Algorithms for Address Assignment in DSP Code Generation

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

Abstract– *This paper presents DSP code optimization techniques, which originate from dedicated memory address generation hardware. We define a generic model of DSP address generation units. Based on this model, we present efficient heuristics for computing memory layouts for program variables, which optimize utilization of parallel address generation units. Improvements and generalizations of previous work are described, and the efficacy of the proposed algorithms is demonstrated through experimental evaluation.*

1 Introduction & related work

Design of embedded VLSI systems in form of heterogeneous single-chip architectures, comprising both hardware and software components, raises new demands on electronic CAD tools. Among the most challenging ones is *code generation for embedded DSPs*: Limited area for on-chip program ROMs as well as real-time constraints demand for generation of extremely compact code, while high compilation speed is no longer a primary goal. However, current DSP compiler technology is still too immature to replace assembly-level programming. Code quality overheads up to several hundred percent compared to hand-crafted code have been reported [1], which are unacceptable in most cases. It has been concluded, that development of more efficient DSP compilers demands for extension of classical compiler technology by novel and thorough optimization techniques [2], in order to eliminate the current bottleneck in DSP software development.

A new means of advanced code optimization is *address assignment*, i.e. optimization of memory layout for program variables. Address assignment has hardly been an issue in classical compilers. In fact, most general-purpose compilers perform "naive" address assignment, i.e. program variables are mapped to memory cells according to the lexicographic or declaration order of identifiers. DSPs provide dedicated address generation units (AGUs) for parallel next-address computations. Address generation on AGUs does not employ datapath resources, and thus leads to higher instruction-level parallelism. Essentially, this is achieved by auto-increment capabilities of AGUs. However, high utilization of auto-increment addressing demands for appropriate placement of program variables in memory. When the exact sequence of variable accesses is known after code generation, address assignment can be performed as a separate code optimization phase.

Address assignment for AGUs with a *single* address register (AR) (the *simple offset assignment* problem, SOA) was first studied by Bartley [3] and Liao [4]. Liao also proposed a generalization, which handles any fixed number k of address registers (*general offset assignment* problem, GOA), but requires external parameters for each problem instance. Complementary to [3], [4] is the contribution by Liem [6], who describes methods for efficient address computation for *array accesses*, based on strength reduction of array index computations. As in [3],[4], in this paper we concentrate on address assignment for basic blocks, and for *scalar* data, e.g. scalar variables or spilled values. The organization of the paper is as follows: In Section II, we examine address generation in DSPs, and we define a generic AGU model, which captures a subset of addressing capabilities of many contemporary DSPs. Based on this model, we describe effective address assignment algorithms: Section III proposes a heuristic improvement of Liao's SOA algorithm. In Section IV, an improved GOA algorithm is presented, which also needs no external parameters. Section V focuses on utilization of *modify registers* in AGUs, and conclusions are given in Section VI.

2 Address generation in DSPs

Address generation hardware in DSPs differs from that of standard processors. Usually, several address registers (ARs) are available, which can be updated in parallel to other machine operations, thereby introducing no code size or speed overhead. On the other hand, addressing may be quite restricted: In order to avoid long combinational delay, many DSPs do not permit indexing with an offset, but only *post-modification*, i.e. additions or subtractions involving ARs take place only at the end of a machine cycle.

Besides high code quality, also *retargetability* is a primary goal in DSP code generation [2], due to the growing diversity of DSPs in form of application-specific designs (ASIPs). Therefore, we consider a *generic* AGU architecture, which reflects a subset of AGU capabilities of many contemporary DSPs. Our AGU model (fig. 1) is parameterized by the number k of *address registers* (ARs) and the number m of *modify registers* (MRs). In case of multiple memory banks we assume separate AGUs for each bank. Typical AGU configurations are $k = 4, m = 4$ (ADSP-210x), or $k = 8, m = 1$ (TMS320C2x). The ARs provide effective memory addresses, while MRs store integer modify values for AR updates. AR and MR files are indexed by designated *pointers*, which select the *current* AR and MR for each machine cycle. AR and MR pointer updates usually does not contribute to code size. The AGU model permits execution of the following primitive *AGU operations* in each machine cycle:

immediate AR load: The current AR is loaded with an immediate value supplied by the instruction word.

