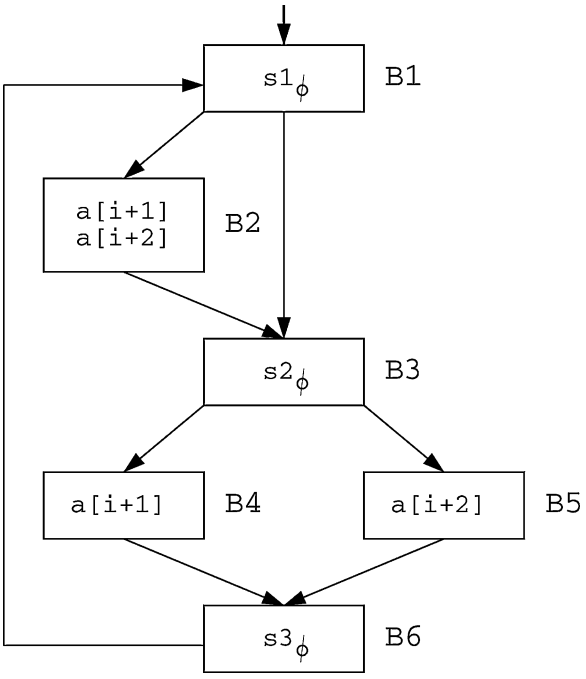

Fig. 8. The CFG from Figure 1(a) in SRF.

situation. First, if the number of array references in the ranges $R \cup S$ is small, as in many typical loops, the compiler should explore all possible combinations of solutions for the ϕ-equations to achieve the optimal solution. Second, if this is not the case, estimates for the exact values of w_k may be computed using some heuristic.

8.1 A Heuristic for $R \bowtie S$

Any candidate heuristic must choose an evaluation order for the ϕ-equations such that, at each assignment $w_k = \phi(.)$, any argument of ϕ that is a (not yet computed) virtual reference is simply ignored. For example, during the estimate of value w_3, in Equation (4), the argument w_1 is ignored and the resulting assignment becomes $w_3 = \phi(a[i+1], a[i+2])$, which results in $w_3 = a[i+1]$ (or $a[i+2]$, for the same cost). The final cost of the merge is dependent on the order that the heuristic uses to evaluate the set of ϕ-equations. In our heuristic (called *Tail-Head*), the first equation in the evaluation order is the one at the tail of the loop. From this point on, our heuristic proceeds backward from the tail of the loop up to the header, evaluating each equation as they are encountered. At each step, the rules from Section 6 are used to compute the new value of w_k, which is then used in the evaluation of future equations.

Example 8.1. Consider for example the application of this heuristic to Equations (2)–(4), from the code fragment in Figure 1(a). The result is shown

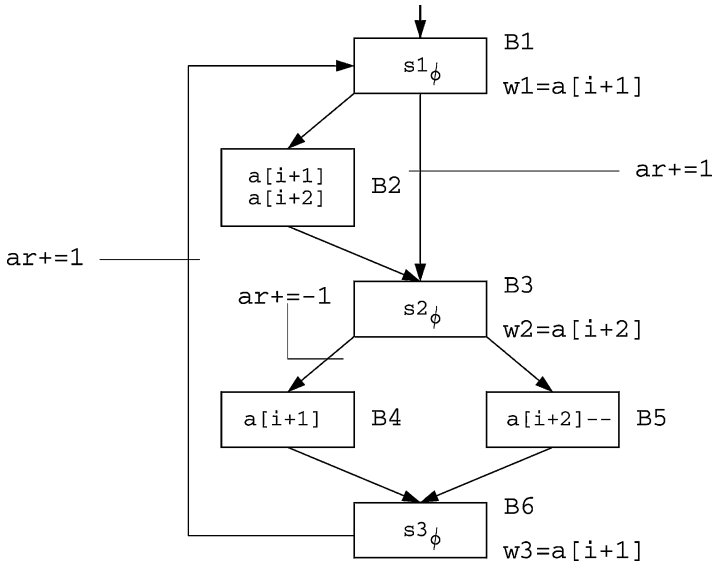

Fig. 9. Mode and update-instruction insertion for Tail-Head heuristic.

