A Uniform Optimization Technique for Offset Assignment Problems

Rainer Leupers, Fabian David

University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany
email: leupers@ls12.cs.uni-dortmund.de

Abstract– A number of different algorithms for optimized offset assignment in DSP code generation have been developed recently. These algorithms aim at constructing a layout of local variables in memory, such that the addresses of variables can be computed efficiently in most cases. This is achieved by maximizing the use of auto-increment operations on address registers. However, the algorithms published in previous work only consider special cases of offset assignment problems, characterized by fixed parameters such as register file sizes and auto-increment ranges. In contrast, this paper presents a genetic optimization technique capable of simultaneously handling arbitrary register file sizes and auto-increment ranges. Moreover, this technique is the first that integrates the allocation of modify registers into offset assignment. Experimental evaluation indicates a significant improvement in the quality of constructed offset assignments, as compared to previous work[1].

1 Introduction

One architectural feature typical for DSPs is the special hardware for memory address computation. DSPs are equipped with address generation units (AGUs), capable of performing indirect address computations in parallel to the execution of other machine instructions. In fact, the AGU architectures of many DSPs, such as the TI C25/C50/C80, the Motorola 56k family, the Analog Devices ADSP-210x, or the AMS Gepard DSP core, are very similar and, for support of indirect addressing, differ mainly in the following parameters:

- The number k of *address registers* (ARs). ARs store the effective addresses of variables in memory and can be updated by load and modify (i.e., adding or subtracting a constant) operations.
- The number m of *modify registers* (MRs). MRs can be loaded with constants and are generally used to store frequently required AR modify values.
- The *auto-increment range* l.

Further differences in the detailed AGU architectures of DSPs are whether MR values are interpreted as *signed or unsigned* numbers, and whether ARs and MRs are *orthogonal*, i.e., whether each MR can be used to modify each AR.

Typically, such AGUs permit to execute two types of address arithmetic operations in parallel to other instructions:

1. *Auto-increment operations* of the form
   `AR[i] += d` or
   `AR[i] -= d`,
   where `AR[i]` is an AR, and $d \leq l$ is a constant less or equal to the auto-increment range.
2. *Auto-modify operations* of the form
   `AR[i] += MR[j]` or
   `AR[i] -= MR[j]`,
   where `AR[i]` is an AR, and `MR[j]` is an MR.

These two operations can be executed without any overhead in code size or speed. In contrast, loading an AR or MR, or modifying an AR by a constant larger than l always requires one extra machine instruction. Therefore, any high-level language compiler for DSPs should aim at maximizing the use of auto-increment and auto-modify operations for address computations, so as to maximize code speed and density. This is extremely urgent in embedded DSP systems with real-time constraints and limited silicon area for program memory.

One method to minimize the code needed for address computations is to perform *offset assignment* of local variables. Let v be a variable accessed at some point of time in a program, and let variable w be the immediate successor of v in the variable access sequence. Whether or not the address of w can be computed from the address of variable v by auto-increment obviously depends on the absolute difference of their addresses. Since the compiler determines the addresses ("offsets") of local variables in the stack frame of a function, the offsets can be chosen in such a way, that the offset distance of v and w is less or equal to l, so that auto-increment is applicable. A more precise description of offset assignment is given in section 2. Alternatively, an MR containing the offset distance may be exploited

[1]Publication: ISSS, Hsinchu/Taiwan, Dec 1998, ©IEEE.

to implement the address computation by auto-modify. Examples are given in section 4.

Since offset assignment requires detailed knowledge about the variable access sequences in a program, this optimization is typically executed as a separate compiler phase subsequent to code generation.

The purpose of this paper is to present a novel technique for solving offset assignment problems, which is more general than previous work and, as a consequence of this, yields better solutions. It is complementary to techniques, which optimize address computations for a *fixed* memory layout with application to global variables or array elements [1, 2, 3, 4]. The exact contributions of this paper are the following:

- In section 3 we present a genetic algorithm (GA) formulation for solving the offset assignment problem given an arbitrary number k of ARs and a fixed auto-increment range $l = 1$.
- In section 4, we extend the GA to incorporate an arbitrary number m of MRs into offset assignment. Utilization of MRs, which is treated separately from offset assignment in previous work, has a large impact on the quality of the constructed offset assignments.
- In section 5, we further generalize the GA, so as to optimize offset assignments for an arbitrary auto-increment range l.