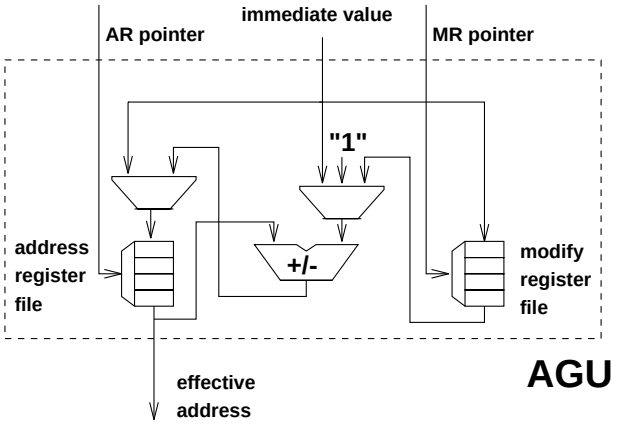

Fig. 1. *Generic AGU model*

immediate AR modify: An immediate value is added to or subtracted from the current AR.
auto-increment/decrement: The constant 1 is added to or subtracted from the current AR.
immediate MR load: The current MR is loaded with an immediate value.
auto-modify: The contents of the current MR are added to or subtracted from the current AR.

The optimization potential of address assignment is induced by the *cost metric* for these operations. The instruction encodings of immediate loads and modifys tend to occupy a large portion of the total instruction-word length. Therefore, these operations usually consume an extra instruction word. In contrast, auto-increment/decrement and auto-modify operations only employ AGU resources, and can thus be executed in parallel to other operations. Thus, we assign *zero cost* to "auto" operations and *unit cost* to AGU operations involving immediate values. The goal of address assignment is maximum usage of zero-cost AGU operations. In order to outline the effect of address assignment, consider a basic block, in which the variable set $V = \{a, b, c, d\}$ is referenced in the sequence

$$S = (b, d, a, c, d, a, c, b, a, d, a, c, d)$$

Fig. 2 a) shows a naive memory layout, with variables placed in lexicographic order, and the corresponding sequence of AGU operations, assuming only one AR is available. The total addressing cost is 9. This cost can be reduced to 3, as shown in fig. 2 b). Using a better variable permutation, most addresses can be generated by zero-cost operations. Modify value 2, which has three occurrences in the sequence, can be assigned to a modify register MR in order to achieve further cost reduction. The next sections describe algorithms which compute such cheap AGU operation sequences.

3 Simple offset assignment

Given a variable set V and a variable access sequence S of a basic block, *simple offset assignment* (SOA) is the problem of finding a permutation

$$\pi : V \rightarrow \{0, \ldots, |V| - 1\}$$

of variables to memory addresses, which minimizes the total addressing costs in presence of a *single* AR. Though SOA is an over-simplified problem, SOA algorithms can be employed as subroutines for more general

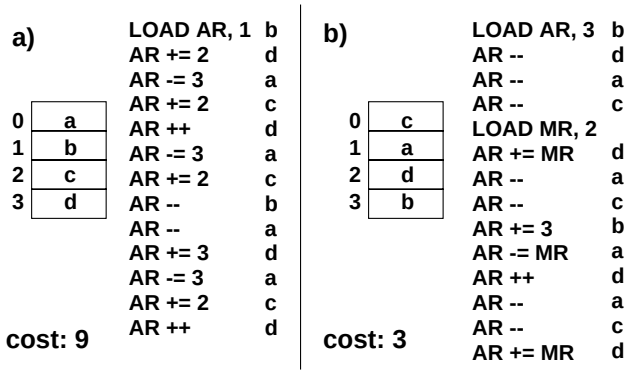

Fig. 2. *Address assignment: a) naive memory layout, b) improved layout, with usage of modify register MR. ++/– – denote auto-increment/decrement, +=/-= denote immediate and auto-modifys.*

