Optimal Live Range Merge for Address Register Allocation in Embedded Programs

Guilherme Ottoni[1], Sandro Rigo[1], Guido Araujo[1], Subramanian Rajagopalan[2], and Sharad Malik[2]

[1] University of Campinas, Campinas SP 6176, Brazil
[2] Princeton University, Princeton NJ 08544, USA

Abstract. The increasing demand for wireless devices running mobile applications has renewed the interest on the research of high performance low power processors that can be programmed using very compact code. One way to achieve this goal is to design specialized processors with short instruction formats and shallow pipelines. Given that it enables such architectural features, indirect addressing is the most used addressing mode in embedded programs. This paper analyzes the problem of allocating address registers to array references in loops using auto-increment addressing mode. It leverages on previous work, which is based on a heuristic that merges address register live ranges. We prove, for the first time, that the merge operation is NP-hard in general, and show the existence of an optimal linear-time algorithm, based on dynamic programming, for a special case of the problem.

1 Introduction

Complex embedded programs running on compact wireless devices has become a typical computer system nowadays. Although this trend is the result of a mature computing technology, it brings a new set of challenges. It has become increasingly hard to meet the stringent design requirements of such systems: low power consumption, high performance and small code size. Due to this design constraints, many embedded programs are written in assembly, and run on specialized embedded processors like *Digital Signal Processors* (DSPs) (e.g. ADSP 21000 [2]) or commercial CISC machines (e.g. Motorola 68000 [15]).

The increase in the size of embedded applications has put a lot of pressure towards the development of optimizing compilers for these architectures. One way to achieve this goal is to design specialized processors with short instruction formats and shallow pipelines. Since it allows such architectural improvements, indirect addressing is the most common addressing mode found in embedded programs. Again, due to cost constraints, the number of address registers available in the processor is fairly small, specially in DSPs (typically ≤ 8). This paper analyzes the problem of allocating address registers to array references in embedded program loops.

The register allocation problem has been extensively studied in the compiler literature [5,4,6,12] for general-purpose processors, but not enough work has been done for the case of highly constrained embedded processors. *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 [3,11,13]. These are efficient graph-based solutions, when references are restricted to basic block boundaries. In particular, Leupers et al [13] is a very efficient solution to LRA. Unfortunately, not much work has been developed towards finding solutions 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 GRA are described in Sect. 3. In [7] we proposed a solution to GRA, based on a technique that we call *Live Range Growth* (LRG). In Sect. 3 of this paper we give a short description of the LRG algorithm. In the current work, we extend our study of GRA in two ways. First, we perform the complexity analysis of the merge operation required by LRG, and prove it to be NP-hard (Sect. 4). Moreover, we prove the existence of an optimal linear-time algorithm for a special case of merge (Sect. 5). In Sect. 6 we evaluate the performance of this algorithm, and show that it can save update instructions. Section 7 summarizes the main results.

2 Problem Definition

Global Reference Allocation (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 such 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 results in efficient code, when traditional optimizations like induction variable elimination and loop invariant removal [1,16] 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 Fig. 1(a). If one register is assigned to each reference, $a[i+1]$ and $a[i+2]$, 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 Fig. 1(a) using a single pointer `p`, as shown in Fig. 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 on one of the loop control paths, which is considerably smaller than the cost of spilling/reloading one register. The C code fragment of Fig. 1(b) is a source level model of the *intermediate representation* code resulting after we apply the optimization described in the next sections.

3 Basic Concepts

This section contains a short overview of some basic concepts that are required to formulate the GRA problem and to study its solutions. For a more detailed description, the reader should report to [7].

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

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

First of all assume, for the rest of this paper, that the array data type is a memory word (a typical characteristic of embedded programs). Moreover, assume that each array reference is atomic (i.e. encapsulated into a specific data type in the compiler intermediate representation), and array subscript expressions are not considered for *Common Sub-expression Elimination*(CSE).

A central issue in GRA is to bound allocation to the number of address registers in the target processor. As a consequence, 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*, which is basically the distance between two index expressions of two array references.

The concept of live range used in this paper is a straightforward extension of idea of *variable liveness* adopted in the compiler literature [1]. The set of array references of a program can be divided into sets of control-flow paths, each of them defining a live range. For example, in Fig. 2(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 Fig. 2, symbol ”++” (”--”) following a reference assigns a post-increment (decrement) mode to that reference. Notice that a live range can include references across distinct basic blocks.

