Algorithms for Global Array Reference Allocation in DSPs

Guilherme de L. Ottoni[1]*, **Guido C. S. de Araújo**[1]

[1]Laboratório de Sistemas de Computação – Instituto de Computação – UNICAMP
Cx. Postal 6176 – 13084-971 Campinas, SP, Brasil

ottoni@acm.org, guido@ic.unicamp.br

Abstract. *This paper tackles the Global Array Reference Allocation (GARA) problem, which is an important code optimization problem for processors with restricted addressing modes such as many embedded processors. First it is proposed an algorithm to solve the Simple GARA problem, which assumes a single available address register. Based on this technique, two approaches to General GARA are proposed. Experimental results of the implementation of these techniques in the GCC compiler compare them with previous work in literature.*

1. Introduction

The technological advances in computing systems have stimulated the growth of the embedded systems market, for example in mobile phones, palmtops and automotive control systems. Because of their characteristics, these new applications demand the combination of low cost, high performance and low power consumption. One way to meet these constraints is through the design of specialized processors. However, processor specialization imposes new challenges to the development of software for these systems. In particular, compilers – generally responsible for code optimization – need to be adapted in order to produce efficient code for these new processors.

In the digital signal processing arena, such as in cellular telephones, specialized processors, known as DSPs (Digital Signal Processors), are largely used. DSPs typically have few general purpose registers and very restricted addressing modes. In addition, many DSP applications include large data streams processing, which are usually stored in arrays. As a result, studing array reference optimization techniques became an important task in compiling for DSPs [Liao et al., 1996, Araujo, 1997]. This work studies this problem, known as *Global Array Reference Allocation* (GARA) [Araujo et al., 2002]. The central GARA subproblem consists of determining, for a given set of array references to be allocated to the same address register, the minimum cost of the instructions required to keep this register with the correct address at all program points. In other words, this subproblem is the problem of GARA when only one address register is available, and is called *Simple GARA*. In [Ottoni et al., 2001], we modeled this subproblem as a graph theoretical problem and proved it to be NP-hard. In addition, it was proposed an efficient algorithm, based on dynamic programming, to optimally solve this subproblem under some restrictions. This algorithm is presented in Section 3.1.. Based on it, two techniques to solve GARA were proposed [Ottoni et al., 2001, Ottoni and Araujo, 2002], and they are introduced in Section 3.2.. Experimental results, from the implementation

*Supported by CNPq and CAPES.

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

Figure 1: (a) Code fragment; (b) Representation of optimized code.

of these techniques in the GCC compiler and comparing them with previous work in the literature, are presented in Section 4. and detailed in [Ottoni and Araujo, 2002].

2. The Global Array Reference Allocation Problem

GARA 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. GARA is very important for many DSPs and other embedded processors that do not have more elaborated addressing modes, such as indexed addressing. As an example, consider the C code fragment in Figure 1(a). In a naive code generator, each array reference would require several arithmetic instructions to adjust the `ar` properly. Assume that a single `ar` is available. Consider the equivalent code in Figure 1(b), which is a source level model of the *intermediate representation* code resulting after the optimization described in this paper is applied. In Figure 1(b), the array references have been substituted by pointer uses, which can be directly mapped into indirect addressing mode (available in virtually any processor) using the `ar`. In this representation, `p++` (`p--`) denotes post-increment (decrement) addressing mode, a commonly available addressing mode which increments (decrements) the `ar` at zero-cost during instruction execution. The cost of the code in Figure 1(b) is just the cost of a pointer *update instruction* (`p += 1`) introduced on one of the loop control paths.

3. Proposed Algorithms for GARA

This section presents the algorithms we proposed for solving GARA. As a generalization of the *Local Array Reference Problem*, which was proved to be NP-hard [Araujo, 1997], we are unlikely to find an efficient algorithm to exactly solve GARA in general. With this dificulty in mind, we first focus on the problem when only one address register is available, which we call *Simple GARA* (Section 3.1.). In Section 3.2., we deal with the case of multiple address registers, called *General GARA*.

3.1. Simple GARA

In Simple GARA, all array references must be assigned to the same, single `ar`. The solution we look for is a set of zero-cost addressing modes and costly update instructions that we need to insert in the code in order to keep the `ar` pointing to the correct array element in all program points, no matter the execution path that is taken. Among these possible solutions, we want the one that minimizes the number of update instructions.