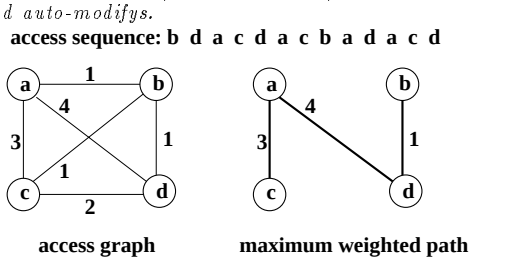

Fig. 3. *Access sequence, access graph, maximum weighted path*

problems. Liao [4] models SOA by an undirected, edge-weighted *access graph* $AG = (V, E)$. Nodes in AG correspond to variables while edges represent the frequency of *access transitions* between each pair of variables in a given access sequence (fig. 3). Obviously, nodes connected by heavy edges should be placed into consecutive memory cells, in order to enable zero-cost addressing. As shown in [4], SOA is equivalent to constructing a *maximum weighted path*, touching each node in AG exactly once. Since the variable pairs with the highest transition frequencies have to be assigned to consecutive addresses, traversing such a Hamiltonian path induces an optimum address assignment. The *cost* of the assignment is equal to the accumulated weight of AG edges *not* contained in the path. Due to NP-hardness of SOA, [4] describes a heuristic procedure, which resembles Kruskal's maximum spanning tree algorithm for graphs:

1) Sort the edges in E in descending order of weight, and begin with an empty path P.
2) While P is not a Hamiltonian path: Select the next *valid* edge e with highest weight, and add e to P. An edge is valid, if it causes neither cycles nor trees in P.

The sorting procedure in step 1 does not prescribe ordering of edges with equal weights. Exploiting AG information, we can heuristically break such ties, so as to further reduce SOA costs: For an access graph $AG = (V, E, w)$, the **tie-break function** $T : E \rightarrow \mathbf{N}_0$ is defined by

$$T(e) = \sum_{e' \in E} w(e'), \quad \text{with} \quad e \cap e' \neq \emptyset$$

Intuitively, giving priority to edges with high T value appears to be a good choice. However, the opposite is true: A high $T(e)$ value indicates high relative probabil-

$\|V\|$	$\|S\|$	naive	Liao	%	with T	%
5	10	4.85	2.22	46	2.18	45
5	20	8.65	5.30	61	5.26	61
15	20	13.38	6.59	49	6.09	46
10	50	30.86	21.80	71	21.30	69
40	50	37.82	19.56	52	17.87	47
10	100	60.86	48.04	79	47.73	78
50	100	76.41	46.91	61	43.78	57
80	100	77.67	40.72	52	36.86	47
100	200	156.61	98.74	63	91.04	58

TABLE I
COMPARISON OF SOA ALGORITHMS

ity, that including neighboring edges of e in the Hamiltonian path would be favorable. Selection of e however restricts the number of valid edges in its neighborhood. Thus, if $w(e_1) = w(e_2)$, we give priority to e_1, exactly if $T(e_1) < T(e_2)$.

The efficacy of tie-breaking is demonstrated in table I. In order to permit unbiased comparison of techniques, we performed experiments on 100 random access sequences. Columns 1 and 2 show the problem parameters ($|V|$ = number of different variables, $|S|$ = access sequence length). The average cost of naive address assignment (100 %) is given in column 3. The absolute and relative costs achieved by Liao's algorithm are listed in columns 4 and 5, while columns 6 and 7 show the corresponding data when using the additional tie-break heuristic. On average, tie-breaking saves additional 3 % addressing cost, at virtually no increase in computation time, which – even for large SOA instances – is in the range of milliseconds on a SPARC-10.

4 General offset assignment

Higher AGU utilization than in SOA can be achieved by using all available ARs, which leads to the problem of *general offset assignment*. GOA is the generalization of SOA towards an arbitrary number k of ARs. A reasonable heuristic approach is to compute a partitioning

$$P : V \rightarrow \{V_1, \ldots, V_k\}$$

of variable set V into k disjoint subsets, for each of which a separate AR is used. Each AR[i] addresses the *subsequence* $S(V_i)$, i.e. the subsequence of S exclusively referencing elements of V_i. In this way, GOA is reduced to k SOA problems.

In [4], the following partitioning heuristic was proposed: Starting with an access sequence S on variable set V and k ARs, a variable subset $V' \subset V$ is determined. Then, the subsequences $S(V')$ and $S(V \backslash V')$ and the corresponding access graphs are constructed. If the sum of SOA costs for $S(V')$ and $S(V \backslash V')$ is lower than the SOA cost for the original sequence S, then subset V' is "accepted", i.e. is assigned to the k-th AR, and the algorithm is called recursively for $S(V \backslash V')$, V', and $k-1$. Otherwise the algorithm terminates. If $k = 1$, then all remaining variables are assigned to one AR, and recursion stops.

