Optimized Array Index Computation in DSP Programs

Rainer Leupers, Anupam Basu[*], Peter Marwedel

University of Dortmund
Department of Computer Science 12
44221 Dortmund, Germany
e-mail: leupers|basu|marwedel@ls12.cs.uni-dortmund.de

Abstract— An increasing number of components in embedded systems are implemented by software running on embedded processors. This trend creates a need for compilers for embedded processors capable of generating high quality machine code. Particularly for DSPs, such compilers are hardly available, and novel DSP-specific code optimization techniques are required. In this paper we focus on efficient address computation for array accesses in loops. Based on previous work, we present a new and optimal algorithm for address register allocation and provide an experimental evaluation of different algorithms. Furthermore, an efficient and close-to-optimum heuristic is proposed for large problems.[1]

I. Introduction

A promising approach to achieve reduced design cycle times for embedded VLSI systems is indicated by the recent trend to migrate from hardware to software implementation of system components. In contrast to custom hardware, software executed on *embedded processors* offers higher flexibility and also facilitates reuse of predefined system components.

Even though construction of software compilers for programmable processors has been subject to intensive research for decades, recent surveys [1, 2] show, that software development still is a bottleneck for embedded processors, because of unacceptable code quality of high-level language compilers. Therefore, time-consuming assembly-level programming is often the only feasible alternative. In particular, this holds for *digital signal processors* (DSPs). Many current C compilers for DSPs have been shown to produce very poor code [1].

While compilers for general-purpose computers usually have to be very fast, lower compilation speed is acceptable for embedded software development. Based on this paradigm, progress in code quality has been made by novel techniques for *phase coupling*, i.e., a tight integration of *code selection*, *register allocation*, and *scheduling* during code generation [3].

A rather new area of DSP code optimization is *memory address generation*. DSPs are equipped with dedicated *address generation units* (AGUs), capable of performing pointer arithmetic in parallel to the central data path. High utilization of AGUs achieved by special compilation techniques increases potential parallelism and therefore allows for more compact machine code.

In this paper, focus is on efficient *generation of memory addresses for array references* in loops. More specifically, we present algorithms that answer the question: Given a loop with a certain array reference pattern, what is the minimum number of address registers (ARs) needed to avoid code size and speed overhead due to explicit address computations ? We discuss two heuristic algorithms and combine them to a new and optimal AR allocation algorithm. Furthermore, we provide experimental results for all three algorithms.

The organization of the paper is as follows. In section II, we define the problem of *AR allocation* encountered in DSP programming. Section III outlines related work in the area. In sections IV and V, we summarize two heuristic algorithms for AR allocation. In section VI we show how to utilize both algorithms for an optimal branch-and-bound procedure. Section VII gives an experimental evaluation of the three algorithms, and the paper ends with concluding remarks.

II. Problem definition

The design of address generation units (AGUs) in DSPs is guided by the general observation, that DSP algorithms such as digital filters show a *high locality in accessing elements of data arrays.* That is, the address distance of subsequently accessed array elements is frequently bounded by a small constant. Furthermore, array index expressions tend to be rather simple and mostly require an addition of a loop control variable and a constant. In many other cases, such a simple form can be constructed by *induction variable elimination* [4].

In order to effectively support these special circumstances, AGUs in DSPs are capable of *post-modify* operations on ARs. For an AR R, a post-modify operation is an assignment $R := R + d$, which increments (or decrements) R by some constant integer *modify value* d. If the address computations for two subsequent array references, say $A[i]$ and $A[i+d]$, are implemented by the same AR R, then executing the post-modify operation $R := R + d$ on R after the access to $A[i]$ provides the necessary next address for accessing $A[i+d]$.

This AGU scheme is found in many DSPs, such as Motorola DSP56k and Texas Instruments TMS320C2x/5x. The corresponding AGU architecture is sketched in fig.

