Array Index Allocation under Register Constraints in DSP Programs

Anupam Basu*
I.I.T. Kharagpur
India
Dept. of Computer Science & Engineering
email: basu@ls12.cs.uni-dortmund.de

Rainer Leupers, Peter Marwedel†
University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany
email: leupers|marwedel@ls12.cs.uni-dortmund.de

Abstract– Code optimization for digital signal processors (DSPs) has been identified as an important new topic in system-level design of embedded systems. Both DSP processors and algorithms show special characteristics usually not found in general-purpose computing. Since real-time constraints imposed on DSP algorithms demand for very high quality machine code, high-level language compilers for DSPs should take these characteristics into account. One important characteristic of DSP algorithms is the iterative pattern of references to array elements within loops. DSPs support efficient address computations for such array accesses by means of dedicated address generation units (AGUs). In this paper, we present a heuristic code optimization technique which, given an AGU with a fixed number of address registers, minimizes the number of instructions needed for address computations in loops.[1]

1 Introduction

Heterogeneous hardware/software systems are finding increasing use as embedded systems in industrial applications. Though software forms a principal component in such systems, software development is still a bottleneck [1]. Instead of the high compilation speed requirement, as for general-purpose compilers, software generation for embedded systems must attend to better code quality, in terms of code size and execution speed. Digital signal processors (DSPs) form 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 [2]. In order to overcome this problem, a number of research efforts have been launched, aiming at development of new DSP-specific code optimization techniques [3].

*The author acknowledges the support of the Humboldt Foundation for this research

†The authors acknowledge the support by HP EEsof

[1]Publication: 12th Int. Conf. on VLSI Design, Goa (India), Jan 1999, ©IEEE

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. Careful allocation of array address pointers in a DSP source program to available on-chip address registers thus can enhance code quality. The purpose of this paper is to give a formulation of this register allocation problem and to present a heuristic algorithm which, under given register constraints, minimizes the number of machine instructions for array address computations. The paper is organized as follows. In the next section we define the problem. In Section 3, we summarize related works. The proposed approach and algorithms are described in the next two sections, followed by performance evaluation and conclusion.

2 Problem definition

Typical DSP algorithms, such as digital filters, demonstrate that the address distance of subsequently accessed array elements are bounded by a small constant. Moreover, the array index expressions are simple and the loop control variable shows a small constant step width between iterations. Accordingly, the address generation units in DSPs, such as DSP56K (Motorola) and TMS320C2X/5X (Texas Instruments) offer post-increment or post-decrement operations on address registers. These operators increment or decrement the content of a register R (serving as the pointer to an array element) by 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.

In such architectures, the range of post-increment/decrement is restricted to a maximum range M, for efficient address computation. Within this range, the address update operations can be done exclusively by the address generation unit (AGU) in parallel to data path operations. We call such address computations *zero cost address computations*. However, for a larger range of modifications of the address registers, an extra instruction word is necessary in the machine code, since the encoding of d values larger than M cannot be accommodated within the same instruction word. This implies an additional instruction cycle in the machine program, as this address computation cannot be parallelized. Thus, whenever two consecutive data accesses take place through the same register R and the address distance $d > M$, then one additional computation is required. We call this overhead *unit reload cost*.

Given a sequence of array references and a set of *available* address registers, one of the goals of a code generator should be to minimize the *total reload cost*. A trivial case is of course to have as many address registers as there are array references. In that case, there would obviously be *zero reload cost*. However, in view of the limited address registers available and a distribution of array references, the problem is nontrivial.

Example: Let us consider an example array reference pattern to illustrate the problem. We shall be referring to this example repeatedly throughout the paper.

```
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  */}
```

It is convenient to plot the access pattern on a grid-structure, where the rows indicate the control steps and the columns denote the offsets. In Fig.1 the X symbol shows the accesses on the grid. Let us assume that the value of $M = 1$, that is there are only auto-increment and auto-decrement operations available on the address registers. Let us assume further that there are only two address registers available for accessing the array elements. Two clusters A and B are shown in Fig. 1(a), each cluster representing one register using which the array elements within a cluster are referenced. Note that for each consecutive access in the cluster, the offset difference is one, and hence can be dealt with using the auto-increment/auto-decrement facilities only. However, as shown in Fig. 1(b), if the clusters were formed differently, then in order to access the reference in control step 7, it would be necessary to reload the address register, supporting cluster A, with the new address value (shown by the arrow). Thus, while the first clustering does not introduce any reload overhead, *in that iteration*, the second clustering introduces unit reload cost. In the upper part of Fig. 1(a), we show the inter-iteration situation (control steps 7 and 8). As may be noted, the reference at control step 1 (offset 1) will be re-accessed at control step 8 (the first control step of the next iteration) and the offset value will be 2 instead of 1, because of the iteration step width. Since