In all three sections we provide experimental results that indicate the improvement achieved over previous techniques. Finally, section 6 gives conclusions.

2 Simple and general offset assignment

Bartley's algorithm [5] was the first that solved the *simple offset assignment* problem (SOA), which can be specified as follows.

Given an AGU with a single AR and an auto-increment range of 1, a set of variables $V = \{v_1, \ldots, v_n\}$ and an access sequence $S = (s_1, \ldots, s_m)$, with each s_i being a member of V, compute a bijective offset mapping $F : V \rightarrow \{1, \ldots, n\}$, such that the following cost function is minimized:

$$C(S) = \sum_{i=1}^{m-1} d(i)$$

$$d(i) = \begin{cases} 1, & \text{if } |F(s_i) - F(s_{i+1})| > 1 \\ 0, & \text{else} \end{cases}$$

Bartley proposed a graph-based heuristic to compute F based on information extracted from S, and showed that using SOA as a compiler optimization (instead of a straightforward offset assignment that neglects S) indeed gives a significant improvement in code quality.

Liao [6] proposed an alternative SOA algorithm, proved the NP-hardness of SOA, and provided a heuristic algorithm for the *general offset assignment* problem (GOA). GOA is the generalization of SOA towards an arbitrary number k of ARs. Liao pointed out that GOA can be solved by appropriately partitioning V into k subsets, thereby reducing GOA to k separate SOA problems. In [7, 8] improved SOA algorithms are described, while in [9] an improved variable partitioning heuristic for GOA has been given. However, these techniques only work for a fixed auto-increment range $l = 1$ and do not exploit modify registers.

3 Genetic optimization for GOA

Genetic algorithms (GAs) are a well-tried optimization technique imitating natural evolution to achieve good solutions (cf. [10] for an overview). GAs are particularly well-suited for nonlinear optimization problems, since they are capable of skipping local extrema in the objective ("fitness") function, and thus in general come close to optimal solutions. Among many other applications, GAs have been successfully applied to data path synthesis [11].

We have chosen GAs for solving offset assignment problems mainly due to two reasons: First, GAs are more robust than heuristics. If enough computation time is invested, then a GA most likely approximates a global optimum, while heuristics in many cases are trapped in a local optimum. Since very high compilation speed is not important for DSP compilers, GAs are more promising than heuristics, whenever a reasonable amount of time is not exceeded. Second, since offset assignment essentially demands for computing a good permutation of variables w.r.t. simple cost functions, offset assignment has a straightforward encoding as a GA.

In this section, we present our solution approach the GOA problem in the form of a GA. The same formulation is also used for the extensions described in sections 4 and 5, where only the fitness function is adapted, so as to solve generalized offset assignment problems.

3.1 Chromosomal representation

The chromosomal representation in our approach closely reflects the offset mapping F defined in section 2. Each variable in V is represented by its unique index in $\{1, \ldots, n\}$. Each gene in the chromosome represents one variable, and its position in the chromosome accounts for its offset. This representation can be immediately used for solving the SOA problem, since SOA essentially demands for a certain permutation of variables.

In order to solve the GOA problem, it must be additionally decided, which of the k available ARs will be used for accessing a certain variable. We accommodate this by using a modified chromosomal representation: In addition to the variable indices $\{1, \ldots, n\}$, the genes may assume a value in $\{n+1, \ldots, n+k-1\}$. In any step of the GA, each chromosome represents one permutation of the values $\{1, \ldots, n+k-1\}$. Indices larger than n serve as "separator symbols" in the chromosome. Starting with AR number 1, each separator represents a transition to the next higher address register, which is then used for all variables following in the chromosome