As mentioned before, not much research work has been done towards finding solutions to the allocation of address registers for a whole procedure. A solution to this problem has been proposed recently in [7], that is based on repeatedly

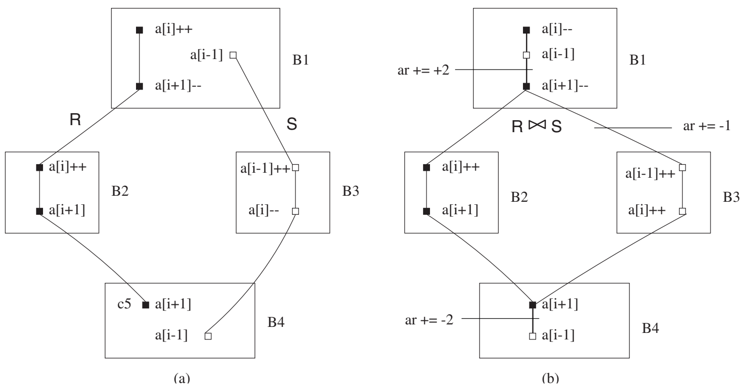

Fig. 2. (a) Two live ranges R and S; (b) A single live range is formed after merging $R \bowtie S$.

merging 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 described in [7] starts by partitioning all references across a set of initial live ranges. The initial set of ranges($\mathcal{R}$) is formed by the individual references in the loop[1]. Ranges are merged pairwise until its number is smaller or equal to the number of address registers available in the processor. This technique is called *Live Range Growth* (LRG). Merge operations for similar problems have also been studied in [14, 9]. 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. For example, after ranges R and S in Fig. 2(a) are merged, a single range $R \bowtie S$ results (Fig. 2(b)) which requires three update instructions. At each step of the algorithm the pair of ranges of minimum cost is merged.

3.1 The Merge Operator ($R \bowtie S$)

In order to enable references to share the same address register, the LRG algorithm needs to enforce, at compile time, that each reference b should be reached by only one reference a, since this allows the computation of the indexing distance $d(a, b)$ at compile time. As a consequence, the compiler can decide, at compile time, between using auto-increment (decrement) mode for a, or inserting an update instruction on the path from a to b. This requirement can be satisfied if the references in the CFG are in *Single Reference Form* (SRF) [7], a variation of *Static Single Assignment (SSA) Form* [8].

After a program is in SRF, any loop basic block for which array references converge will have a ϕ-equation, including the header and the tail of the loop. After reference analysis, which is performed by solving *reaching definitions* [1,16] with the data items being the array references, any ϕ-equation has associated to it sets UD and DU, with elements $a_i \in UD$ and $b_j \in DU$. The value w that

[1] Each array access is assigned to a unique range.

results from the ϕ-equation depends on how distant the elements in UD and DU 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 by substituting exhaustively w for all the elements in $UD \cup DU$, and measuring the cost due to the update instructions.

The problem of merging two live ranges can now be formulated as follows. Assume that, during some intermediate step in the LRG algorithm, two ranges R and S are considered for merge. Consider, for the following analysis, only those references which are in $R \cup S$. The set of ϕ-equations associated to these references forms a system of equations. The unknown variables in this system are virtual references w_k. These equations may have circular dependencies, caused basically by the following reasons: (a) it originates 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 Fig. 4(a) corresponding to the loop in Fig. 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:

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

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

$$s3_\phi : w_3 = \phi(a[i+1], a[i+2], w_1) \tag{3}$$

The work in [7] assumed that the optimal merge of any two ranges is a NP-complete problem. For those instances of the problem in which the exact solution is too expensive, a heuristic should be applied. 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 virtual reference is simply ignored. For example, during the estimate of value w_3, in Equation 3, 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 merge is dependent on the order that the heuristic uses to evaluate the set of ϕ-equations. In the heuristic proposed in [7] (called *Tail-Head*), the first equation in the evaluation order is the one at the tail of the loop. From this point on, the heuristic proceeds backward from the tail of the loop up to the header evaluating each equation as it is encountered, and computing the new value of w_k, which is then used in the evaluation of future equations.