in Figure 9. First, the ϕ-equation $s3_\phi$ at the tail of the loop (block B_6) evaluates to $w_3 = a[i+1]$. This reference is equivalent to $a[i]$ in the next loop iteration. Next, $s2_\phi$ in B_3 is evaluated using case (3) of Section 6. The ϕ-function results in $w_2 = a[i+2]$. Next, $s1_\phi$, in block B_1, is evaluated to $w_1 = a[i+1]$. At this point, all virtual references (i.e., the values of w_k) have been computed. In the next step, update instructions and auto-increment/decrement modes are inserted into the code such that the references associated to the live ranges being merged satisfy the computed ϕ-functions. This results in the addition of auto-decrement mode to $a[i+2]$ in B_5, and the insertion of three update instructions: (a) $ar+=1$ on the edge (B_1, B_3); (b) $ar+=1$ on the edge (B_6, B_1); and (c) $ar+=-1$ on the edge (B_3, B_4). Notice that this solution has been achieved by a heuristic, and thus it might not result in the optimum address register allocation. In fact, the code fragment in Figure 1(b) adds a single update instruction to the code (`p += 1`) and thus is the optimum reference allocation that can be achieved for the code in Figure 1(a). The Tail-Head heuristic, on the other hand, introduces three update instructions, as shown in Figure 9. In general, when the Tail-Head heuristic is applied for the majority of the programs in our benchmark (Section 9), we noticed that only a few extra update-instructions are required,[6] resulting in a better allocation than if the previous known local techniques are used. In Ottoni et al. [2001] we described an optimal linear-time algorithm for the merge operator that results in optimal address register allocation under some special conditions, as in Figure 1(b). Preliminary results suggest that such conditions seem to be frequent in typical programs. Nevertheless, enough experimental evidence still needs to be collected to support this approach.

[6]In some cases, the optimal solution has been achieved.

```
(1)  void f(int *in, int  *out, int  n, int *bittbl)
(2)  {
(3)     int  i, j;
(4)     unsigned int  G0, G1, D0, D1, D2, D3;
(5)     unsigned long  ITEMP=0, ITEMP2;
(6)
(7)     for(j=0;j<16;j++) bittbl[j]=1<<j;
(8)
(9)     for(j=0; j<n; j++)  {
(10)        ITEMP >>=16;
(11)        ITEMP2 = ((U32)in[j]) & 0x0000ffff;
(12)        ITEMP = (ITEMP2<<5) | ITEMP;
(13)        D3 = (U16)(0x0000ffff & (ITEMP >> 2));
(14)        D2 = (U16)(0x0000ffff & (ITEMP >> 3));
(15)        D1 = (U16)(0x0000ffff & (ITEMP >> 4));
(16)        D0 = (U16)(0x0000ffff & (ITEMP >> 5));
(17)        G0 = D0 ^ D1 ^ D3;
(18)        G1 = G0 ^ D1 ^ D2;
(19)        out[2*j]   = 0;
(20)        out[2*j+1] = 0;
(21)        for (i=0; i<8; i++)  {
(22)            out[2*j] |= ((G0>>i) & 0x0001)?bittbl[2*i]:0;
(23)            out[2*j+1] |= ((G0>>(i+8)) & 0x0001)?bittbl[i]:0;
(24)        }
(25)     }
(26) }
```

Fig. 10. Code fragment from a real application.

8.2 A Real-World Example

The fragment of code in Figure 10 will be used to illustrate the application of the LRG algorithm. This code was extracted from one of the program benchmarks of Mindspeed Technologies Inc. Let's assume that the target architecture has four registers with auto-increment(decrement): `ar1`, `ar2`, `ar3`, and `ar4`.

