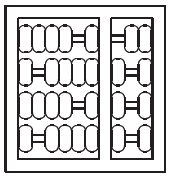

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Algorithms for Array Reference Allocation in Loops of Embedded Programs

Guilherme Ottoni *Guido Araujo*

Technical Report - IC-02-05 - Relatório Técnico

May - 2002 - Maio

Algorithms for Array Reference Allocation in Loops of Embedded Programs

Guilherme Ottoni* Guido Araujo†

28 May 2002

Abstract

Efficient address register allocation has been shown to be a central problem in code generation for processors with restricted addressing modes. This paper extends previous work on *Global Array Reference Allocation* (GARA), the problem of allocating address registers to array references in loops. It describes two heuristics to the problem, based on the SSA Form, presenting experimental data to support them. In addition, it proposes an approach to solve GARA optimally which, albeit computationally exponential, is useful to measure the efficiency of other methods. Experimental results, using the MediaBench benchmark, reveal that the proposed heuristics can solve the majority of the benchmark loops near optimality in polynomial-time. A substantial execution time speedup is reported for the benchmark programs, after compiled with the original and the optimized versions of GCC.

1 Introduction

The increase in the size and complexity of embedded system applications has induced designers to adopt architectures that offer low power consumption, enhanced performance and reduced cost. Processors that run embedded programs range from commercial CISC machines (e.g. Motorola 68000) to specialized *Digital Signal Processors* (DSPs) (e.g. DSP16xx [24]), 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 1 out of every 6 instructions for a typical general-purpose program [19]. In order to speedup address computation, most embedded processors offer specialized *addressing modes*. A typical example is the auto-increment (decrement) mode, which enables the encoding of very short instructions. All commercial DSPs and most CISC processors *Instruction Set Architectures* (ISAs) have auto-increment (decrement) modes. In fact, in order to reduce the instruction size, many embedded processors do not allow the typical base-register plus offset addressing mode

*Research supported by CAPES.

†Research partially supported by CNPq (Proc. 300156/97-9 and 68.0059/99-7) and FAPESP (Proc. 2000/15083-9).

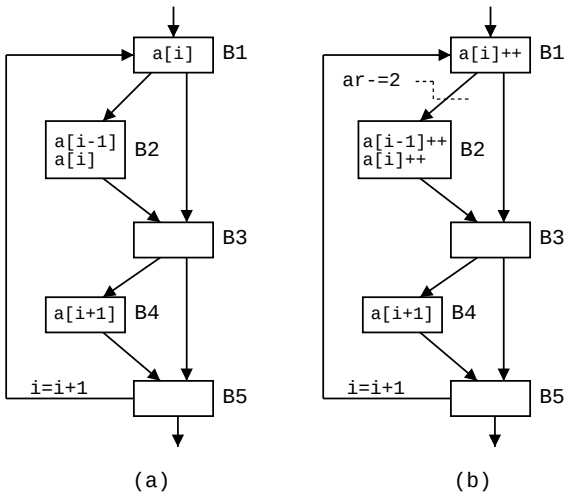

Figure 1: (a) CFG fragment; (b) Inserting auto-increment mode and update instructions.

frequently found in general-purpose architectures. Even worse, very few registers are available in these processors (typically 4-16), and addressing is usually performed only through specialized address register.

This paper extends previous work [13, 26] on *Global Array Reference Allocation* (GARA), which is the problem of allocating address registers to array references in loops running on embedded processors. As an example, consider the *Control-Flow Graph* (CFG) from Figure 1(a), where only the array references are shown. Solving GARA on this code, with a single address register available, produces the code in Figure 1(b). In Figure 1(b), all the array references are performed through the single address register (`ar`), and symbol `++` (`--`) following a reference implies that post-increment (post-decrement) addressing mode is used.

This paper describes two heuristics to GARA, presenting experimental data to support them. In addition, it proposes an approach to solve GARA optimally which, albeit computationally exponential, is useful to measure the efficiency of the other methods. The experimental results are very encouraging. Using the exact solution as the baseline, the experiments reveal that program loops in the MediaBench benchmark [27] can be solved near optimality in polynomial-time through the heuristics. An average speedup of 14.3% is reported for the benchmark programs after compiled with the original and optimized versions of GCC.

This paper is divided as follows. Section 2 lists the previous work on GARA. Section 3 describes, for the first time, the *Extended Single Reference Form* (ESRF), which is required to guarantee the optimality of the dynamic programming algorithm proposed in [26]. Two heuristics for GARA are summarized in Section 4, and Section 5 proposes a method to compute its exact solution. Finally, Section 6 reports the experimental results when our implementation in GCC compiles MediaBench programs.

2 Previous Work

Register allocation is a well studied problem in compilers. Many of the first problems in code generation involved finding good algorithms for register allocation [30, 2, 29]. Global register allocation is an important problem in code generation which has been extensively studied [11, 8, 12, 18]. Other researchers have considered the interaction of register allocation and scheduling in code generation for RISC machines [17, 7], and inter-procedural register allocation [10]. The allocation of local variables to the stack-frame, using auto-increment (decrement) mode, has been studied in [22, 21, 5, 23, 28, 15].