Example 1. Consider the application of this heuristic to the Equations 1-3, from the code fragment in Fig. 1(a). The result is shown in Fig. 3(a). 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, $s2_\phi$ and $s1_\phi$ are evaluated, resulting in $w_2 = a[i+2]$ and $w_1 = a[i+1]$, respectively. At this point, all virtual references (i.e. the values of w_k) have been computed. In the last 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

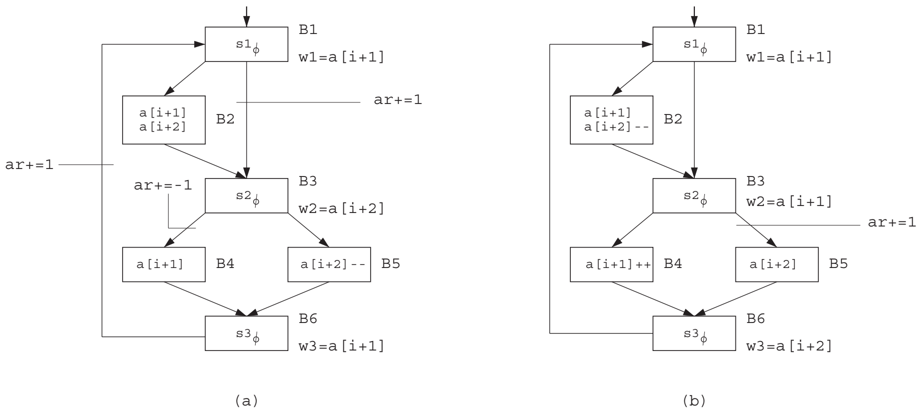

Fig. 3. (a) Mode and update-instruction insertion for Tail-Head heuristic; (b) Optimal mode and update-instruction insertion using dynamic programming.

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).

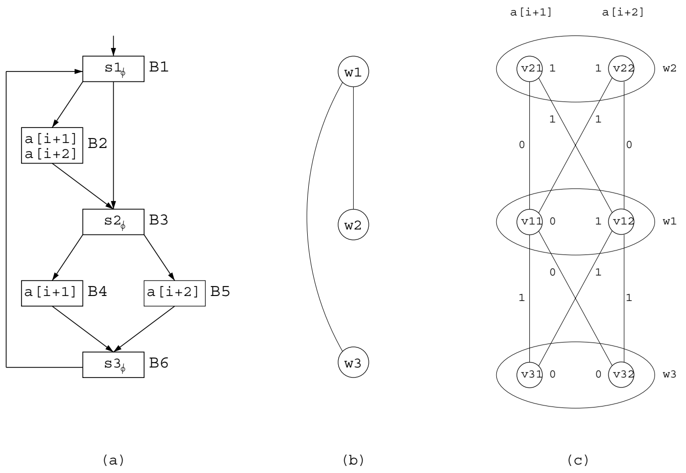

Fig. 4. (a) The CFG in SRF; (b) The DG$_\phi$ representation; (c)The SG$_\phi$ for the DG$_\phi$.

4 Complexity Analysis of $R \bowtie S$

Given that the merge operation is central to the LRG algorithm, in the following sections we perform the analysis of this operation. We complete the work started in [7] by proving that the merge operation (i.e., finding the minimal cost for it) is an NP-hard problem in general, through a reduction from the minimal vertex-covering problem. Moreover, we show that in a special case the merge operation admits an optimal solution, based on a dynamic programming algorithm. For the other cases, we show that a simple heuristic leads to effective solutions.

4.1 The Φ-Dependence Graph

The problem of solving the set of equations above can be formulated as a graph theoretical one as follows. Consider the code fragment of Fig. 4(a). We build a dependence graph where vertices are associated to the ϕ-equations and edges define dependencies between equations. This graph is called *ϕ-Dependence Graph* (DG$_\phi$) and is constructed as follows:

Definition 1. *The DG$_\phi$ is an undirected graph for which there exist a vertex associated to each ϕ-equation, and there exist an edge (w_i, w_j), between two vertices w_i and w_j, if and only if w_i is a ϕ-function of w_j and vice-versa.*

Figure 4(b) shows the DG$_\phi$ corresponding to the equations in Fig. 4(a).

4.2 The Φ-Solution Graph

Having constructed the DG$_\phi$ for a set of ϕ-equations, we build a graph which reflects all possible solutions for these equations. But before doing so, we need to define some concepts. Let m be the number of distinct references that can result from the ϕ-equations. The range of references can be obtained by a simple code inspection, searching for the minimum and maximum values for each reference in the loop[2], which are stored into vector refs$[i], 1 \leq i \leq m$. Using these concepts we can now construct a graph that encodes all the possible solutions for the ϕ-equations.