Consider the loop at line (7) of Figure 10. Register `ar1` is allocated to reference `bittbl[j]`, and auto-increment is used to redirect `ar1` to the next iteration of this loop. The inner loop starting at line (21) requires two registers to reference `bittbl` in lines (22)–(23). Registers `ar1` and `ar2` are allocated to them. Two update instructions are inserted at the end of this inner loop to redirect `ar1` and `ar2` to the next iteration, since the two references to `bittbl` are inside conditional statements.

The outer loop from line (9) to (25) has five references using the induction variable j, at lines (11), (19), (20), (22), and (23), which would require five address registers if we allocate an `ar` for each reference. The LRG algorithm is applied to reduce the number of registers required, since only registers `ar3` and `ar4` are still available.

Table I. Live Range Growth Speed-up and Size Overhead

	Priority based		LRG optimized		Comparison (%)	
Program name	Cycles	Size	Cycles	Size	Speedup	Size
convenc	4331	4667	3943	4647	9%	0%
convolution	1220	2068	1042	2077	17%	+1%
dot_product	165	1305	160	1269	3%	−2%
biquad_N_sections	1380	2980	1218	2905	13%	−2%
fir_array	1471	2626	1263	2666	16%	+2%
fir2dim	7684	4546	6728	4566	14%	+1%
lms_array	2276	3644	1919	3665	18%	+1%
mat1x3	1202	2668	1113	2705	7%	+2%
matrix1	34657	3057	30520	3135	13%	+3%
n_complex_updates	2985	3300	2336	3410	27%	+4%
n_real_updates	1855	2716	1452	2785	27%	+3%
fft	173931	10103	165549	10097	5%	0%
autcor	179633	4003	167238	3990	7%	0%
fir8	280324	5143	256476	5088	9%	−1%
latsynth	3115	3408	3050	3402	2%	0%
fir_lms2	3454	3353	3317	3298	4%	−1%
latanal	703662	3425	691662	3411	2%	0%

In the first iteration, references at lines (19) and (20) are merged with no cost, since auto-increment can be used in both references to redirect the `ar` correctly. Then, the live range (19)–(20) is merged with (23). In this case, the auto-increment in reference at line (20) is removed, and an update instruction is needed to add 1 to the `ar` after line (24). After these two merges, three address registers are still required, and so one more iteration is needed. In the last LRG iteration, the live range (19)–(20)–(23) is merged with (22). Suppose that `ar3` is allocated to this live range. This new live range requires an update instruction `ar3 += 2` after line (24), in addition to auto-increment mode at references (19) and (22), and the auto-decrement mode at references (20) and (23). The reference at line (11) is the only one based on the array `in`, and so it is allocated to a different `ar` (`ar4`). Auto-increment mode is used to keep `ar4` correctly pointing to `in[j]` across loop iterations.

The algorithm ends with a total of three update instructions inserted.

9. EXPERIMENTAL RESULTS

Table I shows the results of applying LRG to a set of typical signal processing programs. The benchmark used in the experiments is composed by a mix of DSP-stone benchmark [Zivojnovic et al. 1994] programs and code from Mindspeed Technologies Inc. applications, running on the DSP described in Section 2. We have implemented our optimization in an optimizing DSP compiler from Mindspeed Technologies Inc., which uses an efficient priority-based register coloring allocation algorithm, combined with local auto-increment/decrement detection. This is a production-quality optimizing compiler that implements all relevant optimizations described in Aho et al. [1986], and that is used to compile real-world applications.

Two experiments have been performed. All compiler optimization flags were set (on) in both experiments. First, we ran the original compiler. The

performance of each program (in cycles) is shown in columns 2 and 3 of Table I. Cycles were measured using an *in-house* cycle accurate simulator. In the second round of experiments (columns 4 and 5), the LRG algorithm proposed in this paper was used for reference allocation, with the initial set of live ranges being composed of the single references in the loop. The experimental results show that LRG results on an average 11% speedup, when compared with the combination of the original address register coloring algorithm and local auto-increment/decrement detection. Notice that in both cases all traditional optimizations that support address register are active, for example, induction variable elimination, loop invariant removal and others. In some cases (e.g. `fir2dim`) the speedup was very good. The number of cycles saved within a basic block varied from one to four depending on the application. For the majority of the programs, we have noticed that our heuristic results in good quality allocation which usually adds only a few update instructions on the execution paths, when compared to the optimal allocation. Nevertheless, we also found cases (latanal.c) in which a manual address register allocation could considerably improve allocation.