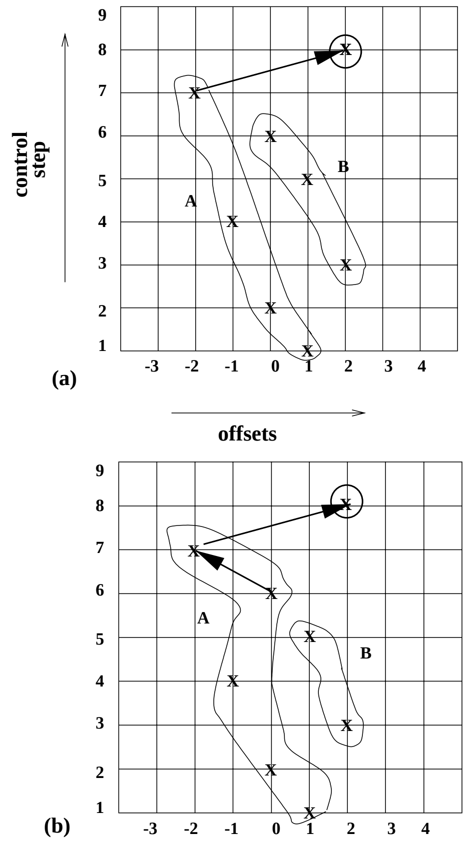

Figure 1: *A grid model to illustrate reloads*

the reference at control step 1 and 8 must take place by the same register[2], an *inter-iteration reload cost* is introduced (shown by the arrow), though there were no *intra-iteration reloads* necessary for the same clustering.

Let us denote an array reference p_j as a pair (of_j, cs_j), where of_j denotes the offset of the reference and cs_j denotes the control step in which the reference is made.

Let $C_k = \{p_k\}$ denote a cluster, that is an address register, using which the references p_k are addressed.

Definition 1: If within a cluster C_k, two consecutive array references p_{k1} and p_{k2} have $|of_{k2} - of_{k1}| > M$, then **a unit reload cost** is introduced.

Definition 2: (inter-iteration reload cost) Let the first reference p_1 and the last reference p_n *in a particular iteration*, in a cluster C_k be such that $|of_1 + step - of_n| > M$, where *step* denotes the inter-iteration step value, then a unit *inter-iteration reload cost* is introduced.

This is obvious, since in the next iteration, the reference p_1 should be addressed by the same address register specified for cluster C_k.

Definition 3: (total reload cost (TC)) The *total reload cost* introduced by clustering the references into

[2]Note that this constraint could be eliminated by applying *loop unrolling* which, however, tends to increase the code size.

address registers is denoted by $TC = \sum_{i=1}^{m}(rc_i + irc_i)$, where m is the total number of clusters and rc_i and irc_i denote the total reload cost and the total inter-iteration reload cost introduced in a cluster C_i.

The problem of **address register allocation** can now be stated as follows:

Given: a set of address registers $A = \{a_i | 1 < i < m\}$ and a pattern of array references $P = \{p_j | 1 < j < n\}$, where each p_j is an ordered pair (of_j, cs_j), of_j denoting the index of an array referred at control step cs_j.

Required: an allocation of all elements of P to the elements of A, so that the total reload cost (TC) is minimized.

Before presenting our approach to solve the problem, we briefly discuss other works related to this problem area.

3 Related works

Several researchers have investigated DSP-specific optimization techniques for address computation for *scalar* program variables [4, 5, 6, 7]. These approaches are based on permutation of variables within available sections of memory. Hence, these techniques cannot be directly applied to arrays. More recently, addressing optimization techniques related to array accesses have been considered. In [8], a C source-to-source transformation is described, which minimizes number of required array pointers. In a contribution by members of the SPAM project [9] an address register allocation algorithm for loops has been given, which makes use of a graph-based problem formulation. It was shown that this formulation enables the use of an optimal *matching-based path covering* algorithm [10] for register allocation. However, this approach neglects both register constraints and inter-iteration reload cost. The approach by Gebotys [11] accepts register constraints but it is also limited to a single loop iteration. Exploiting the results of [9] as a lower bound, and a heuristic algorithm for upper bound, an optimal branch and bound algorithm has been presented to find the minimum number of registers required to ensure *zero reload cost solution* [12]. However, in this work register constraints are not taken into account, and consequently, the minimum number of registers for a zero reload cost solution might exceed the number of available registers. In this case the allocated registers have to be taken as "virtual" registers which still have to be mapped to a smaller number of physical registers.