[*]On leave from IIT Kharagpur, India, supported by Humboldt Fellowship

[1]Publication: ASP-DAC, Yokohama/Japan, Feb 1998, ©IEEE

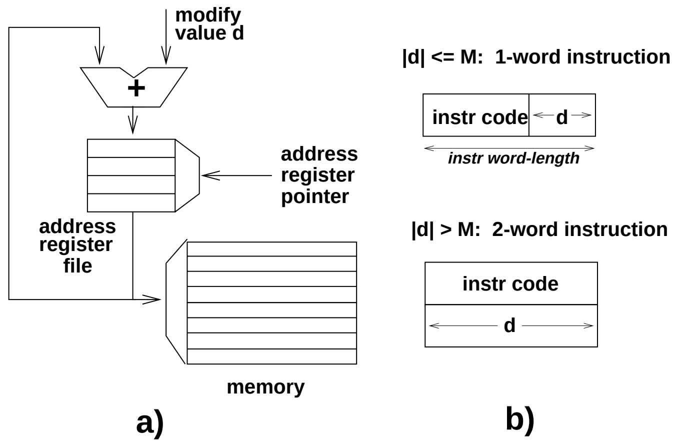

Fig. 1. *a) Partial AGU architecture, b) zero-cost and unit-cost address computation*

1 a). The *address register pointer* is usually part of the instruction word, so that switching between ARs does not require an extra instruction.

In such an architecture, the range of modify values that can be implemented *efficiently* is restricted to a *maximum modify range* M. For $d \in [-M, M]$ a post-modify operation $R := R + d$ can be executed by AGU resources only and thus *in parallel* to other data path operations. We call this a *zero-cost address computation.*

Larger modify values are still possible, but for $|d| > M$, an extra instruction word in the machine code is necessary, since the encoding of large d values no longer fits into the limited instruction word-length. Since such an address computation cannot be parallelized, also an additional cycle in the machine program is incurred. Thus, whenever the address distance between two subsequent memory accesses is larger than $|M|$, and both accesses take place via the *same* AR, then a *unit-cost address computation* is required (fig. 1 b). For sake of exposition, we assume $M = 1$ (auto-increment/decrement) in the following, although the presented algorithms work for arbitrary M values.

Given a sequence of array references in a loop, one must organize address computations in such a way, that the use of unit-cost address computations is minimized. If the organization is such that *only zero-cost* address computations are required, we call this a *zero-cost solution.* Zero-cost solutions are highly desirable, since the speed penalty of each unit-cost address computation is multiplied by the (usually large) number of loop iterations. Due to the limited number of available registers, it is reasonable to minimize the number of registers used for array index computation by *sharing of ARs.* An extension of the work presented here, capable of handling register constraints, is described in [5].

Two array references a_1 and a_2 within a loop body can potentially share an AR, if the distance of the memory locations accessed by a_1 and a_2 is constant over all loop iterations. The relation "a_1 and a_2 can share an AR" is normally static so that it can be analyzed at compile time. It induces a partitioning of the set of array references in a loop into disjoint groups. We now consider optimized address computation for each of these groups separately, that is, we focus on loops of the form

```
for (i = N1; i <= N2; i += S)
{ array reference a_1
  array reference a_2
   ...
  array reference a_n
}
```

Each array reference $a_j = A[i+d]$ is characterized exactly by its integer offset value $f(a_i) = d$. As an example, consider the following reference pattern with respect to some array A.

```
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  */
}
```

A naive approach to obtain a zero-cost solution is to allocate a separate register R_i for each reference a_i. Before the loop is executed, each register R_i is initialized with the address of the first array element that will be accessed by reference a_i in the loop. Within the loop body, only a zero-cost address computation (auto-increment) on R_i is required in order to generate the address for a_i in the next loop iteration. However, this solution consumes many registers. Now consider another extreme solution using only a single AR R1. This would lead to the following addressing scheme for the above example, given in a C-like notation:

```
R1 = &A[3]  /* initialize R1 with &A[2+1] */
for (i = 2; i <= N; i++)
{ /* a_1 */   *R1 --     /* access A[i+1] */
  /* a_2 */   *R1 += 2   /* access A[i]   */
  /* a_3 */   *R1 -= 3   /* access A[i+2] */
  /* a_4 */   *R1 += 2   /* access A[i-1] */
  /* a_5 */   *R1 --     /* access A[i+1] */
  /* a_6 */   *R1 -= 2   /* access A[i]   */
  /* a_7 */   *R1 += 4   /* access A[i-2] */
}
```

All array references share the same AR, and except for the initialization, each array reference is computed by a post-modify operation on R1. In the last control step (7), the address for the first reference in the next loop iteration needs to be computed. One obtains a solution with a cost of 5, because there are 5 unit-cost and only 2 zero-cost (auto-decrement) address computations. This implies an overhead of 5 extra instructions for address computation *in each loop iteration.*

In order to minimize such overhead, code optimization techniques are required which read an array reference pattern of a loop body and compute good address generation schemes. This can be done based on the graph model specified as follows:

Definition: Let $(a_1, \ldots, a_n)$ be a sequence of array references in a loop. For all a_i, a_j, with $1 \leq i < j \leq n$,

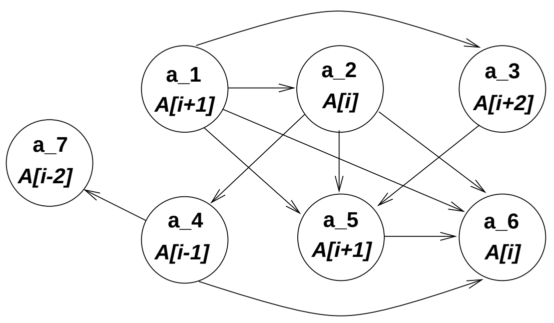

Fig. 2. *Distance graph for the example loop*

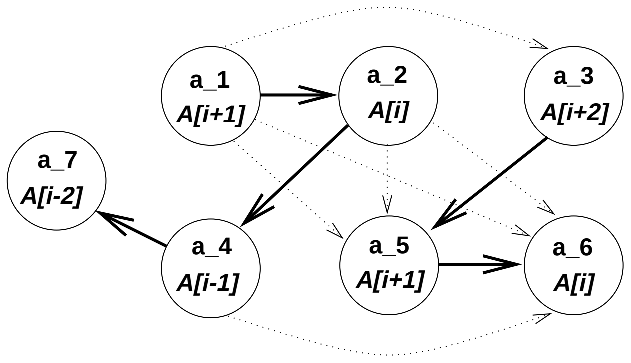

Fig. 3. *Solution computed by the matching-based algorithm (selected edges in boldface)*

the **intra-iteration distance** $\delta(a_i, a_j) := f(a_j) - f(a_i)$ 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 AR 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, k_j with $i < j$ 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.

In summary, minimizing the number of ARs required for a zero-cost solution is equivalent to solving the following problem.

Definition: Let $G = (V, E)$ with $V = \{a_1, \ldots, a_n\}$ be the distance graph of a loop, and let M be the maximum modify range. The problem of **AR allocation** is to find a minimum path cover of 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 $|\delta'(a_{k_1}, a_{k_m})| \leq M$.

III. Related work

There is a large amount of results on general compiler optimization techniques aiming at maximizing instruction-level parallelism within loops. However, DSP-specific loop optimizations are still few. Nicolau et al. [6] have proposed assignment algorithms for data registers in loops without emphasis on memory address generation. Bartley [7] has proposed an algorithm that optimizes address computations for *scalar* variables. His work has later been improved and generalized [8, 9, 10].

Liem [11] has implemented a tool for a variant of the AR allocation problem for arrays. Unfortunately, no concrete algorithms have been published, but his results indicate that optimizing array index computation yields significant improvements in code size (up to 30 %) and speed (up to 60 %) for C programs, as compared to non-optimized array addressing.

IV. The matching-based algorithm

In [12] is has been proposed to apply a matching-based algorithm developed in the area of graph theory [13] to AR 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.

The matching-based algorithm works as follows. Given a distance graph $G = (V, E)$, a bipartite graph $G' = (V', E')$ is built. For each $v \in V$, its node set V' contains two nodes v' and $\bar{v}$. For all $(u, v) \in E$, the edge set E' contains $(\bar{u}, v')$. On G' a *maximum cardinality matching* $Z \subseteq E$ is computed. Z is a maximum set of non-incident edges in E. Since G' is bipartite, computation of Z can be performed in $\mathcal{O}(|E'| \cdot \sqrt{|V'|})$.

It is shown in [13] that an edge $e = (u, v)$ of the original graph G is part of an optimum path cover, if and only if the corresponding edge $e' = (\bar{u}, v')$ is contained in the matching Z. Fig. 3 shows the result of applying the matching-based algorithm to the example from fig. 2. The number of allocated ARs can be easily computed from G, G', and Z: Remove all edges $e = (u, v)$ from E, for which the corresponding edge $e' = (\bar{u}, v')$ is not contained in the matching Z. Then, the number of registers is equal to the number of connected components (i.e. node-disjoint paths) in G. This number can be determined in

$\mathcal{O}(|V| + |E|)$.
Using the matching-based algorithm, we obtain the following address generation scheme with two ARs:

```
R1 = &A[3]   /* initialize R1 with &A[2+1] */
R2 = &A[4]   /* initialize R2 with &A[2+2] */
for (i = 2; i <= N; i++)
{  /* a_1 */    *R1 --    /* access A[i+1] */
   /* a_2 */    *R1 --    /* access A[i]   */
   /* a_3 */    *R2 --    /* access A[i+2] */
   /* a_4 */    *R1 --    /* access A[i-1] */
   /* a_5 */    *R2 --    /* access A[i+1] */
   /* a_6 */    *R2 += 3  /* access A[i]   */
   /* a_7 */    *R1 += 4  /* access A[i-2] */
}
```

Since the matching-based algorithm neglects inter-iteration distances, two unit-cost address computations on R1 and R2 must be executed at the end of each iteration. The algorithm presented in the following section guarantees to avoid such overhead at the expense of additional registers.