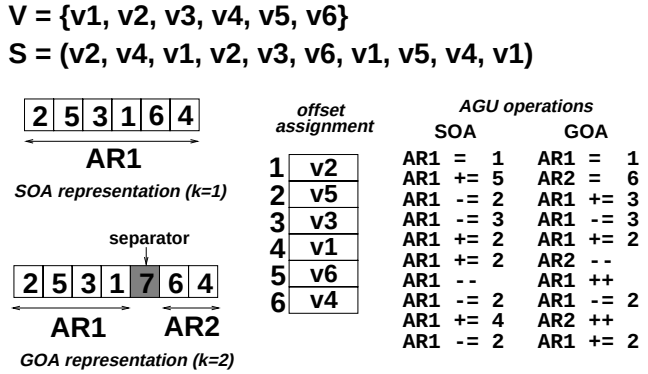

Figure 1: *Chromosomal representation for SOA and GOA*

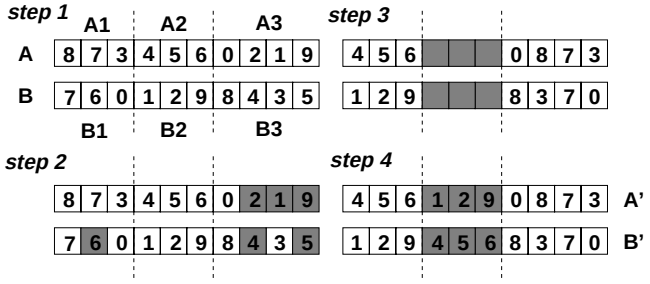

Figure 2: *Permutation preserving order crossover*

before the next separator (fig. 1). Since there are $k-1$ separators, one obtains an offset assignment for k ARs. The chromosomal representation is exemplified in fig. 1.

3.2 Mutation

Since any chromosome must represent a permutation of $\{1,\ldots,n+k-1\}$, mutation operators have to be *permutation preserving*, i.e., they must only generate new permutations of $\{1,\ldots,n+k-1\}$. This can be achieved by using *transpositions* for mutation of chromosomes. A transposition denotes the exchange of the contents of two genes in a chromosome. The positions of the two genes are randomly chosen. A transposition can either modify the offsets of two variables or change the AR assignment in GOA. Since any permutation can be composed by series of transpositions, all GOA solutions, in particular the optimal one, can be reached by using only transpositions as mutation operators.

3.3 Crossover

In order to accelerate the convergence of the GA, also crossover operations are applied to generate new individuals in the GA population. Just like mutations, the crossover operator has to be permutation preserving, so as to obtain only valid solutions. In our approach, we use the standard *order crossover* operation (fig. 2), which generates two offspring individuals from two parent individuals as follows:

1. Randomly choose two gene positions in the parents' chromosomes A and B. These positions induce a three-partitioning $A = (A_1, A_2, A_3)$ and $B = (B_1, B_2, B_3)$.

2. Mark the genes occurring in A_2 in B and the genes occurring in B_2 in A.

3. Generate a new chromosome A' from A: Starting with the leftmost position of interval A_2, in left-to-right order, write the non-marked genes of A into A_1 and A_3, while leaving interval A_2 as a gap. Generate a new chromosome B' from B analogously.

4. Copy the center interval from B to A', and copy the center interval from A to B'.

3.4 Fitness function

The fitness function Z is a metric for the quality of an individual I in the population. For a given variable access sequence $S = (s_1,\ldots,s_m)$, function Z decodes a chromosome (w.r.t. offset and AR assignment) and counts the number of address computations that can be implemented either by switching the AR index or by auto-increment. Like auto-increment, switching the AR index does not require an extra machine instruction. Thus, higher fitness corresponds to a cheaper offset assignment. Each individual I induces a partitioning of the variable set V into disjoint subsets $V_1,\ldots,V_k$, as well as k "local" offset functions $F_1,\ldots,F_k$. Each variable v in a subset V_i is addressed by AR number i and has a global offset of

$$F(v) = |V_1| + \ldots + |V_{i-1}| + F_i(v)$$

In summary, the fitness function is defined as

$$Z(I) = \sum_{i=1}^{m-1} d(i)$$

with

$$d(i) = \begin{cases} 1, & \text{if } |F(s_i) - F(s_{i+1})| \leq 1 \text{ and} \\ & s_i \text{ and } s_{i+1} \text{ are in the same variable} \\ & \text{subset } V_j \in \{V_1,\ldots,V_k\} \\ 1, & \text{if } s_i \text{ and } s_{i+1} \text{ are} \\ & \text{in different variable subsets} \\ 0, & \text{else} \end{cases}$$

3.5 GA and parameters

The initial population consists of a set of randomly generated permutations, where exactly one individual is computed by a GOA heuristic [9], so as to provide a "seed" to the GA, which accelerates the convergence towards the optimum. In each iteration (generation) of the GA, the fitness of all individuals in the population is evaluated. The "strong" individuals (i.e., a fraction of the total population with highest fitness) are selected

for crossover. On a fraction of the offspring generated by crossover, a mutation is performed. Finally, the offspring replaces the "weak" individuals in the population. This is iterated until a termination condition is satisfied. After termination, the fittest individual in the final population is emitted as the best solution.