Local Array Reference Allocation (LARA) 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. LARA has been studied in [20, 4, 16], which are efficient graph-based solutions, when references are restricted to basic block boundaries. Global register allocation for array references, on general-purpose architectures, has been studied before by Bodik and Gupta [6] and Callahan et al [9]. In [6] and [9] array references are allocated to general-purpose registers. As the loop iteration progresses, references are moved among registers in a *pipelined* fashion. Unfortunately, many embedded processors are highly constrained architectures containing very few specialized registers, what makes the application of these techniques impossible.

In [13], a technique based on live range growth and a variation of *Static Single Assignment* (SSA) Form [14] was proposed. It consists on consecutively merging pairs of live ranges until the number of ranges equals the number of address registers in the target processor architecture. A heuristic was used to decide which pair of live ranges should be merged. The problem of finding the minimal number of update instructions when merging pairs of live ranges has been proved to be NP-complete in general [26]. The difficulty of the problem lies on choosing the best (minimum cardinality) set of update instructions among a combinatorial number of possible sets. The large number of sets results from the need to keep correct, on every possible execution path, the value of the address register for the array references on the merged live range. In [26], Ottoni et al. proves the existence of an optimal dynamic programming algorithm to find the minimal set of update instructions, for a special case of live range topology. Preliminary experimental results in [26] speculated that this particular topology would be very common in practice, although not enough benchmark data was presented to support that. The experimental results from Section 6 confirm this hypothesis to be true for the MediaBench programs.

3 The Extended Single Reference Form (ESRF)

Any approach that aims at solving the GARA optimization problem should be able to perform two central tasks. First, it has to choose the points inside the code where update instructions would be needed to adjust the address registers. Second, it has to allocate an address register to each array reference such that the cost of the required update instructions is minimized. In order to achieve an optimal GARA solution, both tasks have to be performed optimally. In this section, we show how to solve the first part of the problem

optimally, i.e. deciding the points where update instructions could be required. Notice that the need of an update instruction, at some selected point, will depend on how efficient is the algorithm that assigns address registers to the array references, as discussed in Section 4.

In [13] it was realized that the problem of deciding where update instructions are needed resembles the problem of choosing places to insert ϕ-functions in the SSA-Form [14]. The points where to insert ϕ-functions are those on the *Iterated Dominance Frontier* [14] of the basic blocks that contain array references. The program representation resulting after ϕ-functions are inserted was called *Single Reference Form* (SRF). For example, Figure 2(a) shows the CFG of a loop body in SRF. However, SRF is not enough to guarantee the optimality of the approach presented in [26].

The problem with SRF is that it identifies the points where more than one array reference reach, but not those points that reach multiple array references (what is the case of the point at the exit of B1 in Figure 2(a)). As a result, in SRF it is possible to have update instructions that are not associated with any ϕ-function, although their values depend on the choice of addressing modes. In order to fix this problem, we propose what we call the *Extended Single Reference Form* (ESRF). In this form, in addition to the ϕ-functions inserted at the *beginning* of the basic blocks that form the Iterated Dominance Frontier (ϕ_b), we also insert ϕ-functions at the exit of the basic blocks that are on the *Iterated Post-dominance Frontier* [25] (ϕ_e). But it is still possible that the insertion of ϕ-functions into one of the dominance frontiers will require the insertion of ϕ-functions into the other one. In order to deal with this, another iteration level is used to compute ESRF such that, at the end, each array reference has only one reference in each of its DU/UD-chains.

Algorithm 1 Combined Dominance Frontier

```
(1) function CDF(Ref_BBs :  Set_of_BBs)
(2)     var IDF, IPF : Set_of_BBs;
(3)     S ← Ref_BBs;
(4)     do
(5)         S' ← S;
(6)         IDF ← Iterated_Dominance_Frontier(S);
(7)         S ← S ∪ IDF;
(8)         IPF ← Iterated_Postdominance_Frontier(S);
(9)         S ← S ∪ IPF;
(10)    while S ≠ S';
(11)    return (IDF, IPF);
```

We call this approach *Combined Dominance Frontier* (CDF) and describe it in Algorithm 1. To illustrate how Algorithm 1 works, consider Figure 2(b), ignoring the ϕ-functions shown. Table 1 shows the value of the sets S, IDF and IPF as they are computed in Algorithm 1. For example, in step 3, S is set to $\{1, 4, 5, 6\}$, and so the iterated post-dominance frontier in step 4 is calculated as if there where array references in all of these basic blocks.

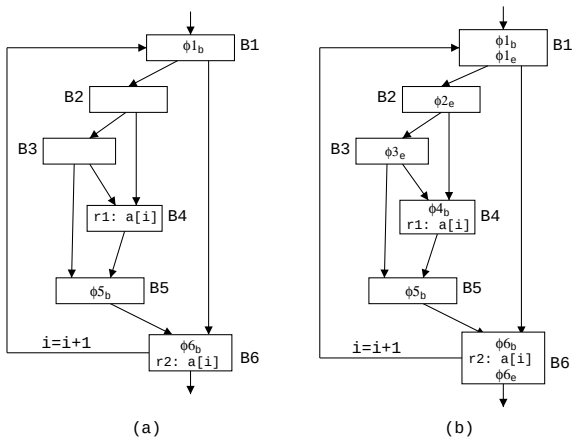

Figure 2: (a) CFG in SRF; (b) CFG in ESRF.