In order to find the solution to Simple GARA, we have to make two types of decisions: (1) find the places where we need to decide the array element to which the `ar` will point, and (2) decide to which element the `ar` must point in each of these points so that the update instructions are minimized. To find the places where we have to decide the contents of the `ar`, we translate the code to the intermediate representation called *Extended Single Reference Form* (ESRF) [Ottoni, 2002]. In ESRF, we insert the so-called ϕ-functions in the *Combined Iterated Dominance Frontier*, which we derive from the classical iterated dominance/post-dominance frontiers [Muchnick, 1997]. To solve the ϕ-functions is to choose the array element whose address will be in the `ar` at that point. Figure 2(a) shows the code from Figure 1 in ESRF.

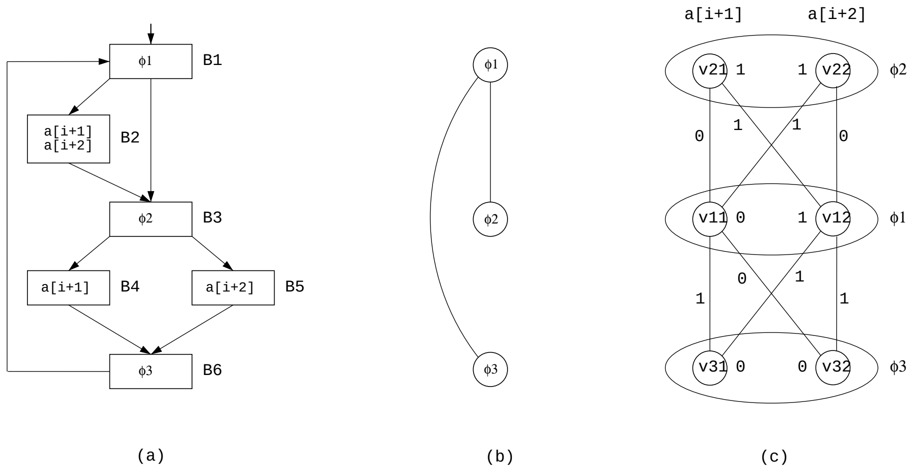

Figure 2: (a) CFG in ESRF; (b) DG$_\phi$ representation; (c) SG$_\phi$ for the DGϕ.

Now we address the problem of solving the ϕ-functions so that the minimum number of update instructions is required. In order to solve this problem, we modeled it as a graph theoretical problem, defining the *ϕ-Dependence Graph* (DG$_\phi$). This graph represents the ϕ-functions' inter-dependences, and Figure 2(b) illustrates the DG$_\phi$ corresponding to Figure 2(a). From the DG$_\phi$, we derive the *ϕ-Solution Graph*, which has a partition for each vertex in DG$_\phi$, and each partition has a vertex for each possible solution. SG$_\phi$ is a weighted graph, with costs (in number of update instructions) both on its vertices and edges. The vertices are valued with a *local cost*, that is the cost of this solution related to the array references on which the ϕ-function depends. The edges are weighted with a *global cost*, that is the cost between two dependent ϕ-functions. The SG$_\phi$ embeds all possible solutions for the set of ϕ-functions. In this graph, we want to find the set of vertices, one from each partition, so that the total cost (local plus global) of the induced graph is minimum. Unfortunately, we found this problem to be NP-hard [Ottoni et al., 2001]. However, we proposed an efficient algorithm to optimally solve this problem for the particular cases when the DG$_\phi$ is acyclic [Ottoni et al., 2001]. This algorithm, called *Leaves Removal Order* (LRO), is based on the dynamic programming technique. The basic idea of LRO algorithm is to iteratively remove the leaves of the DG$_\phi$ and, when processing a leaf node, consider each combination of solution for this node and its parent, accumulating the best costs in the parent node. Applying LRO to our example gives a solution with cost 1, the single update instruction inserted if Figure 1(b) at line 10. This solution consists of choosing the following vertices in Figure 2(c): v_{11}, v_{21} and v_{32}.