A crucial aspect in the applicability of GAs are the parameters that guide the optimization procedure. Based on extensive experimentation, we have selected the following parameters:
Mutation probability per gene: $1/(n+k-1)$
Population size: 30 individuals
Replacement rate: 2/3 of the population size
Termination condition: 5,000 generations or 2,000 generations without a fitness improvement

For GOA problem sizes of practical relevance ($|V| \leq 90, |S| \leq 100$), we have measured maximum runtimes of 12 CPU seconds on a 300 MHz Pentium II PC. This is about 10 times more than for previously published heuristics. However, as shown in the following, the GA procedure achieves better solutions, so that the additional computation time is justified.

3.6 Results

We have statistically evaluated the performance of the above genetic optimization, as compared to the GOA heuristic described in [9]. For different problem parameters (k, $|S|$, and $|V|$) we have measured the average quality of computed GOA solutions over 100 random variable access sequences[2]. The results are summarized in table 1 which gives the average number of extra instructions for address computations for each parameter combination, as well the percentage of the GA as compared to the heuristic.

The results show, that the proposed GA on the average outperforms the heuristic by 18 %. The generalization described in the next section, however, provides even higher improvements.

k	$\|S\|$	$\|V\|$	GOA [9]	GA	%
2	50	10	14.3	11.9	83
2	50	20	18.0	16.3	91
2	50	40	10.4	9.8	94
2	100	10	33.0	29.3	89
2	100	50	41.6	41.0	98
2	100	90	11.2	11.0	98
4	50	10	5.7	4.3	75
4	50	20	10.7	7.0	65
4	50	40	9.5	7.1	75
4	100	10	9.8	6.6	67
4	100	50	29.0	27.5	95
4	100	90	9.5	8.2	86
8	50	10	5.0	4.2	84
8	50	20	9.0	5.7	63
8	50	40	11.9	7.1	60
8	100	10	5.0	5.0	100
8	100	50	20.6	15.9	77
8	100	90	12.7	9.0	71
				average	82

Table 1: *Experimental results: Genetic optimization for GOA*

4 Inclusion of modify registers

As explained in section 1, also auto-modify operations can be exploited to avoid code overhead for address computations. Auto-modifys permit to implement some address computations in parallel which cannot be covered by auto-increments. This requires that MRs be loaded with appropriate values, i.e., offset differences. While previous GOA algorithms neglect auto-modifys, exploitation of such operations may lead to reduced address computation code.

So far, few approaches have been reported to exploit MRs during offset assignment. Wess [15] considered the AGU architecture of an ADSP-210x, which contains $m = 4$ MRs. In his technique, these MRs are *statically* loaded with the values {-2,-1,1,2}. Conceptually, this approach is equivalent to an increase in the auto-increment range l, which will be discussed in section 5.

[2]Text files containing all detailed problem instances and computed offset assignments can be obtained from the authors.

In [9], we have argued that *dynamic* loading of MRs yields a higher optimization potential, since a larger search space is explored. It has been shown, that an extension of a page replacement algorithm (PRA) for operating systems [12] can be applied to determine the best values of m MRs at each point of time. The main idea of the PRA is, that a new offset difference d should be loaded into a MR currently storing a value $d' \neq d$, whenever the next use of d in the remaining sequence of address computations precedes the next use of d'. The effect of the PRA, as compared to a pure GOA solution, is exemplified in fig. 3 a) and b). The algorithm is not affected by orthogonality or signed-ness of MRs (cf. section 1). In fact, the PRA yields optimal results, but needs to be applied as a *post-pass* optimization after solving GOA. Thus, its potential optimization effect is already reduced by the computed offset assignment.

Considering auto-modifys *during offset assignment* gives much higher opportunities for optimization. As illustrated in fig. 3 c), better and completely different memory layouts than with the pure GOA approach may be computed when simultaneously considering auto-modifys.