Step	CDF Line	S	IDF	IPF
1	3	4,6		
2	6		1,5,6	
3	7	1,4,5,6		
4	8			1,2,3,6
5	9	1,2,3,4,5,6		
6	6		1,4,5,6	
7	7	1,2,3,4,5,6		
8	8			1,2,3,6
9	9	1,2,3,4,5,6		

Table 1: CDF computation for the code from Figure 2(b).

At the end of the algorithm, IDF=$\{1, 4, 5, 6\}$ and IPF=$\{1, 2, 3, 6\}$. The resulting code in ESRF is illustrated in Figure 2(b), with the corresponding ϕ-functions inserted.

ESRF holds the important property that any update instruction depends on a ϕ-function[1]. Thus, update instructions are inserted between a ϕ-function and an array reference or another ϕ-function, but never between two array references. As the variables of our optimization problem are the values to be chosen for the ϕ-functions, this is a fundamental property that guarantees the optimality of the algorithm described in [26].

[1]Update instructions may not depend on a ϕ-function, for example for consecutive array references inside a single basic block. But these cases can be locally solved in an optimal fashion.

4 GARA Heuristics

In this section we summarize the methods proposed in [13] and [26]. Both techniques are based on merging live ranges. Initially, each array reference from the loop being optimized is assigned to a separate live range, and then a sequence of live range merge operations is performed, until the number of live ranges reaches the number of address registers available on the target processor. In both approaches, the choice for the pair of live ranges to be merged, at each step, is performed by computing the cost of all live ranges obtained by pairwisely merging the current live ranges, and then choosing the least costly one. The two approaches differ on the way the cost for a live range is computed: [13] uses a heuristic called *Tail-Head* (TH), while [26] uses an optimal dynamic programming algorithm called *Leaves Removal Order* (LRO) whenever the topology of the live range allows, resorting to TH otherwise. We call this combined approach LRO-TH.

In order to illustrate both methods, we use a loop from the `pegwit` program in MediaBench. Figure 3 shows the CFG representation of this loop and its corresponding array references. The loop has nine references (`r1` to `r9`), associated to two arrays (`a` and `b`), and the loop step is 2. The edges are labeled with the corresponding estimated execution frequencies. When an update instruction is needed, we use the estimated execution frequency on the edge or basic block where it will be inserted as its corresponding cost. This way, our goal becomes to minimize the total estimated execution frequency for the required update instructions, instead of simply minimizing the number of update instructions as in [13, 26].

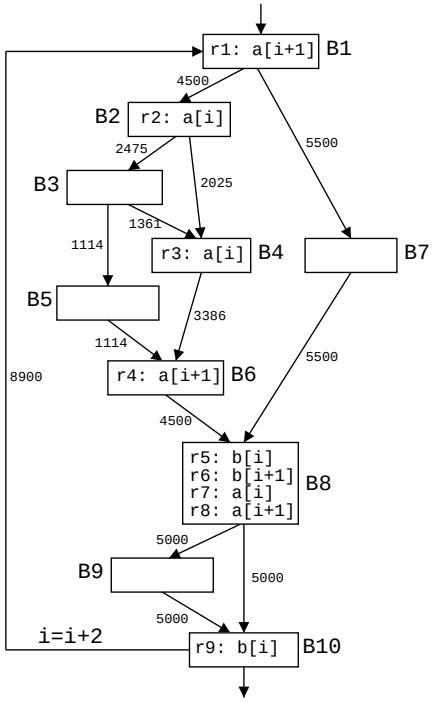

Figure 3: The control-flow graph for a loop example from `pegwit`.

4.1 The Tail-Head Heuristic

In this section we shortly describe how the Tail-Head (TH) heuristic is used to estimate the cost of a merge operation during GARA. For further details the reader should report to [13].

When GARA starts, the live range growth approach takes place, merging at each step the pair of current live ranges that leads to the best total cost. Hence, at each merge operation the cost of the new live range must be determined. When the TH heuristic is used to compute the cost, the following operations take place. Initially, the loop is transformed to SRF, in order to determine the points where ϕ-functions are required. Then, the ϕ-functions are solved, starting at the loop tail toward the loop head. For each ϕ-function, the solution is chosen among the values of all references in its UD/DU-chains, ignoring the ϕ-functions which have not been solved yet. The cost of the merged live range is given by the summation of the expected execution frequency of the update instructions required to set the address registers correctly. Zero cost auto-modify addressing modes are used whenever possible.