Unfortunately, no general procedure was described for determining the subsets V', which is a crucial step of the procedure. Instead, it was suggested to select subsets of fixed size s (typically 2 to 6), based on a certain edge-weight criterion. However, this leads to badly

```
algorithm SolveGOA(V, S, k)
begin
  AG = (V, E, w) := access graph for S;
  L := sorted list of non-zero edges in E
    in descending order of weight;
  V_1, ..., V_k := ∅;
  i := 0;
  repeat
    i := i + 1;
    {u, v} := next edge in L with u, v ∉ V_1 ∪ ... ∪ V_k;
    V_i := {u, v};
  until (i = k) or ({u, v} = ∅);
  l := i;
  for all v ∈ V, v ∉ V_1 ∪ ... ∪ V_l do
    V* := the element of {V_1, ..., V_l}, such that
          SOA_cost(S(V* ∪ {v})) − SOA_cost(S(V*)) → min;
    V* := V* ∪ {v};
  end for
  return SolveSOA(S(V_1)) ∘ ... ∘ SolveSOA(S(V_l));
end algorithm
```

Fig. 4. *Algorithm for General Offset Assignment*

$\|V\|$	$\|S\|$	k	Liao	SolveGOA	gain (%)
10	50	2	13.14	11.93	9
10	50	4	9.30	5.02	46
40	50	4	10.40	7.65	26
40	50	8	8.14	8.22	-1
10	100	4	17.16	8.45	51
10	100	8	17.29	5.00	71
50	100	8	21.34	12.69	41
80	100	8	17.80	11.30	37
100	200	8	59.61	37.84	37

TABLE II
COMPARISON OF GOA ALGORITHMS

balanced partionings with $k-1$ subsets of size s, and one large subset of size $|V| - s \cdot k$. Moreover, Liao observed that the optimal s value strongly depends on the given access sequence, so that manual parameterization for each problem instance is required.

Our GOA algorithm (fig. 4) avoids these problems, and determines an *individual* variable subset size for each AR, using SOA as a subroutine for cost estimations. We start with l, $l \leq k$, subsets $V_1, \ldots, V_l$ of size two, by selecting the l disjoint edges of highest weight in the access graph and assigning the corresponding node pairs to one subset each. If the access graph contains k or more disjoint non-zero edges, then $l = k$. Otherwise, all of these edges are taken, and $l < k$. The initial SOA cost for each subset is one, due to AR initialization. Then, all remaining variables v are consecutively assigned to the subset V^*, for which the increase in SOA cost caused by adding v to V^* is minimal. After all variables have been assigned, the solution is obtained by concatenating l SOA solutions. Statistical performance results for different V, S, k configurations are listed in table II. Columns 4 and 5 show GOA cost values achieved by Liao's procedure (with $s = 3$) and algorithm SolveGOA, respectively. In total, we observed that SolveGOA outperforms Liao's algorithm by 22 % on average. Both algorithms require comparable CPU times (up to a few seconds on a SPARC-10).

5 Modify registers

Further reduction of addressing costs is possible by assignment of multiply required modify values to MRs