We can exploit this additional optimization potential in the GA technique from section 3 by *integrating the PRA into the fitness function Z*. Using an appropriate implementation, the PRA takes only linear time in $|S|$ and thus hardly increases the runtime required to compute $Z(S)$, which also takes linear time. The integration works as follows: The PRA is executed on each offset assignment represented by an individual I in the population. It returns the number $M(I)$ of address computations that can be saved (besides application of auto-increment) by optimal exploitation of auto-modifys for the represented offset assignment. The fitness function

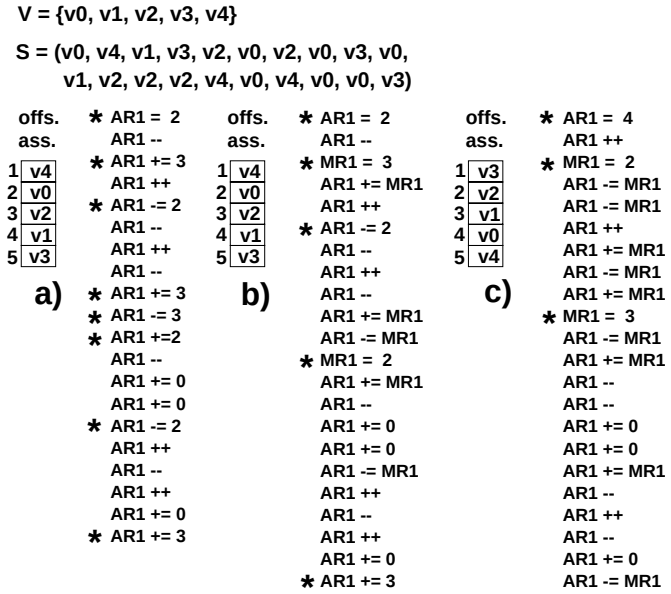

Figure 3: *Example for exploitation of modify registers ($k = 1, m = 1$), "*" denotes address computations not implemented by auto-increment or auto-modify: a) GOA solution, b) post-pass utilization of auto-modifys with PRA, c) integrated GOA and MR exploitation (optimality can be shown for this example)*

from section 3 is replaced by

$$Z(I) = \sum_{i=1}^{m-1} d(i) + M(I)$$

This approach uses the PRA no longer as a post-pass procedure, but as an additional means to measure the quality of an offset assignment during the GA. Therefore, the potential optimization due to MRs influences the computed offset assignment itself.

We have compared the performance of the GA using the new fitness function to a "conventional" approach that uses the PRA only as a post-pass optimization after GOA. The results are shown in table 2. The parameters are like in table 1. The number m of MRs (unsigned values assumed) has been set equal to k. The GA based technique yields better results than the previous technique for all parameter combinations, with an average reduction of costly address computations of 32 %.

5 Inclusion of larger auto-increment ranges

So far, we have considered a fixed auto-increment range (AIR) of $l = 1$, which is, for instance, given in the TI 'C25 and the Motorola 56000 DSPs. Other machines, such as the TI 'C80 or the AMS Gepard have an AIR of $l = 7$, i.e., much larger offset differences can be covered by auto-increment operations.

$k = m$	$\|S\|$	$\|V\|$	post-pass	integrated	%
2	50	10	5.9	3.6	61
2	50	20	11.4	6.5	57
2	50	40	9.1	6.9	67
2	100	10	12.6	4.0	32
2	100	50	38.6	28.7	74
2	100	90	11.0	9.5	86
4	50	10	5.0	4.0	80
4	50	20	7.6	5.1	67
4	50	40	9.0	6.2	69
4	100	10	5.2	4.1	79
4	100	50	17.3	9.3	54
4	100	90	8.7	7.2	83
8	50	10	5.0	4.0	80
8	50	20	8.9	5.2	58
8	50	40	11.7	7.3	62
8	100	10	5.0	4.1	82
8	100	50	15.0	9.9	66
8	100	90	12.2	8.6	70
				average	68

Table 2: *Experimental results: GOA including auto-modify optimization*

An optimization technique for AGUs with an arbitrary AIR $l \geq 1$ has been given in [13]. It is an extension of the graph-based heuristic reported in [6], and actually considers the AIR only when solving the SOA problem, i.e., after the variables have been assigned to ARs. Therefore, its optimization effect for GOA is presumably very limited. Furthermore, the potential use of MRs is neglected.

As already mentioned in section 4, Wess' technique [15] uses static MR values to achieve an AIR of $l = 2$. The offset assignment is computed by simulated annealing. Similarly, also the technique in [14] only works for a fixed AIR of $l = 2$.