As an example, Figure 4 shows the live range formed by merging references `r3`, `r7` and `r8`. The UD/DU-chains for the ϕ-functions associate to these references are also shown. First, ϕ_{8b} is solved, resulting in `a[i]` (the only value in its UD/DU-chains). Then, ϕ_{6b} results in `a[i]`, for the same reason. Finally, ϕ_{1b} is solved, and in this case two solutions are possible: `a[i]`, because of `r3` and because it is the solution for ϕ_{6b} and ϕ_{8b}, and `a[i-1]`, because of the reference `r8` from the previous loop iteration (note that `a[i+1]` from the previous iteration is `a[i-1]` in the current one, as the loop step is 2). `a[i]` is chosen because it leads to cost zero, with the insertion of a post-increment addressing mode at reference `r8`.

Example: GARA Using the TH Heuristic

We now illustrate the GARA solution using the TH heuristic. Consider the code from Figure 3. Table 2 shows the initial set of live ranges. For each live range, this table presents the basic blocks where ϕ-functions are required when transforming it to SRF (column 2), and the live range cost (column 3) calculated using the Tail-Head heuristic.

Ref.	LR ϕ-funcs.	LR cost
1	1b	10000
2	1b,8b	8900
3	1b,6b,8b	13386
4	1b,8b	8900
5	1b	8900
6	1b	8900
7	1b	8900
8	1b	8900
9	1b	8900
Total cost		85686

Table 2: Initial live ranges; costs computed using the Tail-Head heuristic.

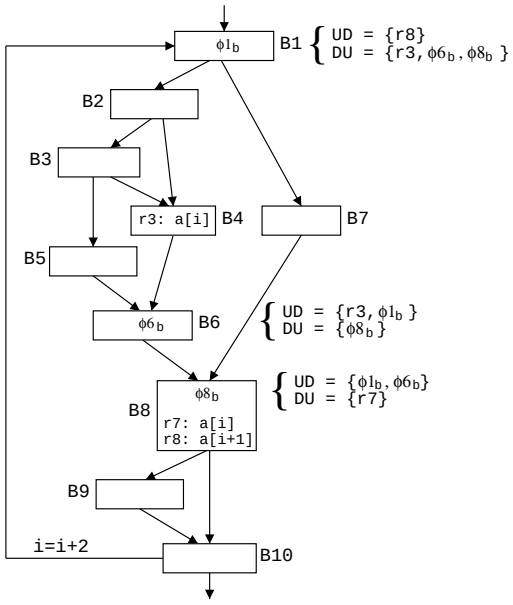

Figure 4: The live range formed by references `r3`, `r7` and `r8` in SRF.

Step	LRs	LR ϕ-funcs.	LR cost	Total cost
0	[1][2][3][4][5][6][7][8][9]	–	–	85686
1	[1][2][3][4]**[5,6]**[7][8][9]	1b	0	67886
2	[1][2][3][4][5,6]**[7,8]**[9]	1b	0	50086
3	[1][2]**[3,7,8]**[4][5,6][9]	1b,6b,8b	0	36700
4	**[1,2]**[3,7,8][4][5,6][9]	1b,8b	8900	26700
5	[1,2]**[3,4,7,8]**[5,6][9]	1b,6b,8b	1114	18914
6	[1,2][3,4,7,8]**[5,6,9]**	1b	8900	18914
7	**[1,2,3,4,7,8]**[5,6,9]	1b,6b,8b	12286	21186

Table 3: Live Range Growth using the Tail-Head heuristic.

As GARA progresses, the live range growth approach takes place, starting with the ranges in Table 2. Table 3 shows the sequence of merge operations that are performed. In column 2, the resulting set of live ranges after each merge is illustrated, with the just merged range highlighted. The third column contains the basic blocks where ϕ-functions are required for the just formed range, and column 4 shows the cost of the merge operation, computed using the Tail-Head heuristic. The last column lists the total cost of all current live ranges at this step of the execution.

The merging of live ranges is performed until two ranges remain (step 7 in Table 3). These ranges cannot be merged, as they refer to different arrays (`a` and `b`), and thus cannot share the same address register. Assuming that 3 address registers are available (what is

the case of the target processor we used), we have two possibilities: either using 2 address registers (one for each final live range), or using 3 address registers (one for each live range after step 6). We choose the last alternative, as it leads to a smaller total cost (18914). The final allocation and the update instructions inserted are shown in Table 4.

Address Register	References	Update Instructions		
		Instr.	Edge	Cost
ar_0	r_1:a[i+1]-- r_2:a[i]	ar_0+=3	B10→B1	8900
ar_1	r_3:a[i]++ r_4:a[i+1]-- r_7:a[i]++ r_8:a[i+1]++	ar_1+=1	B3→B5	1114
ar_2	r_5:b[i]++ r_6:b[i+1]-- r_9:b[i]	ar_2+=2	B10→B1	8900

Table 4: The final allocation using TH.

4.2 The Leaves Removal Order Algorithm

The LRO approach for computing a live range cost, in opposition to the Tail-Head heuristic, guarantees that the optimal solution is found for the values of the ϕ-functions, although it does not apply to every code in ESRF. Fortunately, the experimental results in Section 6 show that the cases to which this method applies are indeed very common in practice.

