Register-Constrained Address Computation in DSP Programs

Anupam Basu*, Rainer Leupers, Peter Marwedel
University of Dortmund
Department of Computer Science 12
44221 Dortmund, Germany
email: basu|leupers|marwedel@ls12.cs.uni-dortmund.de

Abstract

This paper describes a new code optimization technique for digital signal processors (DSPs). One important characteristic of DSP algorithms are iterative accesses to data array elements within loops. DSPs support efficient address computations for such array accesses by means of dedicated address generation units (AGUs). We present a heuristic technique which, given an AGU with a fixed number of address registers, minimizes the number of instructions needed for array address computations in a program loop.[1]

1 Introduction

DSPs are a special class of embedded processors, which show highly specialized instruction sets and pose challenges both to compilers and assembly programmers. Many of today's C compilers for DSPs have been shown to produce code of poor quality. In order to overcome this problem, new DSP-specific code optimization techniques are required.

In DSP algorithms frequent references to elements of data arrays are very common. Mostly, such array elements are iteratively accessed in loops. DSPs support this scheme by dedicated address generation units (AGUs), which are capable of performing pointer arithmetic in parallel to the operation of the central data path. Clever allocation of array address pointers in a DSP source program to available on-chip address registers thus can enhance code quality.

We give a formulation of this register allocation problem and present a heuristic algorithm which, under a given register constraint, minimizes the number of machine instructions for array address computations. Our approach, which handles both register constraints and inter-iteration dependencies in a loop, eliminates some restrictions of earlier work [1, 2, 3] in this area. It is complementary to work done on optimized addressing of *scalar* program variables [4, 5].

2 Problem definition

The AGUs of most DSPs offer post-increment or post-decrement operations on address registers. These operations modify the contents of a register R (serving as the pointer to an array element) by adding some constant integer d. Thus, if the two array elements $A[i]$ and $A[i+d]$ are to be accessed consecutively by the same address register R, then the post-increment operator $R+d$ applied after accessing $A[i]$, will yield the necessary next address. The range of *efficient* post-increment/decrement is restricted to a maximum range M, for only within this range the address update operations can be done *in parallel* to data path operations. Whenever two consecutive array accesses take place through the same register R and the address distance $d > M$, then *one extra instruction* is required to compute the next address.

This observation induces a partitioning of address computations into *zero-cost* and *unit-cost* computations. Our goal is to allocate the N array accesses in a program loop to a given number K of address registers, such that the number of unit-cost computations is minimized.

Let us consider an example array access pattern to illustrate the problem:

```
for (i = 2; i <= N; i++)
{ /* a_1 */   A[i+1]   /* offset  1  */
  /* a_2 */   A[i]     /* offset  0  */
  /* a_3 */   A[i+2]   /* offset  2  */
  /* a_4 */   A[i-1]   /* offset -1  */
  /* a_5 */   A[i+1]   /* offset  1  */
  /* a_6 */   A[i]     /* offset  0  */
  /* a_7 */   A[i-2]   /* offset -2  */
}
```

Assume that a linear arrangement of array elements in a contiguous address space is used. Assume further that a maximum modify range $M = 1$ is given. We can represent the access pattern by means of a graph $G = (V, E)$ (fig. 1), in which each node represents an array access, denoted by its offset w.r.t. the loop variable. An edge $(a_i, a_j) \in E$, with $i < j$, indicates that, within one loop iteration, computing the address for a_j from the address of a_i can be done at zero cost, because the address distance is $\leq M$. That is, no unit-cost computation would be incurred, if a_i, a_j shared an address register. We can insert additional graph edges which represent the same relation *between consecutive loop iterations*.

Due to the construction of graph edges, each *path* in G represents an opportunity for allocating multiple accesses to a single register without incurring unit-cost

*On leave from IIT Kharagpur, India

[1] Publication: DATE, Paris/France, Feb 1998, ©1998 EDAA

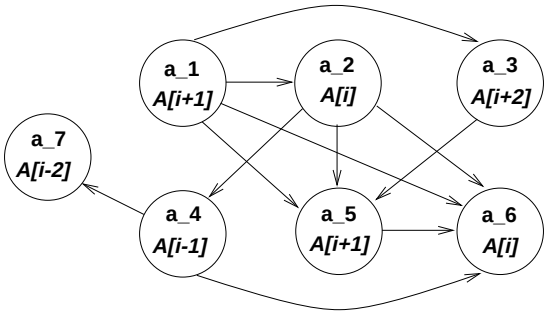

Figure 1: *Graph model for the example loop*

address computations. For instance, the access subsequence (a_1, a_3, a_5, a_6) (which is a path in G) could be realized with a single register R and using only auto-increment and auto-decrement operations on R. One can show that completely covering G by K node-disjoint paths corresponds to a zero-cost allocation of all array accesses to K registers.