4 Array access optimization

Now we address ourselves to the issue of optimization of array accesses in loops given a constrained set of address registers. We approach the solution in two passes. In the first pass we arrive at a quick estimate of the *upper bound* of the number of address registers required to ensure zero reload cost. If the number of registers, thus obtained exceeds the given constraint on the set of available registers, we adopt the second pass, where we attempt to merge the accesses allocated to some of the registers with others. Such mergings may introduce reload costs, and hence the objective of the merging should be to minimize the incremental TC.

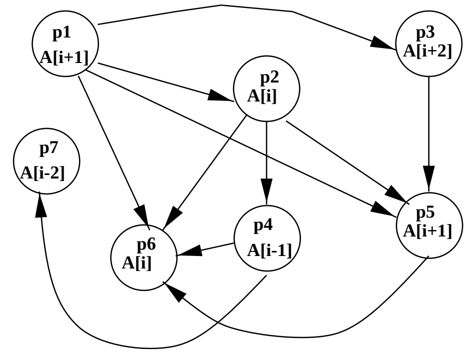

Figure 2: *Distance graph for the example loop*

4.1 Arriving at an upper bound

In order to arrive at a tight upper bound of the number of registers required to ensure zero reload cost we model the problem as an *Iteration Distance Graph.* In order to present this concept, a review of some background seems to be in order. We first present the notion of a *distance graph.*

Definition: Let $(a_1, \ldots, a_n)$ be a sequence of array references in a loop. For all a_i, a_j, with $1 \leq i < j \leq n$, the **intra-iteration distance** $\delta(a_i, a_j) := \overline{f}(a_j) - \overline{f}(a_i)$ is the (constant) offset difference between a_i and a_j in a fixed loop iteration. Let S be the loop step-width. The **inter-iteration distance** $\delta'(a_i, a_j) := -\delta(a_i, a_j) + S$ is the (constant) offset difference between a_j in the *current* loop iteration and a_i in the *following* iteration.

Let M denote the maximum modify range. The **distance graph** $G = (V, E)$ is a directed acyclic graph (DAG) with $V = \{a_1, \ldots, a_n\}$. The edge set E contains all edges $e = (a_i, a_j)$ with $1 \leq i < j \leq n$ and $|\delta(a_i, a_j)| \leq M$.

An edge $e = (a_i, a_j)$ is present in E, if using the same address register for both a_i and a_j allows for generating the address for a_j from the address for a_i with a zero-cost address computation. Fig. 2 shows the distance graph for our above example loop and $M = 1$.

According to the definition of the distance graph $G = (V, E)$, any subsequence $(a_{k_1}, \ldots, a_{k_m})$ of an array reference sequence $(a_1, \ldots, a_n)$ can be implemented by zero-cost address computations, only if for all $k_i \in \{1, \ldots, m-1\}$ the edge $e = (a_{k_i}, a_{k_j})$ is present in E. That is, there must exist a *path* $P = (a_{k_1}, \ldots, a_{k_m})$ in G. However, since the address of reference a_{k_1} for the *next* loop iteration must be computed from the address of a_{k_m} in the *current* iteration, it must be also ensured that the difference between those two addresses does not exceed the maximum modify range M. Otherwise, a unit-cost address computation would be required. Thus, if the objective is to minimize the number of registers required for a zero cost solution, it could be obtained by finding a minimum path cover of the distance graph G, i.e., a minimum number K of node-disjoint paths $P_1, \ldots, P_K$ in G, such that all nodes in V are touched by exactly one path, and for each path $P_k = (a_{k_1}, \ldots, a_{k_m})$ it holds that $|\delta'(a_{k_1}, a_{k_m})| \leq M$.

In [9] it has been proposed to apply a matching-based algorithm developed in the area of graph theory [10] to address register allocation. This algorithm computes a minimum path cover of the distance graph, however without considering post-modify operations *across loop iteration boundaries*. That is, in our terms, the constructed paths $P_k = (a_{k_1}, \ldots, a_{k_m})$ do not necessarily satisfy $|\delta'(a_{k_1}, a_{k_m})| \leq M$. As a consequence, unit-cost address computations can be incurred, unless the computed cover by coincidence represents a zero-cost solution.