Ref.	LR ϕ-functions	DG_ϕ acyclic	LR cost
1	1b,10e	yes	8900
2	1b,1e,8b,10e	no	8900
3	1b,1e,2e,3e,4b,6b,8b,10e	no	13386
4	1b,1e,8b,10e	no	8900
5	1b,10e	yes	8900
6	1b,10e	yes	8900
7	1b,10e	yes	8900
8	1b,10e	yes	8900
9	1b,10e	yes	8900
Total cost			85686

Table 5: Initial live ranges; costs computed using the LRO algorithm whenever possible, and the Tail-Head heuristic otherwise.

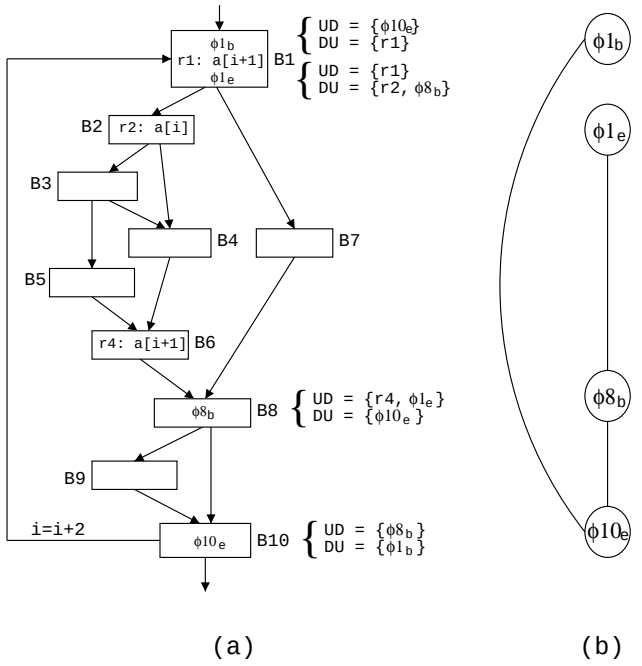

Figure 5: (a) The live range formed by references `r1`, `r2` and `r4` in ESRF. (b) The corresponding DG_ϕ.

In [26] we introduced the concept of *ϕ-Dependence Graph* (DG_ϕ). This is an undirected graph in which there is one vertex for each ϕ-function, and an edge between two vertices if and only if the solution to one of the ϕ-functions depends on the solution to the other. The DG_ϕ can be constructed using an algorithm similar to reaching definitions and DU/UD-chains [3] on the ϕ-functions. In order to illustrate the concepts of ESRF and DG_ϕ, Figure 5(a) presents, for the same example from Figure 3, the live range formed by references `r1`, `r2` and `r4` in ESRF. The corresponding DG_ϕ is shown in Figure 5(b). For example, there is an edge between ϕ_{1e} and ϕ_{8b} because $\phi_{8b} \in DU_{1e}$ (and so $\phi_{1e} \in UD_{8b}$).

Whenever DG_ϕ is a *tree*, a dynamic programming algorithm can be used to solve the ϕ-functions optimally [26]. To achieve that, we must first find all possible solutions for any ϕ-function in ESRF. These values are exactly the array references that appear in the loop, and the references in the next loop iteration. In practice, the number of possible solutions for the ϕ-functions is usually restricted to a few values. The dynamic programming algorithm executes in a bottom-up fashion, following a *Leaves Removal Order* (LRO) of the DG_ϕ. For each tree leaf l, all the possible pairs of solutions for l and its single adjacent vertex v are tested, and the best costs are accumulated in v. Before, the costs related to other vertices adjacent to l have already been accumulated in l. This algorithm has a running time linear on the number of ϕ-function. More details on this algorithm are described in [26].

Example: GARA Using the LRO-TH Approach

We now illustrate the GARA solution using the LRO-TH approach. Consider again the code from Figure 3. Table 5 shows the initial live ranges for the live range growth approach.

Step	LRs	LR ϕ-functions	DG_ϕ acyclic	LR cost	Total cost
0	[1][2][3][4][5][6][7][8][9]	–	–	–	84586
1	[1][2][3][4]**[5,6]**[7][8][9]	1b,10e	yes	0	66786
2	[1][2][3][4][5,6]**[7,8]**[9]	1b,10e	yes	0	48986
3	[1][2]**[3,7,8]**[4][5,6][9]	1b,1e,2e,3e,4b,6b,8b,10e	no	0	35600
4	**[1,2]**[3,7,8][4][5,6][9]	1b,1e,8b,10e	yes	8900	26700
5	**[1,2,4]**[3,7,8][5,6][9]	1b,1e,8b,10e	yes	8900	17800
6	[1,2,4][3,7,8]**[5,6,9]**	1b,10e	yes	8900	17800
7	**[1,2,3,4,7,8]**[5,6,9]	1b,1e,2e,3e,4b,6b,8b,10e	no	12286	21186

Table 6: Live Range Growth using the LRO algorithm whenever possible, and the TH heuristic otherwise.

Column 3 shows whether the corresponding DG_ϕ is a tree or not. Whenever DG_ϕ is a tree, the LRO algorithm is used to compute the optimal cost of the range resulting after merge. Otherwise, the Tail-Head heuristic cost estimate is determined.