V. The path-based algorithm

If address computations between subsequent loop iterations are neglected, then the matching-based algorithm computes an optimal solution in polynomial time. 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 the problem becomes more complex. This can be seen by including inter-iteration distances in the distance graph model:

Definition: Let $G = (V, E)$ with $V = \{a_1, \ldots, a_n\}$ be the distance graph of a loop. The **extended 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

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

Presence of an edge $e = (a_j, a'_i)$ in E' indicates, that if the references a_i and a_j share an AR 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 ARs is equivalent to covering all nodes $\{a_1, \ldots, a_n\}$ in the extended 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. 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 [14] the AR allocation problem is (most likely) of exponential complexity.

One can compute a (potentially suboptimal) solution efficiently by the following path-based heuristic, a variant of which has been described in [15].

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 $e \in E'$.

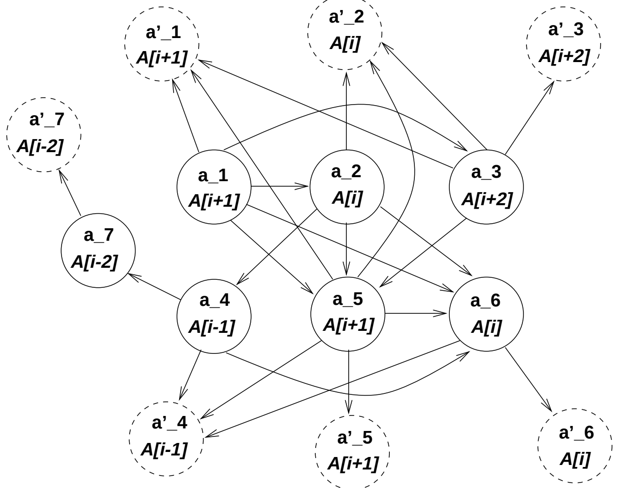

Fig. 4. *Extended distance graph*

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 AR 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.

Computation of longest paths takes $\mathcal{O}(|V'| \cdot |E'|)$. In the worst case, each execution of step 3 removes only a single node pair (a_i, a'_i) from G'. Therefore, the runtime of the algorithm is bounded by $\mathcal{O}(|V'|^2 \cdot |E'|)$. Fig. 4 shows the extended distance graph for our example loop. The path-based algorithm produces a zero-cost solution with four ARs:

```
R1 = &A[3]  /* initialize R1 with &A[2+1] */
R2 = &A[1]  /* initialize R1 with &A[2-1] */
R3 = &A[4]  /* initialize R1 with &A[2+2] */
R4 = &A[0]  /* initialize R1 with &A[2-2] */
for (i = 2; i <= N; i++)
{  /* a_1 */    *R1 --   /* access A[i+1] */
   /* a_2 */    *R1 ++   /* access A[i]   */
   /* a_3 */    *R3 ++   /* access A[i+2] */
   /* a_4 */    *R2 ++   /* access A[i-1] */
   /* a_5 */    *R1 ++   /* access A[i+1] */
   /* a_6 */    *R2      /* access A[i]   */
   /* a_7 */    *R4 ++   /* access A[i-2] */
}
```

VI. The branch-and-bound algorithm

The matching-based algorithm computes an *optimal* number of registers for a *relaxed* problem definition, and thus gives a lower bound on the minimum number of ARs. The path-based algorithm emits a potentially *sub-optimal* number of ARs, but guarantees a *zero-cost solution*. Therefore, it provides an upper bound. Using these bounds one can decide, whether or not a certain edge e of the distance graph must be included in the minimum path cover. An optimum solution to the AR allocation problem is therefore obtained by the following branch-and-bound algorithm:

algorithm MinRegs
input: distance graph $G = (V, E)$
output: minimal number of ARs