However, if each path $P_k = (a_{k_1}, \ldots, a_{k_m})$ must satisfy $|\delta'(a_{k_1}, a_{k_m})| \leq M$, then we must reformulate the model by including inter-iteration distances in the distance graph model, yielding the proposed *Iteration Distance Graph* defined below.

Definition:

Let $G = (V, E)$ with $V = \{a_1, \ldots, a_n\}$ be the distance graph of a loop. The **iteration distance graph** is a DAG $G' = (V', E')$ with $V' = V \cup \{a'_1, \ldots, a'_n\}$, where each node $a'_i \notin V$ represents the array reference a_i in the *following* loop iteration, and

$$E' = E \cup \{(a_j, a'_i) \quad | \quad 1 \leq i \leq j \leq n \wedge |\delta'(a_i, a_j)| \leq M\}.$$

Presence of an edge $e = (a_j, a'_i)$ in E' indicates, that if the references a_i and a_j share an address register R, then the address computation on R *between loop iterations* can be implemented at zero cost. Thus, computing a zero-cost solution with a minimum number of address registers is equivalent to covering all nodes $\{a_1, \ldots, a_n\}$ in the iteration distance graph by a minimum number of node-disjoint paths $P_1, \ldots, P_K$, such that if a path P_k starts in node a_i it must end in node a'_i. The iteration distance graph for the example problem is shown in Fig.3.

As a special case, this problem comprises the decision whether a DAG can be covered by two node-disjoint paths with given start and end nodes. Since this problem is NP-complete [13] the address register allocation problem is most likely of exponential complexity. However, it is possible to compute a (potentially suboptimal) solution efficiently, if address registers are allocated greedily based on the following longest path based heuristic.

1. Given a distance graph $G = (V, E)$, construct the extended distance graph $G' = (V', E')$ with $V = \{a_1, \ldots, a_n\} \cup \{a'_1, \ldots, a'_n\}$, and assign a unit weight to each edge $e \in E'$.

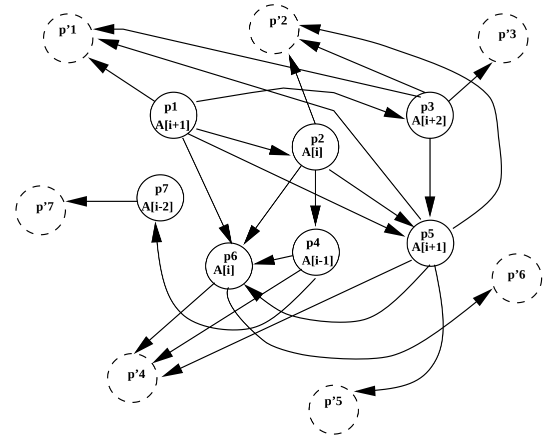

Figure 3: *Iteration Distance Graph*

2. Let a_i be the source node in $\{a_1, \ldots, a_n\} \subset V'$ with minimum index, i.e., there is no node a_j with $(a_j, a_i) \in E'$ and $j < i$. Compute the longest path $P = (a_i, a_{k_1}, \ldots, a_{k_m}, a'_i)$ in G' between a_i and a'_i. If P does not exist then stop, because no zero-cost solution is possible.

3. Allocate a new address register for the array references represented by the nodes $\{a_i, a_{k_1}, \ldots, a_{k_m}\}$ in path P. Remove these nodes as well as the nodes $\{a'_i, a'_{k_1}, \ldots, a'_{k_m}\}$ from G', and remove all their incident edges.

4. If G' is not empty goto step 2, else stop and return the number r of allocated registers.

Below we show a longest path solution to the example problem. This solution results in three registers addressing the following references. None of the registers require any inter-iteration or intra-iteration reloads.
$R1 : a_1, a_3, a_5, a'_1$;
$R2 : a_2, a_4, a_6, a'_2$;
$R3 : a_7, a'_7$

Thus the references would be

```
R1 = &A[3]
R2 = &A[2]
R3 = &A[0]
for (i = 2; i <= N; i++)
{  /* a_1 */    *R1 ++
   /* a_2 */    *R2 --
   /* a_3 */    *R1 --
   /* a_4 */    *R2 ++
   /* a_5 */    *R1 ++
   /* a_6 */    *R2 ++
   /* a_7 */    *R3 ++
}
```

The longest path heuristics ensures obtaining a zero-cost-solution (if one exists), but may result in suboptimal number of address registers.