The average size overhead due to update instructions was insignificant (0.65%), as it was the LRG execution time. Table I shows the size overhead numbers for each program after the LRG algorithm was used. Notice that for some cases (e.g., dot_product) the overhead was negative. This was possibly due to the reduction in the numbers of instructions achieved when spill/reload instructions were substituted by single update instructions.

We have also performed a third experiment, in which the LRG algorithm was again applied, but this time using for the initial set of live ranges the solution of an LRA problem for each basic block described in Araujo et al. [1996]. The resulting speedup was the same for all programs but two, and in those cases the speedup improvement was smaller than 2%. Hence, there was no substantial difference in the speedup if the initial set of live ranges was determined by first solving LRA or not. In other words, in many cases the greedy technique used in LRG is also enough to find the best allocation for basic block. On the other hand, it is possible that better results could be achieved if one of the improved solutions for LRA (Section 4), like Basu et al.'s [1999] approach, were used to compute the initial set of live ranges.

10. CONCLUSIONS

This paper proposes a technique that improves address register allocation for auto-increment (decrement) addressing modes. It uses SSA form and a simple, yet effective, algorithm, to extend previous work in the area toward considering allocation beyond basic blocks. The algorithm (LRG) is based on the growth of address register live ranges using a merge operator. Experimental results reveal an average 11% performance improvement when compared to an approach based on local auto-increment optimization and priority-based register coloring. We plan to continue our work on this problem. In Ottoni et al. [2001] we described an optimal linear-time algorithm for the merge operator that results in optimal address register allocation under some special conditions. Preliminary

results suggest that such conditions seem to be frequent in typical programs. Nevertheless, more experimental evidence still needs to be collected to support this approach.

ACKNOWLEDGMENTS

The authors thank Alan Taylor and the compiler group at Mindspeed Technologies Inc. for their strong support to this project.

REFERENCES

AHO, A. AND JOHNSON, S. 1976. Optimal code generation for expression trees. *Journal of the ACM 23*, 3 (July), 488–501.

AHO, A., JOHNSON, S., AND ULLMAN, J. 1977a. Code generation for expressions with common subexpressions. *Journal of the ACM 24*, 1 (Jan.), 146–160.

AHO, A., JOHNSON, S., AND ULLMAN, J. 1977b. Code generation for machines with multiregister operations. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*. 21–28.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers, Principles, Techniques and Tools*. Addison Wesley, Boston, MA.

ANALOG DEVICES. 1995. *ADSP-2100 Family User's Manual*. Analog Devices.

APPEL, A. AND SUPOWIT, K. 1987. Generalizations of the Sethi-Ullman algorithm for register allocation. *Software—Practice and Experience 17*, 3 (June), 417–421.

ARAUJO, G. 1997. Code generation algorithms for digital signal processors. Ph.D. thesis, Princeton University, Princeton, NJ.

ARAUJO, G., SUDARSANAM, A., AND S., M. 1996. Instruction set design and optimizations for address computation in DSP processors. In *Proceedings of the 9th International Symposium on Systems Synthesis*. IEEE Computer Society Press, Los Alamitos, CA, 31–37.

BARTLEY, D. H. 1992. Optimizing stack frame accesses for processors with restricted addressing modes. *Softw. Practice Experience 22*, 2 (Feb.), 101.

BASU, A., LEUPERS, R., AND MARWEDEL, P. 1999. Array index allocation under register constraints in DSP programs. In *Proceedings of the International Conference on VLSI Design*. IEEE Press, Los Alamitos, CA.