(cf. fig. 2 b) in a post-pass phase, because, as argued in Section II, retrieving modify values from MRs causes zero cost. On AGU operation sequences computed by SolveGOA, optimal utilization of m MRs can be obtained efficiently: Let $U = (u_1, \ldots, u_n)$ denote the sequence of required modify values, i.e. the sequence of immediate values that occur in AGU operations of type "immediate AR modify". For the example in fig. 2 a), $U = (2, -3, 2, -3, 2, 3, -3, 2)$. If only one MR is available, it must be decided, whether a certain u_i should be kept as an immediate value or should be implemented by "immediate MR load" and "auto-modify" operations, so as to maximize potential reuse of MR contents for later occurences of u_i in U. The optimal choice can be made sequentially for each u_i (starting with u_1) by inspecting the current MR contents $c(MR)$ (initially undefined) and the rest sequence $U_i = (u_{i+1}, \ldots, u_n)$. Whenever $u_i = c(MR)$, then u_i can be generated by "auto-modify", which saves one cost unit compared to the original AGU operation sequence. Otherwise, there is a choice to either keep $c(MR)$ in MR, or to overwrite MR by u_i. For any value x, let $next(x)$ denote the index of the next occurrence of u in U_i (∞ in case of no further occurrence). If $next(c(MR)) = \infty$ or $c(MR)$ is undefined, then $c(MR)$ cannot be reused, so that overwriting MR causes no disadvantage, and MR is loaded with u_i, if $next(u_i) < \infty$. For $next(c(MR)) < \infty$ and $next(u_i) < \infty$, the better choice depends on whether $c(MR)$ or u_i recurs first in U_i. If $next(c(MR)) < next(u_i)$, reusability of u_i implies reusability of $c(MR)$. Conversely, if $c(MR)$ cannot be reused because MR is overwritten again before $next(c(MR))$, then also u_i cannot be reused. Therefore, keeping $c(MR)$ in MR is at least as good as overwriting MR by u_i. If $next(c(MR)) > next(u_i)$, then a dual argumentation shows that loading MR with u_i is the better alternative.

In case of $m > 1$ available MRs, it must be additionally decided, *which* MR should be overwritten. It is easily seen, that this corresponds to the problem of *minimizing memory page faults* in demand-paging operating systems, i.e. which page frame should be overwritten in case a memory page not present in main memory is referenced. Since the sequence U is known in advance, one can apply Belady's optimum replacement algorithm [7]. Belady showed, that if the complete page access sequence is known, overwriting the frame containing the page with the *largest forward access distance* is optimal. Analogously, in case a new modify value has to be loaded into the MR file, the register MR* must be selected, for which $next(c(MR^*))$ is maximal. Whenever assigning a modify value to an MR is favorable, MR* is guaranteed to be the optimal one. This permits to combine the above criteria and Belady's algorithm to obtain optimal utilization of m MRs for a previously generated sequence of AGU operations. Table III gives experimental results for different $|U|$ and m values. Column 3 shows the average cost reduction achieved by applying MR optimization on AGU operation sequences generated by SolveGOA (cf. Section III). The results show that the efficacy of MR optimization depends on both the problem size and the number of MRs. For small $|U|$, the optimization potential of MR usage is low, and an increase in m does not lead to further gains, because of a "saturation" effect. For larger sequences, the cost reduction becomes significant and also grows with m. The runtime complexity of MR optimization is $\mathcal{O}(m \cdot |U|^2)$. In practice, however, runtimes linear in $|U|$ can be observed, because the number of occurrences of identical modify values in U is mostly bounded by a constant $c \ll |U|$.

$\lvert U\rvert$	m	% cost reduction
20	2	2
20	4	2
50	2	3
50	4	4
100	2	20
100	4	22
100	8	29
300	4	33
300	8	48

TABLE III
Experimental results of MR optimization

Compared to naive address assignment, SolveGOA with post-pass MR optimization often achieves 80–90 % reduction on the number of instructions for address generation. According to [5], these instructions may constitute about 20 % of total machine code size. Hence, address assignment may provide a signficant increase in code quality at low computational effort.

6 Conclusions

Generation of efficient machine code for embedded DSPs demands for new optimization techniques. In this paper, we have presented algorithms, which yield high utilization of parallel AGUs by computing appropriate memory layouts for program variables, and which immediately apply to contemporary DSPs. Further improvements can be expected from algorithms for more general definitions of address assignment problems, and direct incorporation of modify registers into address assignment.

Acknowledgements
The authors would like to thank Bernhard Wess from TU Vienna for helpful comments. This work has been supported by ESPRIT project 9138 (CHIPS).

References

[1] V. Zivojnovic, J.M. Velarde, C. Schläger: *DSPStone – A DSP-oriented Benchmarking Methodology*, Technical Report, Dept. of EE, University of Aachen, Germany, 1994

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

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

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

[5] S. Liao: *Code Generation and Optimization for Embedded Digital Signal Processors*, PhD thesis, Dept. of EE and CS, MIT, 1996

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

[7] L.A. Belady: *A Study of Replacement Algorithms for a Virtual-Storage Computer*, IBM System Journal 5(2): pp. 78-101, 1966