Table 6 shows the sequence of live range mergings, similarly as in Table 3. Here again, two options for allocation are possible: using either 2 or 3 address registers. The best choice is again to allocate 3 address registers to the live ranges in step 6 of Table 6. This gives a total cost of 17800, which is slightly better than the cost achieved with the Tail-Head-only approach (18914). The final allocation is presented in Table 7.

Address Register	References	Update Instructions		
		Instr.	Edge	Cost
ar_0	r_1:a[i+1]-- r_2:a[i]++ r_4:a[i+1]--	ar_0+=3	B10→B1	8900
ar_1	r_3:a[i] r_7:a[i]++ r_8:a[i+1]++	–	–	–
ar_2	r_5:b[i]++ r_6:b[i+1]-- r_9:b[i]	ar_2+=2	B10→B1	8900

Table 7: The final allocation using LRO-TH.

5 The GARA Exact Solution

In this section, we present a method (EXACT) for computing the exact, minimum cost GARA solution for a given loop. This approach relies on the LRO algorithm whenever

# ARs	Partition			
	`A (Array a)`		`B (Array b)`	
	Cost	LRs	Cost	LRs
0	140000	–	80000	–
1	10014	[1,2,3,4,7,8]	8900	[5,6,9]
2	8900	[1,2,4][3,7,8]	8900	[5,6][9]
3	14400	[1,3][2,4][7,8]	26700	[5][6][9]

Table 8: The C matrix holding the best solution for each entry in the *# ARs* $\times$ *Partition* space. Here each partition corresponds to one of the arrays from Figure 3.

possible. As the experimental results from Section 6 show, LRO is applicable in the great majority of the cases, and this is what makes the EXACT approach feasible[2].

Let R be the number of address registers in the target processor which are available for allocation, and A be the number of array references in a loop. First of all, we should identify all the A array references inside the loop, and partition them such that two references are put into the same partition if and only if their *indexing distance* [31] can be statically determined. We call K the number of partitions inside the loop, and $P_1, ..., P_K$ the partitions themselves.

It is clear that only references in the same partition are eligible for sharing an address register, although references in the same partition can be allocated to different address registers. Therefore, one of the decisions that must be made is how to divide the R address registers among the K partitions. The second decision EXACT has to make is how to sub-partition each partition P_j into live ranges. Finally, for each live range, we should choose the best solution for the ϕ-functions in a way to minimize the update instruction cost. Algorithm 2 describes a top-level pseudo-code for our approach.

Procedure `EXACT` (Algorithm 2) is the entry point for the pseudo-code. Its first step is to identify and partition the array references inside the loop. Then, for each partition P_j, it calls the procedure `Compute_Minimum_Costs`, which fills in the j^{th} column of the matrix C (C_{ij} is the minimum possible cost if i address registers are assigned to partition P_j). In order to fill in this column, `Compute_Minimum_Costs` exhaustively generates all the possibilities of sub-partitioning the array references in P_j in a number of live ranges that varies from 1 to R. For each live range, the corresponding minimum update instruction cost is computed using the LRO algorithm if possible, or a brute force, exponential algorithm otherwise, which simply tests all the combinations of solutions to the ϕ-functions. In addition, an estimate of the cost if no address register is assigned to this partition is made. This estimate is dependent on the target processor, and considers any other addressing mode available, or the cost of spilling an address register. Table 8 shows the C matrix computed for the loop from Figure 3.

Having the C table computed, the `EXACT` procedure calls the `Optimal_AR_Distribution` procedure, which is responsible for choosing the best way to divide the R address register

[2]By *feasible* here we mean that it requires a computational time that we can deal with for the purposes of this research, although it may not be practical to be performed inside a compiler.

Algorithm 2 GARA Exact Solution

```
(1) procedure EXACT (L :  loop)
(2)     identify the array references in L, partitioning them
(3)       in P_1, ..., P_K;
(4)     for each P_j, 1 ≤ j ≤ K do
(5)       Compute_Minimum_Costs(P_j);
(6)     Optimal_AR_Distribution({P_1, ..., P_K}, C);
(7)
(8) procedure Compute_Minimum_Costs(P_j)
(9)     fill in C_0j with the cost estimated if no address
(10)      register is allocated to P_j;
(11)    C_ij ← +∞, 1 ≤ i ≤ R;
(12)    for each combination of partitioning the references
(13)        in P_j in live ranges {LR_1, ..., LR_i}, 1 ≤ i ≤ R do
(14)      total_cost ← 0;
(15)      for each LR_k , 1 ≤ k ≤ i , do
(16)        build DG_φ for LR_k;
(17)        if DG_φ is a tree then
(18)          cost ← LRO_cost(LR_k);
(19)        else
(20)          cost ← Brute_Force_cost(LR_k);
(21)        total_cost ← total_cost + cost;
(22)      if total_cost < C_ij then
(23)        C_ij ← total_cost;
(24)
(25) procedure Optimal_AR_Distribution({P_1, ..., P_K},C)
(26)    min_cost ← +∞;
(27)    for each combination of values r_1, r_2, ..., r_K |
(28)        Σ_{i=1}^{K} r_i ≤ R and r_i ≥ 0 do
(29)      cost ← 0;
(30)      for k ← 1 to K do
(31)        cost ← cost + C_{r_k,k};
(32)      if cost < min_cost then
(33)        min_cost ← cost;
(34)
(35) procedure Brute_Force_cost(LR_k)
(36) /* Backtracking to generate all the combinations of
(37)    solutions for the φ-functions in the ESRF of LR_k.
(38)    Return the minimum cost among the costs for all
(39)    of these combinations. */
```