Definition 2. *The Φ-Solution Graph (SG$_\phi$) is an undirected graph constructed from the DG$_\phi$ as follows: (1) For each vertex u in DG$_\phi$ insert m vertices into SG$_\phi$, one for each possible solution of the equation associated to u; (2) If there is an edge in DG$_\phi$ between two vertices u and v, and being sets $\{u_1, u_2, \ldots, u_m\}$ and $\{v_1, v_2, \ldots, v_m\}$ the vertices in the SG$_\phi$ associated with u and v respectively, insert edges (u_i, v_j) into SG$_\phi$, for all $1 \leq i, j \leq m$.*

Figure 4(c) shows the SG$_\phi$ obtained from the DG$_\phi$ in Fig. 4(b). Notice that SG$_\phi$ is a multipartite graph, i.e. SG$_\phi = \{P_1, P_2, \ldots, P_n\}$, with $P_i = \{v_{ij}, \mid 1 \leq j \leq m\}$. SG$_\phi$ is a weighted graph, with costs both on its vertices and edges. We

[2] It may be necessary to consider the loop increment.

assign a local cost $lcost(v_{ij})$ to each vertex v_{ij}, where $1 \leq i \leq n$ and $1 \leq j \leq m$, and define it as follows:

$$lcost(v_{ij}) = \sum_{j=1}^{m} cost(\text{refs}[j], a) \ , \tag{4}$$

for each real reference a in $args[w_i]$, where $args[w_i]$ is the set of all arguments of the ϕ-function associated with w_i. The local cost is computed using the same cost function defined in [7] (which considers the possibility of using the auto-increment(decrement) modes), by assuming that $w_i = \text{refs}[j]$ and considering only the arguments of ϕ_i which are real references (i.e. not w). In other words, *lcost* gives the contribution to the total merge cost when the solution for the ϕ-equation associated to w_i is $w_i = \text{refs[j]}$. We also assign to each edge $e = (v_{ij}, v_{kl})$ in SG_ϕ, a global cost given by

$$gcost(v_{ij}, v_{kl}) = cost(\text{refs}[j], \text{refs}[l]) \ . \tag{5}$$

Notice that both arguments of *gcost* are virtual references, and reflect the cost of assigning solutions refs$[j]$ and refs$[l]$ respectively to virtual references w_i and w_k. Figure 4(c) illustrates the local and global costs associated with each SG_ϕ vertex and edge.

Now we are able to state the solution of the ϕ-equation system in terms of the SG_ϕ. We want to determine the set $V = \{v_i | v_i \in P_i, 1 \leq i \leq n\}$ such that the total merge cost $cost_{\bowtie}[V] = cost_{\bowtie}(R, S)$ is minimum, where

$$cost_{\bowtie}[V] = \sum lcost(v_i) + \sum gcost(e) \ , \tag{6}$$

for all $1 \leq i \leq n$ and $e \in \text{SG}_\phi[V]$, where $\text{SG}_\phi[V]$ denotes the sub-graph of SG_ϕ induced by the vertices of V.

4.3 $R \bowtie S$ is NP-hard

In this subsection, we prove that the problem stated above is NP-hard. To do this, we re-state it as a decision problem, and call it the *Minimum Cost Induced Sub-graph Problem (MCISP).*

Definition 3 (MCISP). *Let G be an instance of a SG_ϕ and k a positive integer. Find a subset V of the vertices of G (as described in Sect. 4.2) such that $cost_{\bowtie}[V] \leq k$.*

We show that MCISP is NP-hard through a reduction from the *Vertices Covering Problem* (VCP) [10], defined as follows.

Definition 4 (VCP). *Given a graph G and an integer k, decide whether G has a vertex-cover C with at most k vertices, such that each edge of G has at least one of its incident vertices in C.*

Theorem 1. *MCISP is NP-complete.*

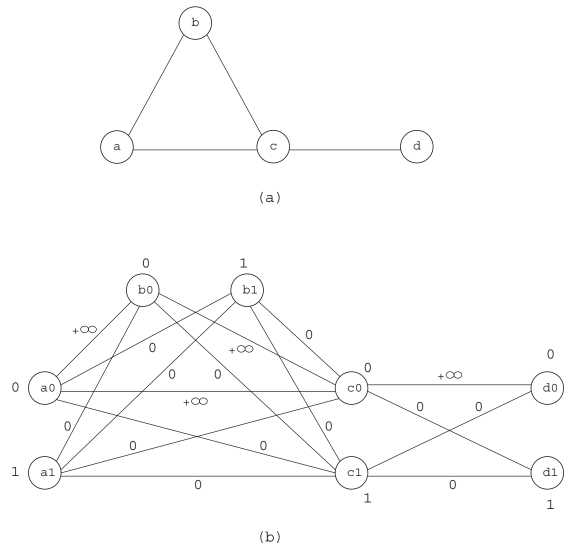

Fig. 5. (a) VCP on original instance graph G; (b) MCISP on reduced instance graph G'.