Our GA approach permits to accommodate arbitrary l values during GOA by a minor modification of the fitness function. Instead of checking $|F(s_i) - F(s_{i+1})| \leq 1$ during computation of the $d(i)$ values from section 3.4, we check, whether $|F(s_i) - F(s_{i+1})| \leq l$. Thereby, an address computation is only considered "costly", if the offset distance exceeds l.

Since this extension is completely independent of the MR utilization technique from section 4, our GA technique actually considers the trade-off between auto-increments and auto-modifys (with dynamic MR reloading) when optimizing the offset assignment. As compared to an approach with static MR contents, this provides better solutions.

For experimental evaluation, we have compared our generalized GA technique to Wess' approach. The parameter combinations and results are taken from [15], while the AGU configuration corresponds to that of an ADSP-210x DSP. Table 3 summarizes our results.

Column 3 gives the results (number of extra instructions for address computations) achieved by Wess' technique. Since in that approach, the 4 available MRs are statically loaded with the values -2, -1, 1, 2, no further MRs are left ($m = 0$), and the effective AIR is

$(\|V\|, \|S\|)$	k	Wess $l=2$ $m=0$	GA $l=2$ $m=0$	%	GA $l=0$ $m=4$	%
(15,20)	1	1.8	2.2	122	0.9	50
(10,50)	1	11.9	14.1	118	10.2	86
(10,50)	2	1.8	1.2	67	1.0	56
(25,50)	2	6.7	7.6	113	6.7	100
(40,50)	4	3.0	1.1	37	0.8	27
(50,100)	4	15.6	17.4	112	13.9	89
average				95		68

Table 3: *Experimental results: Generalized GA*

$l = 2$. Column 4 shows the GA results for the same configuration, i.e., $l = 2, m = 0$, while column 5 shows the percentage. The results are of similar quality on the average. This is no surprise, since the simulated annealing technique is known to produce results of a quality comparable to GAs.

However, as indicated by the data in column 6 (percentage in column 7), the GA outperforms Wess' technique, when the restriction of static MR contents is removed, so that all 4 MRs become available for dynamic reloading. Since the ADSP-210x permits no auto-increments *without* using MRs, we have set $l = 0$. As can be seen, the larger flexibility in most cases results in better solutions, since the GA itself decides whether static or dynamic values of MRs are preferable for a certain instance of the offset assignment problem. On the average, the amount of extra instructions for address computations is reduced to 68 %.

6 Conclusions

Offset assignment is a relatively new code optimization technique for DSPs, which exploits the special address generation hardware in such processors. We have presented a novel GA based approach to solve offset assignment problems for different AGU configurations of DSPs. One main advantage of the proposed technique is, that the functionalities of a number of previous techniques are covered by a uniform optimization procedure, which is only adjusted by parameters in the fitness function. In addition, as a consequence of its generality, the GA yields significantly better offset assignments than previous algorithms. Practical applicability in compilers is ensured, because the GA is sufficiently fast and comparatively easy to implement. We are currently investigating further generalizations of the GA technique, such as incorporation of variable live range information and offset assignment in presence of multiple AGUs.

References

[1] 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

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

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

[4] R. Leupers, A. Basu, P. Marwedel: *Optimized Array Index Computation in DSP Programs*, Asia South Pacific Design Automation Conference (ASP-DAC), 1998

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

[6] 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

[7] N. Sugino, H. Miyazaki, S. Iimure, A. Nishihara: *Improved Code Optimization Method Utilizing Memory Addressing Operation and its Application to DSP Compiler*, Int. Symp. on Circuits and Systems (ISCAS), 1996

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

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

[10] L. Davis: *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991

[11] B. Landwehr, P. Marwedel: *A New Optimization Technique for Improving Resource Exploitation and Critical Path Minimization*, 10th Int. Symp. on System Synthesis (ISSS), 1997

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

[13] A. Sudarsanam, S. Liao, S. Devadas: *Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures*, Design Automation Conference (DAC), 1997

[14] N. Kogure, N. Sugino, A. Nishihara: *Memory Address Allocation Method for a DSP with ± 2 Update Operations in Indirect Addressing*, Proc. ECCTD, Budapest, 1997

[15] B. Wess, M. Gotschlich: *Optimal DSP Memory Layout Generation as a Quadratic Assignment Problem*, Int. Symp. on Circuits and Systems (ISCAS), 1997