among the K partitions. This is another brute force algorithm, which explores all the possibilities for making this distribution. This procedure uses the precomputed values previously stored in the C table, in order to avoid recomputing the minimum costs at each time. For our example, using the C matrix from Table 8, the best possible solution is to attribute 2 ARs to partition `A` and 1 to partition `B`, resulting in a total cost of 17800. Note that this is the same solution found by the LRO-TH approach in Section 4.2.

6 Experimental Results

In order to test the methods described in this paper, we have implemented all approaches inside GCC version 3.0.2 [1]. The GARA optimization takes place right after the traditional loop optimizations, making use of the loop induction variable information available at this point. Moreover, we used the GCC infrastructure for profiling-driven optimizations to improve the update instruction cost estimation (as described in Section 4).

The target processor for the experiments was the Lucent DSP16xx [24], which has a total of 4 address registers (one of which is used as stack-pointer), and post-increment (decrement) addressing modes. The results presented here are based on static profiling information. We measured the expected execution cycles for inner-most loops from MediaBench [27] applications. Only loops with any array reference have been considered. Three set of experiments have been performed. In the first set (Table 9), the speedup between all approaches and the original GCC was measured. The second set of experiments (Table 10) aimed at comparing the compilation time between the heuristics and the exact solution. The last set of experiments (Table 11) computed the percentage of DG_ϕ graphs in the loops which are trees.

Table 9 shows a comparison between the following approa-ches: (a) live range growth using the Tail-Head heuristic (TH); (b) live range growth using the combination of the Leaves Removal Order algorithm and the Tail-Head heuristic (LRO-TH); and (c) the exact solution (EXACT). The speedup was measured with respect to the original GCC implementation [1]. Table 9 shows that the speedup achieved by LRO-TH approaches the speedup of the time-consuming EXACT method (average difference of 0.09%). In addition, the TH approach also leads to a speedup close to the exact solution (average difference of 0.54%), although not as good as LRO-TH does.

Table 10 compares the execution time performance of the GCC implementations of TH, LRO-TH and EXACT. It shows that the TH and LRO-TH heuristics do not increase the compilation time noticeably (in fact, they reduced the average compilation time slightly). On the other hand, the EXACT method demands a great amount of time for loops with many array references, as in some MediaBench programs (e.g. `pegwit`). This is due to the intrinsic exponential time-complexity of EXACT. Notice that for some programs (e.g. `mpeg2`), GARA improved the compilation time, what should be due to the simplifications it performs in the code, accelerating other optimizations.

Table 11 presents data regarding the topology of the DG_ϕ's, during the application of both the LRO-TH and the EXACT techniques. The results show that the great majority of the DG_ϕ's are trees, meaning that the linear-time optimal LRO algorithm for computing

Program	# of loops	Speedup (%)		
		TH	LRO-TH	EXACT
`adpcm`	2	0.80	0.80	1.01
`epic`	6	10.24	11.24	11.50
`g721`	1	0.00	0.00	0.00
`ghostscript`	37	13.46	13.95	13.96
`jpeg`	32	13.86	14.42	14.44
`mpeg2`	7	13.13	13.13	13.93
`pegwit`	5	25.22	25.22	25.22
`pgp`	3	23.96	23.96	23.96
Average	–	13.85	14.30	14.39

Table 9: Comparison in terms of speedup between the original GCC, the Tail-Head (TH) approach, the Leaves Removal Order and Tail-Head (LRO-TH) combined approach, and the EXACT solution.

Program	Compilation Time (s)			
	Baseline GCC	TH	LRO-TH	EXACT
`adpcm`	0.800	0.800	0.790	0.800
`epic`	4.430	4.530	4.530	4.880
`g721`	0.790	0.750	0.760	0.760
`ghostscript`	41.400	40.980	41.830	273.470
`jpeg`	23.410	22.240	23.380	259.310
`mpeg2`	12.020	10.840	10.280	2.440
`pegwit`	6.160	4.000	4.540	38933.750
`pgp`	4.690	4.330	4.320	4.620
Average	11.713	11.059	11.304	4935.004

Table 10: Comparison in terms of compilation time between the original GCC, the Tail-Head (TH) approach, the Leaves Removal Order and Tail-Head (LRO-TH) combined approach, and the EXACT solution.

the cost of live ranges is frequently executed in LRO-TH and EXACT. In LRO-TH, the execution of LRO reduces the cost of the update instructions by diminishing the number of times that the Tail-Head heuristic is evoked. In EXACT, the optimal cost for the live ranges can almost always be computed in linear time by LRO, thus enabling EXACT to run in feasible time.