5 Satisfying register constraints by Path Merging

The longest path algorithm explained in the previous section provides a tight upper bound on the number of address registers required. In the following, we will refer to this algorithm as **FIND-TUB** (find tight upper bound). If the number of required address registers exceeds the number of *available* address registers, then the accesses allocated to some of the registers can be merged with others in a way that minimizes the incremental reload cost. The process of merging is iteratively performed till the number of address registers required equals the number of available registers. The merging algorithm **MERGE** will be explained in the following.

5.1 MERGE

The algorithm **FIND-TUB** will return the clusters or paths along with the upper bound **Reg-Bound**. If **Reg-Bound** exceeds the number of available registers then the algorithm **MERGE** is iteratively applied, until a solution is obtained or there are no further possibilities of merging, that reduce the value of **Reg-Bound**.

Let **REQ** denote the number of clusters (paths). It is initialized to the number of paths (that is the number of registers) returned by **FIND-TUB**. Let C be the set of all clusters, $C = \{C_k\}$, where $C_k = \{p_{k_i}\}$, with each p_{k_i} is of the form (of_{k_i}, cs_{k_i}). Let the set of all possible *candidate* combinations be $\tilde{C} \subset C \times C$, such that $\forall m, n, m \neq n, (C_m, C_n) \in \tilde{C}$. Note that for any i in $p_{k_i} \in C_k$, $|of_{k_i} - of_{k_{i+1}}| \leq M$.

Now let us consider the procedure (**Path-Merge-Cost**) to merge two candidate paths C_i and C_j belonging to $\tilde{C}$ and compute the associated reload costs introduced.

Path-Merge-Cost (C_i, C_j)**:**
1. Let the nodes of the two paths C_i and C_j be concatenated in a list L. Each element of L will be a 3-tuple (path-id, offset, control step).
2. Sort L on the values of the control steps of the elements.
3. Initialize TC_{ij} to zero.
4. Let s and t be the number of nodes in C_i and C_j respectively.

for index = 1 to $s + t - 1$ **do**
  if L[index].path-id $\neq$ L[index + 1].path-id **and**
  $|L$[index].offset - L[index+1].offset$| >$ M
   $TC_{ij} := TC_{ij} + 1$

5. Return (L and TC_{ij})

The merging of two paths is illustrated in Fig. 4. The directed edges show the points of merging, labelled with associated reload costs. Now we present the algorithm **MERGE**, which iteratively invokes **Path-Merge-Cost**.

1. Form $\tilde{C}$.
2. Initialize REQ.
3. For all elements in $\tilde{C}$ invoke **Path-Merge-Cost**
4. Find the element of $\tilde{C}$ with minimum TC_{ij}. In case of conflict resolve arbitrarily. Let the chosen element be (C_x, C_y). Delete this element from $\tilde{C}$ and add the merged path (C_x, C_y) to $\tilde{C}$.
5. Decrement REQ

As mentioned at the beginning of this section, **MERGE** is called iteratively, until all possibilities are exhausted or a combination of paths are found which can be allocated to the available registers.

On applying the path merging approach to the longest path solution, shown earlier, *subjected to the constraint of two available address registers*, we obtain the following solution with a TC value of 2. Here, the accesses by registers R2 and R3 has been merged. The reload costs stem from transitions from references a_6 to a_7 and from a_7 to a'_2 in register R2.

$R1 : a_1, a_3, a_5, a'_1;$
$R2 : a_2, a_4, a_6, a_7, a'_2$

5.2 Complexity

The address register allocation technique shown in the beginning of this section is a polynomial-time procedure. It utilizes two algorithms, the **FIND-TUB** algorithm to compute the minimum number of registers and the **MERGE** algorithm for combining the paths. The first algorithm is of complexity $\mathcal{O}(|V|^2 \cdot |E|)$, where V and E denote respectively the vertex and edge sets of the *Iteration Distance Graph*. The **Path-Merge-Cost** procedure of **MERGE** is of linear complexity in the number n of array accesses, while the invocation of this procedure by **MERGE** is bounded by $\mathcal{O}(n^2)$. Hence, the worst case complexity of **MERGE** is $\mathcal{O}(n^3)$. Since $|V| = \mathcal{O}(n)$, the total runtime is dominated by **FIND-MIN** and is in $\mathcal{O}(n^4)$. In practice this means, that the computation time is in within the range of CPU milliseconds on a SparcStation-10.