Proof. First, it must be noticed that MCISP $\in$ NP. This is an easy task, as we can verify in polynomial-time if a given solution V has a total cost less than or equal to k. We will not give more details on this.

Now, for MCISP to be NP-complete, it is missing to show that it is NP-hard. To do this, we use a reduction from a known NP-complete problem to MCISP. We choose the vertex-cover problem (VCP) for this purpose.

Consider now the reduction of VCP, with instance graph G and integer k, to MCISP, with G' and k'. First we show how to create G' from G. For example, consider the graph G in Fig. 5(a). For each vertex $v \in G$, we insert two vertices v_0 and v_1 in G'. Then, for each edge $e = (u, v) \in G$, we insert four edges in G': (u_0, v_0), (u_0, v_1), (u_1, v_0) and (u_1, v_1).

Next, we add the costs to G', both to its vertices and edges, to reflect the local and global costs defined above. For each v_0 we make $lcost(v_0) = 0$, and for each vertex v_1 we set $lcost(v_1) = 1$.

On the edges, the following costs are assigned: $gcost(u_0, v_0) = +\infty$, and $gcost(u_0, v_1) = gcost(u_1, v_0) = gcost(u_1, v_1) = 0$. Fig. 5(b) illustrates the graph G' obtained from G in Fig. 5(a). It is clear that this reduction can be done in polynomial-time. Now we are going to prove that, having obtained G' from G as described above, and setting $k = k'$, VCP on G and k is satisfied if and only if MCISP on G' and k' is satisfied. Let's first see why the forward argument holds. Let C be a solution to VCP. So, for each edge in G, at least one of its incident vertices is in C. We argue that $V = V_1 \cup V_0$, where $V_1 = \{v_1 | v \in C\}$ and $V_0 = \{v_0 | v \notin C\}$ is a solution to MCISP. Let $q \leq k$ be the cost of C (i.e., the number of vertices). Consider now the cost of the solution V to MCISP. The

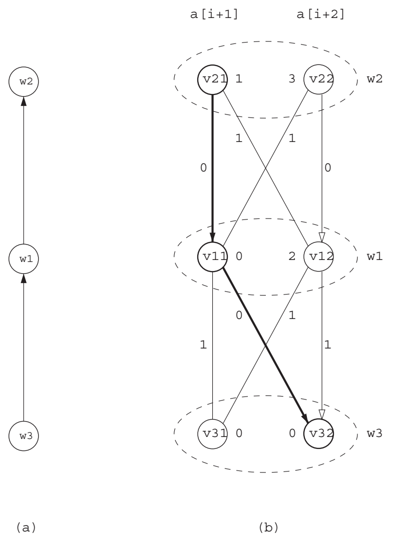

Fig. 6. (a) The DDG$_\phi$ for the DG$_\phi$ of Fig. 4; (b) The DSGϕ for the DDG$_\phi$ in (a).

first summation on Equation 6 is exactly the number of vertices in V_1, which is also the number of vertices in C (i.e. q). Now consider the second summation in Equation 6. As C is a solution to VCP, there are no two vertices $u_0, v_0 \in V$ such that the edge $(u_0, v_0) \in G'[V]$. Hence, every edge in $G'[V]$ has its *gcost* equal to 0, and $cost_{\bowtie}[V]$ is exactly q.

We still have to prove the backward argument, i.e., that given a solution V to MCISP with cost $q \leq k'$, there is a solution C to VCP with the same cost q. Being k' a finite integer number, $G'[V]$ has no edge between any two vertices u_0 and v_0. Therefore, choosing $C = \{v | v_1 \in V\}$, any edge in G will have at least one of its incident vertices in C, and so C is a vertex-cover for G.

Now it is just missing to show that the cost of solution C to VCP is also q. This is easy to see, as the cost q is exactly the number of vertices in V with subscript 1, which is also the number of vertices in C. And so, the correctness of the polynomial-time reduction is proved.

Having proved the decision version of the problem to be NP-complete, and being the problem an optimization one, we conclude that the problem of finding the minimal cost for the merge operation is NP-hard.