BODIK, R. AND GUPTA, R. 1996. Array data-flow analysis for load-store optimizations in superscalar architectures. *Int. J. Parallel Program. 24*, 6, 481–512.

BOESCH, F. AND GIMPEL, J. 1977. Covering the points of a digraph with point-disjoint paths and its application to code optimization. *Journal of the ACM 24*, 2 (April), 192–198.

BRADLEE, D., S. J., E., AND HENRY, R. 1991. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 122–131.

BRIGGS, P., COOPER, K., KENNEDY, K., AND TORCZON, L. 1989. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 275–284.

BRUNO, J. AND SETHI, R. 1975. The generation of optimal code for stack machines. *Journal of the ACM 22*, 3 (July), 382–396.

BRUNO, J. AND SETHI, R. 1976. Code generation for one-register machine. *Journal of the ACM 23*, 3 (July), 502–510.

CALLAHAN, D., CARR, S., AND KENNEDY, K. 1990. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*. ACM Press, New York, NY, 53–65.

CALLAHAN, D. AND KOBLENZ, B. 1991. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 192–202.

CHAITIN, G. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN'82 Symposium on Compiler Construction*. ACM Press, New York, NY, 98–105.

CHOW, F. AND HENNESSY, J. L. 1990. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst. 12*, 4 (Oct.), 501–536.

CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, F. 1989. An efficient method of computing static single assignment form. In *Proceedings of the ACM POPL'89*. ACM Press, New York, NY, 23–25.

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

GEBOTYS, C. 1997. DSP address optimization using a minimum cost circulation technique. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE Computer Society Press, Los Alamitos, CA, 100–103.

GOODMAN, J. AND HSU, A. 1988. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 Conference on Supercomputing*. 442–452.

GUPTA, R., SOFFA, M., AND OMBRES, D. 1994. Efficient register allocation via coloring using clique separators. *ACM Trans. Program. Lang. Syst. 16*, 3 (May), 370–386.

HITCHCOCK III, C.Y. 1986. Addressing modes for fast and optimal code generation. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA.

HORWITZ, L., KARP, R., MILLER, R., AND WINOGRAD, S. 1966. Index register allocation. *Journal of the ACM 13*, 1 (Jan.), 43–61.

LEUPERS, R., BASU, A., AND MARWEDEL, P. 1998. Optimized array index computation in DSP programs. In *Proceedings of the ASP-DAC*. IEEE Computer Society Press, Los Alamitos, CA.

LEUPERS, R. AND DAVID, F. 1998. A uniform optimization technique for offset assignment problems. In *Proceedings of the ACM SIGDA 11th International Symposium on System Synthesis*. ACM Press, New York, NY, 3–8.

LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. 1995. Storage assignment to decrease code size. In *Proceedings of 1995 ACM Conference on Programming Language Design and Implementation*. ACM Press, New York, NY.

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

OTTONI, G., RIGO, S., ARAUJO, G., RAJAGOPALAN, S., AND MALIK, S. 2001. Optimal Live Range Merge for Address Register Allocation in Embedded Programs. In *Proceedings of the 10th International Conference on Compiler Construction* (CC2001). Lecture Notes in Computer Science, vol. 2027. Springer, Berlin, Germany, 274–288.

RAO, A. AND PANDE, S. 1999. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 128–138.

SETHI, R. 1975. Complete register allocation problems. *SIAM J. Comput. 4*, 3 (Sept.), 226–248.

SETHI, R. AND ULLMAN, J. 1970. The generation of optimal code for arithmetic expressions. *Journal of the ACM 17*, 4 (Oct.), 715–728.

TARJAN, R. E. 1983. *Data Structures and Network Algorithms*. SIAM, Philadelphia, PA.

ZIVOJNOVIC, V., VELARDE, J., AND SCLÅAGER, C. 1994. DSPstone, a DSP benchmarking methodology. Technical report, Aachen University of Technology (Aug.), Aachen, Germany.

Received October 2000; accepted February 2002