3 Address register allocation

Given a loop with N array accesses, it is in general not possible to achieve a zero-cost addressing scheme with K registers. We use a two-phase approach to heuristically compute a minimum-cost addressing scheme:

1. Compute the minimum number $\tilde{K}$ of "virtual" registers, for which all addresses for array accesses could be computed by zero-cost computations only. If $\tilde{K} \leq K$, then a zero-cost allocation under the given register constraint is possible.

2. Otherwise, if $\tilde{K} > K$, select two registers R_i, R_j with $i, j < \tilde{K}$. Merge the sets of array accesses assigned to R_i, R_j and decrement $\tilde{K}$. Repeat this step, until the number K of physical registers is no longer exceeded.

In the following, we explain these two phases in more detail.

3.1 Computation of $\tilde{K}$

If address computation dependencies across single loop iterations are taken into account, then computing $\tilde{K}$ is an exponential problem. In [3], we have proposed a fast branch-and-bound procedure for solving this problem. The procedure computes a cover of the graph model G by $\tilde{K}$ node-disjoint paths. We use a technique proposed in [2] to compute a lower bound on $\tilde{K}$, and a heuristic algorithm for determination of a tight upper bound. Based on these bounds, one can quickly decide whether or not a certain graph edge e must be included in the path cover. The result is a set $\{P_1, \ldots, P_{\tilde{K}}\}$ of node-disjoint paths. Each path is mapped to one address register and represents a subsequence of the original array access pattern.

3.2 Meeting the register constraint

If $\tilde{K} > K$, then the number of paths must be reduced further, so as to meet the physical register limit. This can be done by *merging of paths*. The merge operation "$\circ$" retains the order of array accesses in the original access pattern. For instance, merging paths $P_1 = (a_1, a_4, a_6)$ and $P_2 = (a_3, a_5)$ results in the path $P_1 \circ P_2 = (a_1, a_3, a_4, a_5, a_6)$.

Selecting candidates for path merging is based on the notion of *path costs*. The costs $C(P)$ of a path $P = (a_{i_1}, \ldots, a_{i_{n_i}})$ are defined as the number of pairs $(a_{i_k}, a_{i_{k+1}})$ in P, such that the address distance between a_{i_k} and $a_{i_{k+1}}$ is larger than M. That is, $C(P)$ denotes the number of unit-cost address computations required for the array references represented by P.

Two implications hold by definition of $\tilde{K}$: The path costs for each element of the initial path set $\{P_1, \ldots, P_{\tilde{K}}\}$ are zero. Furthermore, each merge operation incurs at least one unit-cost address computation. In order to minimize the total number of unit-cost address computations incurred by merging, it is reasonable to select that pair (P_i, P_j) of paths in $\{P_1, \ldots, P_{\tilde{K}}\}$ for merging, such that $C(P_i \circ P_j)$ is minimal among all pairs. After merging of (P_i, P_j), we obtain a new path set $\{P_1, \ldots, P_{\tilde{K}-1}\}$. On this set, merging is iterated until only K paths are left.

4 Results

In order to determine the net effect of the proposed path-merging heuristic, we have performed a statistical analysis as compared to a non-optimized address register allocation, which repetitively merges two arbitrary paths until the register constraint is met.

We have determined the number of unit-cost address computations for random access patterns and a variety of parameters N, M, and K. As a result, we have observed that the address register allocation determined by path merging reduces the addressing cost by about 40 % on the average, as compared to the "naive" solution. Experimental studies for realistic DSP programs [1] indicate possible improvements up to 30 % and 60 % in code size and speed due to optimized array index computation, as compared to code compiled by a regular C compiler.

References

[1] C. Liem, P.Paulin, A. Jerraya: *Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures*, 33rd Design Automation Conference (DAC), 1996

[2] G. Araujo, A. Sudarsanam, S. Malik: *Instruction Set Design and Optimizations for Address Computation in DSP Architectures*, 9th Int. Symp. on System Synthesis (ISSS), 1996

[3] R. Leupers, A. Basu, P. Marwedel: *Optimized Array Index Computation in DSP Programs*, ASP-DAC, 1998

[4] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Storage Assignment to Decrease Code Size*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995

[5] R. Leupers, P. Marwedel: *Algorithms for Address Assignment in DSP Code Generation*, Int. Conf. on Computer-Aided Design (ICCAD), 1996