5 The Case when DG$_\phi$ Is a Tree

As shown in the previous section, the merge operation is NP-hard. In this section we present an optimal exact algorithm for merging two live ranges, under the assumption that the corresponding DG$_\phi$ is a tree. The algorithm is based on the dynamic programming technique, and uses a special order to traverse the nodes of the DG$_\phi$. Ordering the traversal of the vertices actually means imposing a direction on each edge of the dependence graph. The arc $(u \rightarrow v)$ obtained by choosing a direction for the edge (u, v) implies that w_u is an argument of w_v, unlike in the previous form of the DG$_\phi$, when edge (u, v) indicated that w_v was also an argument of w_u. In this directed form of the DG$_\phi$ (called *DDG*$_\phi$) the other arguments of the ϕ-functions are not changed.

Our algorithm is based on a particular traversal of the DDG$_\phi$ that we call *Leaves Removal Order* (LRO). This order is obtained by successively removing any leaf of the tree. If l is a leaf, it has a unique adjacent node, say v, such that the direction of (l, v) is set to $l \rightarrow v$. When l is removed, w_v is dropped out of the list of arguments of the ϕ-function for w_l.

This ordering algorithm can be implemented with time-complexity $O(n)$[3], for example by taking into account the degree of each vertex.

Figure 6(a) shows the DDG$_\phi$ obtained from the DG$_\phi$ of Fig. 4(a), using the algorithm just described above. We call the attention to the fact that, although we described this ordering algorithm separately, in practice it can be implemented on the same pass as the next step.

The next step in our solution is a dynamic programming algorithm which computes the minimum merge cost for any solution of our problem. We call this algorithm the *LRO Algorithm*, and show its pseudo-code in Alg. 1. We need a vector $accost[1, \ldots, m]$ for each vertex of DDG$_\phi$, to hold the cumulative cost at this vertex, for each one of the m possible solutions of the corresponding ϕ-equation, from refs[1] to refs[m], respectively. In other words, for each vertex v of the DG$_\phi$ we assign an element $accost[i]$ to each one of its corresponding v_i vertices in SG$_\phi$. The *accost* vectors are initialized with the respective *lcost* vectors. Then, following the LRO, for each leaf node l being removed, such that $l \rightarrow v$ is its last arc, we add the cost c_{v_i} to $accost_v[i]$, where c_{v_i} is the minimum value between $accost_l[j] + gcost(v_i, l_j)$, for all $1 \leq j \leq m$.

Algorithm 1 LRO algorithm

```
(1) procedure LRO($DG_\phi$, $SG_\phi$)
(2) for each vertex $v_{ij} \in SG_\phi$, where $1 \leq i \leq n$ and $1 \leq j \leq m$, do
(3)       $accost_i$[j] $\leftarrow$ $lcost(v_{ij})$
(4) for all but the last vertex $v_p \in DG_\phi$, according to LRO, do
(5)       let ($v_p$,$v_q$) be the last edge incident to $v_p$
(6)       for j $\leftarrow$ 1 to $m$ do
(7)            $min \leftarrow +\infty$
(8)            for i $\leftarrow$ 1 to $m$ do
(9)                 if $accost_p$[i] + $gcost(v_{pi}, v_{qj}) < min$ then
(10)                     $min \leftarrow accost_p$[i] + $gcost(v_{pi}, v_{qj})$
(11)           $accost_q$[j] $\leftarrow$ $accost_q$[j] + $min$
```

[3] Remember that n is the number of vertices of DG$_\phi$

The minimum total merge cost is the minimum value in the DG_ϕ last vertex's *accost* vector. For simplicity, we omitted from Alg. 1 the code needed to recover the solution (i.e. the set $S = \{v_i | v_i \in P_i, 1 \leq i \leq n\}$), though it should be clear that it can be implemented without any extra computational complexity cost. The time-complexity of this algorithm is $O(n.m^2)$. In practice, since m is generally bounded by a small constant[4], the algorithm complexity reduces to $O(n)$.

Lemma 1. *Given a LRO sequence* $S = (v_1, v_2, ..., v_n)$ *of* DG_ϕ*'s vertices, the LRO Algorithm properly computes the minimal value for* $accost[v_{ij}]$*, for every* $1 \leq i \leq n$ *and* $1 \leq j \leq m$*.*

Proof. First, it must be noticed that, as long as DG_ϕ is a tree, there always exists an LRO. Moreover, the ϕ-functions are inserted such that the cost of every arc in the CFG affected by the ϕ-functions' values is considered exactly once. Let $S_k = (v_1, ..., v_k)$. We prove this lemma by induction in k.

Basis: $k = 1$. In this case, $accost[v_{1j}]$, $1 \leq j \leq m$, is just equal to $lcost(v_{1j})$, which is minimal, as all the *lcost*'s are chosen to be locally optimal.