3.2. General GARA

Based on the LRO algorithm, we proposed two new techniques to solve GARA when multiple address registers are available. The first one is a heuristic solution, based on the Live Range Growth (LRG) approach used in [Cintra and Araujo, 2000]. It is called *live range* a set of array references that will be assigned to the a single address register. The LRG approach initially assigns each array reference in the loop to a different live range. Then, a greedy strategy takes place, iteratively merging the pair of current live ranges that best improves the total cost. In order to calculate the cost of a live range, [Cintra and Araujo, 2000] always used a heuristic called Tail-Head (TH). Our new technique, named LRO-TH, improves this approach by considering the possibility of using the LRO algorithm to optimally compute the cost of live ranges. So, for a given live range, we first build the DG_ϕ and check whether it is acyclic or not. If it is, we use LRO; otherwise, the TH heuristic is applied. As the experimental results show, more than 93% of the DG_ϕ's were acyclic for the benchmarks we used, enabling an effective use of the LRO algorithm.

We suspected the results of LRO-TH were very close to the optimum solution possible, and this motivated the search for an optimal (although computationally inefficient) solution for GARA. So, we proposed an optimal algorithm, called EXACT, to find the best solution for GARA. This algorithm is based on backtracking, and uses the LRO algorithm whenever possible to improve its running time. The EXACT algorithm tries all the combinations of partitioning the array references in a number of live ranges less than or equal to the number of address registers. For each combination, it is calculated the minimum update cost for each of the live ranges. For each live range, the corresponding DG_ϕ is built and, if it is acyclic, the LRO algorithm is used to compute the minimum cost. In case of cyclic DG_ϕ, we used a combinatorial algorithm to try all the combinations of solutions to the ϕ-functions. Although computationally inefficient, the EXACT algorithm was able to run in reasonable time for research purposes, except in a few cases where the number of array references in the loop was very large. This result can also be attributed to LRO, as the percentage of acylcic DG_ϕ's during EXACT was 99.84%.

4. Experimental Results

To evaluate the efficiency of the LRO-TH and EXACT techniques, we implemented them inside the GNU Compiler Collection [GCC,], using the Lucent DSP16xx [Lucent, 1998] as target processor. For comparison purposes, we also implemented the TH approach. We used the inner-most loops from the MediaBench [Lee et al., 1997] applications as benchmark. Table 1 presents the speedups of TH, LRO-TH and EXACT when comparing to the base GCC compiler. We can see that the speedup achieved by LRO-TH approaches the speedup of the 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%). No compilation-time penalty was observed for LRO-TH and TH.

5. Conclusions and Future Work

We have developed and extensive study of the Simple GARA problem, proving it to be NP-hard and proposing an optimal algorithm (LRO) that can be applied in special cases (when the DG_ϕ is acyclic). Based on LRO, we proposed two techniques to solve General GARA. The first technique is a heuristic solution (LRO-TH) which improves the best

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
`gs`	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 1: Speedup comparison between original GCC, TH, LRO-TH, and EXACT.

previous approach (TH), by [Cintra and Araujo, 2000]. The second technique is an exact, optimal algorithm for GARA, which is used to evaluate the quality of the allocation for the heuristic solutions to the GARA problem. The experimental results show that the LRO-TH approach generally achieves solutions very close to optimal.

Finally, we believe that the DG$_\phi$ concept we defined in this work can be applied to other global code optimization problems. In particular, we have formulated in [Ottoni, 2002] a solution for the global register allocation problem, for general purpose registers. The implementation and evaluation of this technique were left as a future work.

References

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

Araujo, G. (1997). *Code Generation Algorithms for Digital Signal Processors*. PhD thesis, Princeton University.

Araujo, G., Ottoni, G., and Cintra, M. (2002). Global Array Reference Allocation. *ACM Trans. on Design Automation of Electronic Systems*, 7(2):336–357.

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

Lee, C., Potkonjak, M., and Mangione-Smith, W. H. (1997). Mediabench: A tool for evaluating and synthesizing multimedia and communications systems.

Liao, S., Devadas, S., Keutzer, K., and Wang, A. (1996). Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18:235–253.

Lucent (1998). *DSP1611/17/18/27/28/29 Digital Signal Processor*. Lucent Technologies.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers.

Ottoni, G. (2002). Global Address Register Allocation for Array References in DSPs. Master's thesis, State University of Campinas. (in Portuguese).

Ottoni, G. and Araujo, G. (2002). Efficient array reference allocation for loops in embedded processors. In *Proc. of the International Workshop on Embedded System Codesign*, pages 63–68.

Ottoni, G., Rigo, S., Araujo, G., Rajagopalan, S., and Malik, S. (2001). Optimal live range merge for address register allocation in embedded programs. In *Proc. of the 10th International Conference on Compiler Construction, LNCS 2027*, pages 274–288.