6 Performance

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. Table 1 gives the results, diversified with respect to three parameters: the length N of the array access sequence, the maximum auto-increment range M, and the number k of available address registers. Columns 2 and 3 show the average total reload cost (TC) obtained by non-optimized and optimized address register allocation, respectively. Each average TC value was computed over a set of 100 random array reference patterns. Column 4 gives the percentage of cost reduction achieved by the path-merging technique as compared to the non-optimized allocation.

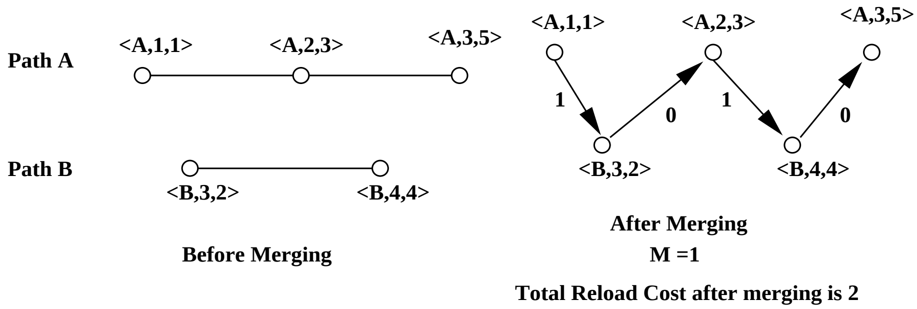

Figure 4: *Example of Path Merging*

parameter	TC non-opt	TC opt	cost red.
$N = 5$	0.35	0.31	11 %
$N = 10$	1.10	0.80	27 %
$N = 15$	2.01	1.31	35 %
$N = 20$	3.00	1.84	39 %
$N = 25$	4.03	2.36	41 %
$M = 1$	5.18	3.50	32 %
$M = 3$	2.36	1.37	42 %
$M = 7$	0.74	0.37	50 %
$M = 15$	0.10	0.05	54 %
$k = 2$	4.12	2.87	30 %
$k = 4$	1.83	0.95	48 %
$k = 8$	0.33	0.16	53 %
total			40 %

Table 1: *Experimental results*

On the average, a cost reduction of 40 % has been observed. The diversified results show, that the proposed optimization technique is robust with respect to variation of AGU resources, i.e., larger auto-increment ranges (M) and larger number of address registers (k) lead to higher cost reductions. Moreover, the cost reduction grows with the reference sequence length N.

7 Conclusion

In this paper, we have presented a DSP-specific code optimization technique, which minimizes the number of explicit (i.e., non-parallel) address computations for array accesses in loops. To our knowledge, the proposed path-merging technique is the first concrete algorithm to tackle this problem for a given constraint on the number of available AGU address registers. Experimental results show that by performing the path-merging technique one obtains a 40 % reduction on the number of explicit address computations as compared to a non-optimizing address register allocation technique.

References

[1] P. Paulin, M. Cornero, C. Liem, et al.: *Trends in Embedded Systems Technology*, in: M.G. Sami, G. De Micheli (eds.): *Hardware/Software Codesign*, Kluwer Academic Publishers, 1996

[2] V. Zivojnovic, H. Schraut, M. Willems, R. Schoenen: *DSPs, GPPs, and Multimedia Applications – An Evaluation Using DSPStone*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1995

[3] P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[4] D.H. Bartley: *Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes*, Software – Practice and Experience, vol. 22(2), 1992, pp. 101-110

[5] 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

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

[7] B. Wess, M. Gotschlich: *Constructing Memory Layouts for Address Generation Units Supporting Offset 2 Access*, Proc. ICASSP, 1997

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

[9] 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

[10] F.T. Boesch, J.F. Gimpel: *Covering the Points of a Digraph with Point-Disjoint Paths and Its Application to Code Optimization*, Journal of the ACM, vol. 24, no. 2, 1977, pp. 192-198

[11] C. Gebotys: *DSP Address Optimization using a Minimum Cost Circulation Technique*, nt. Conf. on Computer-Aided Design (ICCAD), 1997

[12] R.Leupers, A. Basu, P.Marwedel: *Optimized Array Index Computation in DSP Programs*, Proc. of ASP-DAC '98, Yokohama/Japan, Feb 1998

[13] N. Robertson, P.D. Seymour: *An Outline of Disjoint Path Algorithms*, pp. 267-292 in: B. Korte, L. Lovasz, H.J. Prömel, A. Schrijver (eds.): *Paths, Flows, and VLSI Layout*, Springer-Verlag, 1990