Induction Hypothesis: We assume that, for every $i < k$, the LRO Algorithm computes the minimal values for $accost[v_{ij}]$.

Inductive Step: We define the set $A = \{v_i | i < k \text{ and } (v_i, v_k) \in \mathrm{DG}_\phi\}$. We now need to show how to compute the minimal value for each $accost[v_{kj}]$, $1 \leq j \leq m$. The $lcost(v_{kj})$ is always in $accost[v_{kj}]$. So we have to minimize

$$\sum_{v_i \in A} min\{accost[v_{iy}] + gcost(v_{kj}, v_{iy}) | 1 \leq y \leq m\}$$

for each $1 \leq j \leq m$. $gcost(v_{kj}, v_{iy})$ is a precalculated constant. By the induction hypothesis, $accost[v_{iy}]$ is minimal, for each $v_i \in A$ and $1 \leq y \leq m$. So, it turns out that $accost[v_{kj}]$ is correctly computed, for each $1 \leq j \leq m$, as we minimize a value among the minimal values for all possible choices. And so the proof is complete, and the LRO Algorithm computes the minimal values for $accost[v_{kj}]$, for all valid values for k and j.

Theorem 2. *The LRO Algorithm is optimal when the* DG_ϕ *is a tree, in the sense that it results in the smallest possible cost for* $R \bowtie S$*.*

Proof. The Lemma 1 states that the LRO Algorithm computes the minimum cost for $accost[v_{ij}]$, for all vertices $v_{ij} \in \mathrm{SG}_\phi$, if the corresponding DG_ϕ is a tree. So, for the algorithm to compute the smallest cost for $R \bowtie S$, we just have take the minimum between $\{accost[v_{nj}] | v_{nj} \in \mathrm{SG}_\phi$, and the v_n is the last vertex in the LRO for $\mathrm{DG}_\phi\}$.

[4] Remember that m is the number of references (inside the loop) in the range defined by the one with the minimum and the one with the maximum index value.

Example 2. Figure 6(b) shows the result of running this algorithm on the SG_ϕ from Fig. 4(c), according to the LRO from Fig. 6(a). In this figure, we show the global costs on the edges, and the final cumulative cost on each vertex. At each vertex v, we use arrows to point to the other vertices u (a single vertex in this example), such that u is used to reach the minimum value at v. The vertices and edges that compose the solution are highlighted in Fig. 6(b).

Figure 3(b) shows the code fragment of Fig. 1(a) with the modifications associated to the solution illustrated in Fig. 6(b). Notice that only one update instruction is necessary, which is in accordance with cost $accost[v_{21}]$ in Fig. 6(b). Figure 3(a) shows the solution found using the Tail-Head heuristic described in [7], for the instance of the problem shown in Fig. 1(a). This solution has a higher cost (3 update instructions).

6 Experimental Results

Our experimental framework is based on programs from the MediaBench benchmark [18], which are typically found in many embedded applications. We have implemented the LRG heuristic [7], and the LRO algorithm described in Sect. 5 into the IMPACT compiler [17]. Both algorithms run at `lcode` level, just before all standard intermediate representation optimizations [1]. At each merge in the LRG algorithm, the DG_ϕ graph is tested for the presence of cycles. If DG_ϕ is cyclic the Tail-Head (TH) heuristic is applied. On the other hand, if DG_ϕ is acyclic, the LRO algorithm is used to compute the ϕ-functions.

Table 1. Comparison between the Tail-Head (TH) and LRO+TH approaches.

Program Name	Loop	Minimum No. ARs	% Trees	# Update Instr. TH	# Update Instr. LRO+TH
jpeg	1	2	100	4	3
	2	1	64	4	3
	3	2	46	4	4
epic	1	1	78	0	0
	2	1	100	3	3
	3	2	77	2	1
pgp	1	2	100	0	0
	2	1	100	1	1
mpeg	1	1	75	3	3
mesa	1	1	75	3	3
	2	1	100	2	2
pegwit	1	1	67	2	2
	2	2	100	1	0
gsm	1	2	100	0	0
	2	1	89	1	1

For each program, we measured the number of update instructions required by its inner loops [5], such that one address register is used to access all array references of each array (Table 1). This is a worst case scenario which reveals the improvement resulting from the LRO algorithm. This table also shows an estimate of the number of times when DG_ϕ is a tree during the live range growth process. These preliminary results indicate that this may be a frequent situation in embedded applications. We acknowledge that measuring the final cycle count would be more realistic, but unfortunately at this point we do not have a complete retarget of the IMPACT architecture to a real-world DSP processor. On the other hand, DSP compilers have few address registers available and some of them are used to perform other tasks (e.g. stack access), such that only very few are left for address register allocation. In this case, the above scenario approaches what occurs in many real situations. Notice from Table 1, that the improvement due to the LRO algorithm is typically one less update instruction in the loop body, what considerably improves loop performance, particularly for tight inner loops found in many critical signal processing applications.