Finally, it is worth noting that, even though most of the DG_ϕ's happen to be trees, the results obtained by TH approaches that of LRO-TH, meaning that even the simple Tail-Head heuristic leads to a good solution to the GARA problem.

Program	LRO-TH			EXACT		
	trees	total	% trees	trees	total	% trees
adpcm	2	4	50.00	2	4	50.00
epic	33	45	73.33	59	113	52.21
g721	1	1	100.00	1	1	100.00
ghostscript	2259	2400	94.12	602599	604353	99.71
jpeg	2190	2329	94.03	583676	585426	99.70
mpeg2	21	23	91.30	24	26	92.31
pegwit	165	199	82.91	1583239	1584209	99.94
pgp	7	7	100.00	10	10	100.00
Total	4678	5008	93.41	2769610	2774142	99.84

Table 11: Proportion of DG_ϕ's that are trees when applying the LRO-TH and the EXACT approaches.

7 Conclusions

In this paper we extended previous work on GARA. We presented the Extended Single Reference Form, which is needed for the optimality of the LRO algorithm [26]. We proposed an exact, optimal algorithm for GARA, which uses the LRO algorithm. The detailed experimental results show that the LRO-TH approach generally achieves solutions close to optimal. The average speedup of LRO-TH for the loops of the MediaBench benchmark was 14.3%, when comparing to the GCC's original address register allocation technique. In addition, we showed that the great majority of DG_ϕ are trees, making it possible to the exact technique to run fast for most of the loops, despite its exponential time-complexity.

8 Acknowledgments

We would like to thank Gang-Ryung Uh from Agere Systems Inc. for his support on the DSP16xx processor, and Michael Collison from Mindspeed Inc. for creating and maintaining the DSP16xx GCC port.

References

[1] The GNU Compiler Collection Project. http://gcc.gnu.org.

[2] A. Aho and S. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, Boston, 1986.

[4] G. Araujo, A. Sudarsanam, and M. S. Instruction set design and optimizations for address computation in DSP processors. In 9*th International Symposium on Systems Synthesis*, pages 31–37. IEEE, November 1996.

[5] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software Practice and Experience*, 22(2):101, February 1992.

[6] R. Bodik and R. Gupta. Array data-flow analysis for load-store optimizations in superscalar architectures. *International Journal of Parallel Programming*, 24(6):481–512, 1996.

[7] D. Bradlee, S. Eggers, and R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.

[8] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. of the ACM SIGPLAN'89 on Conference on Programming Language Design and Implementation*, pages 275–284, July 1989.

[9] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 53–65, June 1990.

[10] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proc. of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 192–203, June 1991.

[11] G. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction*, pages 98–105, June 1982.

[12] F. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, October 1990.

[13] M. Cintra and G. Araujo. Array reference allocation using SSA-Form and live range growth. In *Proceedings of the ACM SIGPLAN LCTES 2000*, pages 26–33, June 2000.

[14] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. In *Proc. of the ACM POPL'89*, pages 23–25, 1989.

[15] E. Eckstein and A. Krall. 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*, pages 20–27, May 1999.

[16] C. Gebotys. DSP address optimization using a minimum cost circulation technique. In *Proceedings of the International Conference on Computer-Aided Design*, pages 100–103. IEEE, November 1997.

[17] J. Goodman and A. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 Conference on Supercomputing*, pages 442–452, July 1988.

[18] R. Gupta, M. Soffa, and D. Ombres. Efficient register allocation via coloring using clique separators. *ACM Trans. Programming Language and Systems*, 16(3):370–386, May 1994.

[19] Hitchcock III, C.Y. *Addressing Modes for Fast and Optimal Code Generation*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, Dec. 1986.

[20] R. Leupers, A. Basu, and P. Marwedel. Optimized array index computation in DSP programs. In *Proceedings of the ASP-DAC*. IEEE, February 1998.

[21] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proceedings of the ACM SIGDA 11th International Symposium on System Synthesis*, pages 3–8, December 1998.

[22] R. Leupers and P. Marwedel. *Retargetable Compiler Technology for Embedded Systems*. Kluwer Academic Publishers, 2001.

[23] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. In *Proc. of 1995 ACM Conference on Programming Language Design and Implementation*, 1995.

[24] Lucent Technologies Inc. *DSP1611/17/18/27/28/29 Digital Signal Processor*, January 1998.

[25] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[26] G. Ottoni, S. Rigo, G. Araujo, S. Rajagopalan, and S. Malik. Optimal live range merge for address register allocation in embedded programs. In *Proceedings of the 10th International Conference on Compiler Construction, CC2001, LNCS 2027*, pages 274–288. Springer-Verlag, April 2001.

[27] C. L. M. Potkonjak and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. December 1997.

[28] A. Rao and S. Pande. 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*, pages 128–138, May 1999.

[29] R. Sethi. Complete register allocation problems. *SIAM J. Computing*, 4(3):226–248, September 1975.

[30] R. Sethi and J. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[31] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.