1. If $E = \emptyset$, then stop and return $|V|$.
2. Call the matching-based algorithm for G, which returns a number r_M of ARs. If the matching-based algorithm (by coincidence) yields a zero-cost solution, then stop and return r_M because in this case r_M is the minimum number of registers.
3. Compute an initial upper bound U on the minimum number of registers by the path-based algorithm.
4. Select a *feasible* edge $e = (a_i, a_j) \in E$. An edge $e = (a_i, a_j)$ is feasible, if and only if there exist a path $(a_j, \ldots, a_i')$ in the *extended distance graph* for G. Only feasible edges can lead to a zero-cost solution. If there are no more feasible edges, then stop and return U.
5. Construct a new distance graph $G_{\bar{e}} = (V, E - \{e\})$ which results from G by deleting edge e from E. $G_{\bar{e}}$ represents the problem if edge e is explicitly *excluded* from the path cover.
6. Construct a new distance graph $G_e = (V', E')$, so that $V' = V - \{a_j\}$ and
$$\begin{array}{rcl} E & = & ((E - \{(a_i, a_k) \in E \quad | \quad a_k \in V\}) \\ & - & \{(a_k, a_j) \in E \quad | \quad a_k \in V\}) \\ & \cup & \{(a_i, a_k) \quad | \quad (a_j, a_k) \in E\} \end{array}$$
This construction corresponds to merging the nodes a_i and a_j, so that G_e represents the problem if e is explicitly *included* in the path cover.
7. Compute lower bounds $L_{\bar{e}}$ for $G_{\bar{e}}$ and L_e for G_e by the matching-based algorithm.
8. If $L_{\bar{e}} > U$ then all solutions for $G_{\bar{e}}$ are suboptimal, and edge e must be selected. Return **MinRegs**(G_e).
9. If $L_e > U$ then all solutions for G_e are suboptimal, and e must be discarded. Return **MinRegs**$(G_{\bar{e}})$.
10. If $L_e \leq L_{\bar{e}}$ then compute $r_e :=$ **MinRegs**(G_e). If $r_e < U$ then set $U := r_e$.
Compute $r_{\bar{e}} :=$ **MinRegs**$(G_{\bar{e}})$.
Otherwise ($L_e > L_{\bar{e}}$) compute $r_{\bar{e}} :=$ **MinRegs**$(G_{\bar{e}})$. If $r_{\bar{e}} < U$ then set $U := r_{\bar{e}}$.
Compute $r_e :=$ **MinRegs**(G_e).
11. If $r_e < r_{\bar{e}}$, then select e, otherwise discard e. Return $\min(r_e, r_{\bar{e}})$.

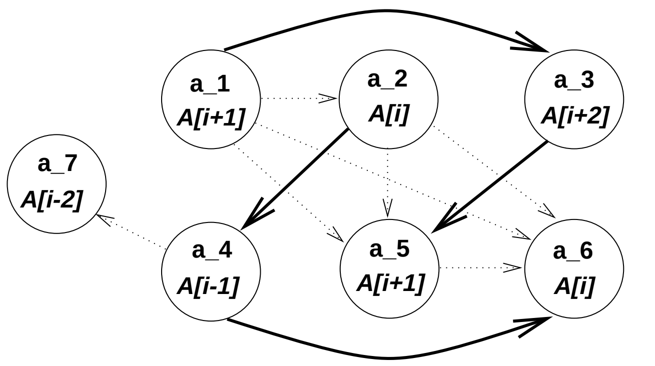

Fig. 5. *Optimal solution computed by the branch-and-bound algorithm (selected edges in boldface)*

The worst case runtime of this branch-and-bound algorithm is exponential in $|E|$. However, pruning the search space in steps 7 – 9 guarantees much lower runtime in most cases. Applying this algorithm to our example loop yields the following zero-cost solution with only three registers (fig. 5):

```
R1 = &A[3]  /* initialize R1 with &A[2+1] */
R2 = &A[2]  /* initialize R2 with &A[2+0] */
R3 = &A[0]  /* initialize R3 with &A[2-2] */
for (i = 2; i <= N; i++)
{  /* a_1 */    *R1 --   /* access A[i+1] */
   /* a_2 */    *R2 --   /* access A[i]   */
   /* a_3 */    *R1 --   /* access A[i+2] */
   /* a_4 */    *R2 ++   /* access A[i-1] */
   /* a_5 */    *R1 ++   /* access A[i+1] */
   /* a_6 */    *R2 ++   /* access A[i]   */
   /* a_7 */    *R3 ++   /* access A[i-2] */
}
```

VII. Experimental results

In order to obtain reliable results on the average performance of the heuristics, we have performed a statistical analysis. The performance of the algorithms is influenced by three parameters: the array reference sequence length n, the maximum modify range M, and the maximum offset difference D between all pairs of array references, so that each offset value $f(a_i)$ is in $[-D/2, D/2]$. We have considered the parameter sets $n \in \{5, 10, 15, 20, 25\}$, $M \in \{1, 3, 7, 15\}$ and $D \in \{4, 8, 16, 32\}$. For each parameter combination, we have run the algorithms on 100 array reference sequences, where the offset value for each array reference was given as a random number in $[-D/2, D/2]$.

The results are listed in table I. Each line in the table refers to a fixed setting of one of the parameters n, M, or D, while varying the other two parameters. Column 2 gives the average number of ARs computed by the matching-based algorithm. The average number of unit-cost address computations incurred by this algorithm is given in column 3. Columns 4 and 5 show the average number of registers computed by the path-based and the optimal branch-and-bound procedure, respectively. Column 6 shows the average computation time (in SPARC-20 CPU seconds) consumed by the branch-and-bound proce-

TABLE I
Experimental results

parameter	# registers matching-based	# unit-cost address comp.	# registers path-based	# registers B&B	CPU sec B&B	overhead path-based
$n = 5$	2.05	0.18	2.18	2.15	< 0.01	1.39 %
$n = 10$	2.65	0.36	2.87	2.79	0.01	3.19 %
$n = 15$	2.94	0.54	3.27	3.11	0.13	5.10 %
$n = 20$	3.18	0.66	3.58	3.36	2.63	6.84 %
$n = 25$	3.29	0.79	3.81	3.49	31.01	9.06 %
$D = 4$	1.54	0.21	1.65	1.56	5.48	5.37 %
$D = 8$	2.07	0.39	2.30	2.17	14.78	5.94 %
$D = 16$	3.05	0.61	3.44	3.24	4.45	6.23 %
$D = 32$	4.63	0.81	5.18	4.94	2.30	4.93 %
$M = 1$	5.11	1.09	5.75	5.63	15.89	2.18 %
$M = 3$	3.08	0.53	3.41	3.17	10.29	7.61 %
$M = 7$	1.84	0.28	2.05	1.85	0.73	10.49 %
$M = 15$	1.25	0.12	1.36	1.26	0.11	7.91 %
average						5.86 %

dure. The measured CPU times both for the matching-based and the path-based algorithm are always less than 10 ms, and can thus be neglected. Finally, column 7 shows the average overhead in percent of the path-based heuristic as compared to the optimum number of registers.

The matching-based heuristic performs better than expected with respect to the incurred unit-cost address computations: For almost all parameter settings, the average incurred cost is less than 1, so that the code size penalty is very low. However, as array references within loops may be executed thousands of times, the program speed penalty is still significant. This penalty is avoided by the path-based algorithm which consumes more registers. However, comparing its results to the optimum solutions shows that the path-based algorithm performs very well in most cases, with an average overhead of less than 6 %. The branch-and-bound algorithm always produces optimal results, but an acceptable computation time is probably exceeded beyond $n = 20$. For realistic problems, however, this is a relatively large number of array references.

VIII. Conclusions

In this paper, we have discussed algorithms for DSP address register allocation, which aim at minimizing code size and speed overhead caused by address computation for array references in loops. We have shown how two earlier heuristic algorithms can be embedded into a branch-and-bound procedure which avoids any overhead and emits the minimal number of registers. Our experimental evaluation indicates that it is fast enough for most practical problems. For very large problems, it may be substituted by the proposed path-based algorithm. This heuristic was shown to be efficient and to produce close-to-optimum results. The algorithms are easy to implement and can be immediately used as subroutines in DSP compilers. Future work could deal with generalizing the algorithms for nested loops.

References

[1] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[2] 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

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

[4] A.V. Aho, R. Sethi, J.D. Ullman: *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986

[5] A. Basu, R. Leupers, P. Marwedel: *Register-Constrained Address Computation in DSP Programs*, poster presentation, Design Automation & Test in Europe (DATE), 1998

[6] D.J. Kolson, A. Nicolau, N. Dutt, K. Kennedy: *Optimal Register Assignment to Loops for Embedded Code Generation*, 8th Int. Symp. on System Synthesis (ISSS), 1995

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

[8] 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

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

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

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

[12] 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

[13] 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

[14] 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

[15] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997