Even more relevant, these numbers also reveal that the TH heuristic that we proposed in [7] results in near optimal allocation.

7 Conclusions and Future Work

This paper addresses the problem of allocating address registers to array references in loops of embedded programs. It is based on a previous work that assigns address registers to live ranges by merging ranges together. The contributions of this work are two. First, it proves that the optimal merge of address register live ranges is an NP-hard problem. Second, it proves the existence of an optimal linear-time algorithm to merge live ranges when the dependency graph formed by the set of ϕ-equations on the references is a tree.

A full evaluation of this technique on a real-world DSP running a wide set of benchmarks is under way. We are also working to add support to new DSP features. One example is the support to modify registers. This kind of register is present in many DSPs and we intend to use them to increment address registers by values greater than one, incurring in no additional cost. By doing so, we can avoid issuing some update instructions and, as a consequence, decrease the update instruction overhead.

Acknowledgments. This work was partially supported by CNPq (300156/97-9), ProTem CNPq/NSF Collaborative Research Project (68.0059/99-7) and by fellowship grants CNPq (141710/2000-4) and (141958/2000-6). We also thank the reviewers for their comments.

[5] Only those loops which make access to arrays have been considered.

References

1. Aho, A., Sethi, R., and Ullman, J. *Compilers, Principles, Techniques and Tools.* Addison Wesley, Boston, 1986.
2. Analog Devices. *ADSP-2100 Family User's Manual.*
3. Araujo, G., Sudarsanam, A., and Malik, S. Instruction set design and optimizations for address computation in DSP processors. In *9th International Symposium on Systems Synthesis* (November 1996), IEEE, pp. 31–37.
4. Briggs, P., Cooper, K., Kennedy, K., and Torczon, L. Coloring heuristics for register allocation. In *Proc. of the ACM SIGPLAN'89 on Conference on Programming Language Design and Implementation* (June 1982), pp. 98–105.
5. Chaitin, G. Register allocation and spilling via graph coloring. In *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction* (June 1982), pp. 98–105.
6. Chow, F., and Hennessy, J. L. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst. 12*, 4 (October 1990), 501–536.
7. Cintra, M., and Araujo, G. Array reference allocation using SSA-Form and live range growth. In *Proceedings of the ACM SIGPLAN LCTES 2000* (June 2000), pp. 26–33.
8. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, F. An efficient method of computing static single assignment form. In *Proc. of the ACM POPL'89* (1989), pp. 23–25.
9. Eckstein, E., and Krall, A. Minimizing cost of local variables access for DSP-processors. In *LCTES'99 Workshop on Languages, Compilers and Tools for Embedded Systems* (Atlanta, July 1999), Y. A. Liu and R. Wilhelm, Eds., vol. 34(7) of *SIGPLAN*, ACM, pp. 20–27.
10. Garey, M., and Johnson, D. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.
11. Gebotys, C. DSP address optimization using a minimum cost circulation technique. In *Proceedings of the International Conference on Computer-Aided Design* (November 1997), IEEE, pp. 100–103.
12. Gupta, R., Soffa, M., and Ombres, D. Efficient register allocation via coloring using clique separators. *ACM Trans. Programming Language and Systems 16*, 3 (May 1994), 370–386.
13. Leupers, R., Basu, A., and Marwedel, P. Optimized array index computation in DSP programs. In *Proceedings of the ASP-DAC* (February 1998), IEEE.
14. Liao, S., Devadas, S., Keutzer, K., Tjiang, S., and Wang, A. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems 18*, 3 (1996), 235–253.
15. Motorola. *DSP56000/DSP56001 Digital Signal Processor User's Manual*, 1990.
16. Muchnick, S. S. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.
17. Pohua P. Chang Scott A. Mahlke William Y. Chen, N. J. W., and mei W. Hwu, W. Impact: An architectural framework for multiple-instruction-issue processors. 266–275.
18. Potkonjak, C. L. M., and Mangione-Smith, W. H. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems.