Optimized Address Assignment for DSPs with SIMD Memory Accesses*

Markus Lorenz, David Kottmann, Steven Bashford, Rainer Leupers, Peter Marwedel
Dept. of Computer Science 12
University of Dortmund, Germany
email: {lorenz, kottmann, bashford, leupers, marwedel}@ls12.cs.uni-dortmund.de

Abstract

This paper deals with address assignment in code generation for digital signal processors (DSPs) with SIMD (single instruction multiple data) memory accesses. In these processors data are organized in groups (or partitions), whose elements share one common memory address. In order to optimize program performance for processors with such memory architectures it is important to have a suitable memory layout of the variables. We propose a two-step address assignment technique for scalar variables using a genetic algorithm based partitioning method and a graph based heuristic which makes use of available DSP address generation hardware. We show that our address assignment techniques lead to a significant code quality improvement compared to heuristics.

I. Introduction

The growing number of *digital signal processors* (DSPs) in embedded systems makes the use of optimizing compilers more and more desirable. However, in order to meet given constraints with respect to execution-time, code size and/or energy consumption, many programs are still written in assembly code. Unfortunately, hand crafting code is a very time consuming process which potentially leads to incorrect and hardly portable code.

Compilation first transforms a given high level source program into an intermediate representation (IR). After performing machine independent standard optimizations, a code generator maps the IR to assembly code by solving the following subtasks:

- *Code selection* covers the nodes of a *data flow graph* DFG using suitable processor instructions.
- *Instruction scheduling* determines the execution order of the processor instructions.
- *Register allocation* determines which variables have to reside in registers or be spilled to memory.
- *Address assignment* determines the memory positions (addresses) of the variables which have to be stored.

In order to meet the specified constraints with respect to code quality, the code generator has to make use of special architecture features in these optimization steps. Thus, it is especially necessary to exploit *fine grain* parallelism to handle restricted interconnection networks and heterogeneous register files in order to reduce execution time, chip area, and/or power consumption.

There are different strategies for memory accesses. For example, the DSP56000 [1] and GEPARD [2] contain dual data-memory banks in order to overcome limited memory bandwidths. In these cases one goal of code generation is to maximize the number of parallel loads in order to reduce the execution time [3, 4].

MicroUnity's media processor [5] and the M3-DSP platform [6] use a wide memory (*group memory*). Here, addressing of one memory word means to access all data words belonging to the addressed group. Processing is performed according to the single instruction multiple data (SIMD) principle, and does not allow to access arbitrary sets of words (groups) in the memory. Thus, in these cases techniques for computing a suitable order of variables in the memory are essential for high code performance.

Furthermore, it is typical for DSPs that they have special *address generation units* (AGUs) which allow address computation in parallel to other machine instructions. Thus, exploitation of AGUs is also essential for efficient code generation.

In this paper we concentrate on address assignment of scalar variables for the M3-DSP which is an instance of the M3-DSP platform [6]. The proposed address assignment technique is subdivided into the *horizontal* and *vertical ordering* step: Horizontal ordering assigns variables to groups. This is done by a partitioning method based on a genetic algorithm that minimizes the number of memory accesses. The vertical ordering step assigns memory groups to addresses by optimizing the use of AGUs with respect to code size.

The remainder of this paper is organized as follows: In the next section we introduce the main features of our target architecture. Algorithms for the horizontal and vertical ordering step are described in section III and IV. Results for both subproblems are given in section V. Section VI will conclude the paper with a summary.

II. Target architecture (M3-DSP)

The M3-DSP (fig. 1) is an instance of the scalable DSP platform [6] for mobile communication applications. The platform allows for a fast design of DSPs adapted to special applications. In order to meet constraints w.r.t. real-time processing, chip area, and energy dissipation the platform supports some special features:

There is a scalable number of data paths that allow processing either on a single data path or on all data paths in parallel according to the SIMD principle. In the case of the M3-DSP

*This work has been sponsored by the DFG (Deutsche Forschungsgemeinschaft) and Agilent Technologies, USA.

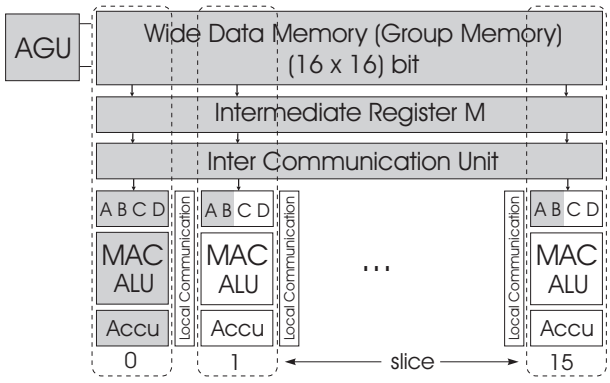

Fig. 1: Architecture of the M3-DSP

there are 16 data path slices. In order to provide an effective use of all data path slices in parallel, the memory is organized as a *group memory*: Addressing one 16-bit data word means addressing an entire group of 16 such words. The addressed group is loaded into an intermediate register from which the values are distributed to the *group registers* in the data paths by an application-specific inter-communication network. The term group register denotes the set of the data path input registers in all slices with the same label (e.g. A or B in fig.1).

Here, the greatest challenges in code generation are to make use of the whole memory bandwidth and the SIMD data path instructions. In order to demonstrate our address assignment technique it is sufficient to perform processing on one slice (slice 0), while still exploiting the whole memory bandwidth. Exploiting all data paths is one topic of our current research. In this paper we will concentrate on code generation for the gray shaded area in fig. 1. We do not use group registers C and D of the other data paths because storing is not allowed from these registers.

The address generation unit (AGU) contains four address pointer registers $P_0, \ldots, P_3$ which allow auto-increment addressing with an offset *off* $\in \{-8, \ldots, 7\}$. If there is need for larger offsets, the address pointer registers can be used orthogonally with the four modify registers $M_0, \ldots, M_3$ in auto-modify operations. The *page pointer register PP* can be used with a 6-bit offset[1].

The M3-DSP is organized as a *very long instruction word* (VLIW) architecture which allows an independent control e.g. for data manipulation, data transfer, program control, and the address generation unit. Unfortunately, the use of VLIW instruction set architectures leads to a code size overhead because sub-instructions for idle units are also stored in the instruction memory. In order to reduce the code size overhead a *Tagged* VLIW (TVLIW) method is used [7]. The idea is that the next VLIW is assembled by an instruction decoder (fig. 2) for one or more TVLIW's which contain only *functional unit instruction words* (FIWs) for two *function units* (FUs). The number of required FIWs for assembling the next VLIW is indicated by the IWC (*instruction word class*).

Only those parts of a VLIW need to be stored, which are required for assembling the next VLIW. This means on the one hand that code size overhead is avoided for idle units and on the other hand that identical sub-instructions of two successive VLIW's can be reused and do not need to be stored as instructions in memory. In order to provide an effective use of this method in loops the M3-DSP also contains an *instruction cache*

[1] In addition, addressing can be also done by using a circular buffer.

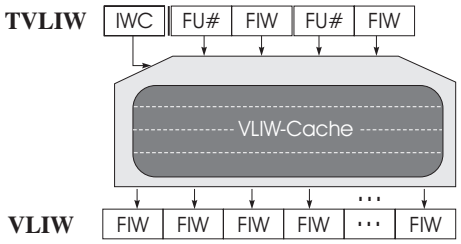

Fig. 2: Tagged VLIW Instruction Decoder

for up to four (pre-assembled) VLIW instructions.

III. Horizontal address assignment

The entire task of address assignment (fig. 3) is subdivided into a *horizontal* and *vertical* ordering step. Inputs are the *variable access sequences* VAS for the horizontal and the *group access sequence* GAS for the vertical (see section IV.) optimization step.

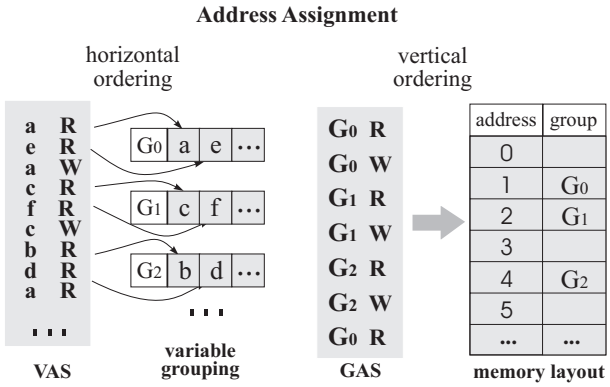

Fig. 3: Horizontal and vertical address assignment

A. Problem definition

The horizontal address assignment has the goal of assigning each variable to a group (partition), while minimizing the number of memory accesses. The *variable access sequence* VAS contains (memory-)accesses of scalar variables and is the result of the instruction selection, instruction scheduling, and register allocation phases. These phases are performed by using only one memory slice. Thus, the same VAS can be taken as input for DSPs with different memory widths. Each element of VAS is tagged either by an R (read) or a W (write) in order to distinguish between different memory access modes.

We say that two variables var_i and var_j of VAS are *neighbors* and have a *neighbor relation* if there are successive accesses in VAS to these variables. For example the variables a and e of fig. 3 are neighbors twice. Two variables var_i and var_j have an *unexploited neighbor relation* if they are neighbors and are not members of the same group.

The main concept of utilizing the whole memory width is as follows: Load the group containing the required data and work on these data as long as possible without further memory accesses. If another group should be loaded into the group register and the currently loaded group is modified (indicated

by a write access of a variable in VAS) it is necessary to store the current group back to memory. Obviously, we can minimize the number of memory accesses by minimizing the number of unexploited neighbor relations.

Optimizing should be done under the condition that

- the number of elements in a group is restricted by the memory width max_{size},
- every variable must be assigned to exactly one group[2],
- the position of a variable in the group is irrelevant, and
- the number of resulting groups is not known in advance.

Now, we can represent the horizontal address assignment problem as a graph partitioning problem using a graph representation as described in [8]:

Definition: The *Variable Access Graph* VAG $= (V, E)$ is an undirected graph with node set $V = \{v_1, \ldots, v_n\}$. Each node v has a corresponding variable var in VAS. The edge set E contains an edge e_{ij} between the nodes v_i and v_j if the corresponding variables var_i and var_j of VAS are neighbors. An edge which represents an unexploited neighbor relation is called *external edge*, otherwise this edge is called *internal edge*. The weight w_{ij} of an edge e_{ij} is given by the number times that var_i and var_j are neighbors.

Fig. 4 depicts the graph representation VAG of a given variable access sequence VAS and three different memory layouts.

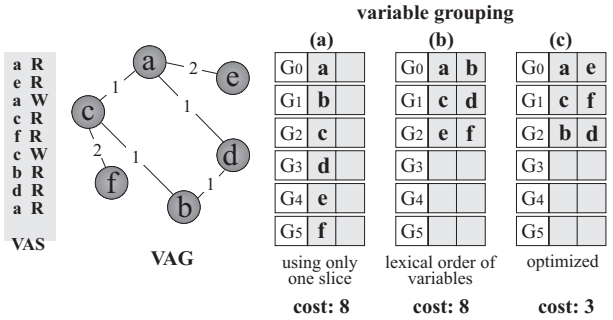

Fig. 4: Horizontal address assignment

The costs of the respective variable groupings are computed by adding the weights of external edges. It can be seen that using only one memory slice (grouping a) or a naive partitioning (grouping b) are both poor. In contrast to this the variable grouping (c) optimized for the concrete VAS has much lower costs.
This example shows that it is desirable to optimize the variable grouping by a suitable partitioning algorithm. Unfortunately, this means solving an NP-complete problem [9]. For this reason we have implemented some simple heuristical approaches and the well known Kernighan-Lin algorithm [10]. In order to improve the heuristical results we have developed a partitioning algorithm based on genetic algorithm, which is presented in the next section.

[2]This is necessary in order to avoid overhead which is caused by accesses.

B. Genetic partitioning

Genetic algorithms (GA) have proven to solve complex optimization problems by imitating the natural optimization process (see e.g. [11, 12] for an overview). A population of a GA consists of several individuals, each of them representing a potential solution for the optimization problem. The representation of an individual is given by a *chromosome* which is subdivided into *genes*. The genes are used to encode the variables of the optimization problem. This means that finding a suitable combination of *alleles* (concrete values) for the genes is the same as finding good solutions for the optimization problem. By applying genetic operators like *selection*, *mutation*, and *crossover* to the members of the population, the fitness of the individuals will increase in the course of the generations.

In the next section, we first describe the coding mechanism and afterwards the initialization, evaluation, crossover, and mutation steps of our genetic partitioning technique in more detail.

B.1. Chromosomal representation

Finding an appropriate chromosomal representation is essential for employing genetic algorithms. In our case we have to assign each variable to a group. Thus, we represent each variable occurring in the given variable access sequence as a gene of a chromosome. An allele indicates to which group the variable should be assigned. For example, variables a and e in fig. 5 are members of the same group 0.

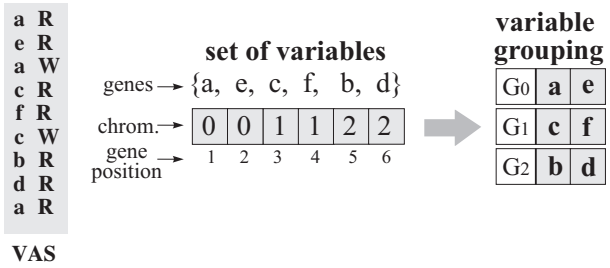

Fig. 5: Chromosomal representation

B.2. Initializing

Initialization of each gene is done by traversing the chromosome from left to right with growing gene positions. In order to meet the given group size constraint we determine the set of alternative alleles whose selection would lead to a valid solution. Elements of this set are all non-empty groups which contain less than max_{size} variables and a new (empty) group. Obviously, the complexity of this step is $O(|V|)$.
Fig. 6 shows an initialized chromosome. The initialization process is done for variables a - d by performing a probabilistical selection of an element of the set of alternatives (depicted below the genes). Assuming a partition size of two, initialization of variable f is only possible to the groups G_0 and G_2 because G_1 already contains the members e and c. The resulting variable grouping shows that the number of groups can be different for the individuals. Thus, an assignment of variable b to group G_0 would result in only three groups.

B.3. Evaluation

The evaluation function of a genetic algorithm is necessary to distinguish the individual's quality within a population. Thus,

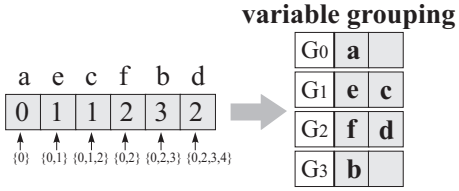

Fig. 6: Initialization of an individual

solving a minimization problem indicates that the individual with the lowest evaluation result is the best solution. In this case we want to minimize the number of memory accesses of a program. As we have seen in the last section, this is the same task as minimizing the sum of external edge weights. However, we need to modify this cost function in order to get a closer relation to code generation: First, costs for accessing the first variable in VAS are not taken into account. Second, writing a variable var to memory means that the group containing this variable must reside in a group register. Thus, we have to look at the variable access sequence and have to add one additional memory access if a write access occurs for a variable while accessing variables of a group. This step can be done in $O(|VAS|)$. In the case presented in fig. 4 this will result in 11 memory accesses for grouping (a) and (b), and 6 for grouping (c).

B.4. Crossover and Mutation

The crossover operator deals with generating new individuals by probabilistically swapping genes between two selected individuals. In our case we are using a simple *one point crossover* (fig. 7).
The crossover is performed by a probabilistic choice of a crossover point (e.g. in fig. 7 this is gene c). Then, all genes of the parents behind this point are swapped.

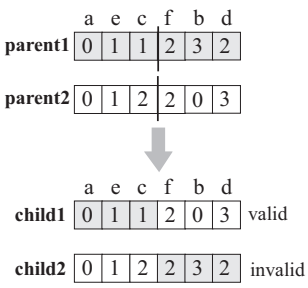

Fig. 7: Crossover

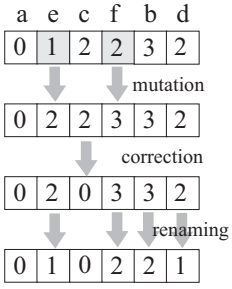

Fig. 8: Mutation

The result consists of two individuals which contain the combined information of the parents. However, the crossover step can lead to invalid solutions. In order to avoid invalid solutions we combine the subsequent mutation operator with a correctness check. Thus, the main tasks of the mutation operator are to generate the new gene material by exchanging alleles and checking the correctness of the actual allele. This is done by performing the following steps:

1. Mark probabilistically all genes which should be mutated.
2. Perform a probabilistic choice of a new allele from the allele set for all marked genes. Let G_{max} be the actual largest assigned group number. Then, the set of alleles includes the groups $G_1, \ldots, G_{max}$ and the empty group G_{max+1}.
3. Correct the number of elements in groups which have more than max_{size} number of elements. This is done by distributing elements either to group G_{max+1} or to groups which can pick up further elements.
4. Perform a renaming of the groups from left to right in the chromosome in order to delete groups which are not used.

Fig. 8 illustrates this procedure for the invalid offspring of fig. 7. Analogous to the initialization phase the complexity of mutation and crossover is $O(|V|)$.

C. Extensions to the partitioning algorithm

We have implemented the following extensions for our partitioning algorithm:

- Initialization of some individuals can be done with the result of heuristic partitioning algorithms.
- In order to enable sharing of memory locations between variables, dependent on their life ranges, we have combined the mutation operator with the *left edge algorithm* [13]. Thus, local variables without overlapping life ranges can share one memory position in a group. This can lead to group sizes which are greater than max_{size}.
- It can be desirable to iterate the partitioning algorithm several times with given partitioning constraints. Thus, it is possible to specify constraints with respect to an existing variable grouping.
- Edge weights for variables used in loops can take into account the number of loop iterations.

In section IV we will propose a heuristical technique which determines the address instructions for a given group access sequence (GAS).

IV. Vertical address assignment

A. Problem definition

The goal of vertical address assignment is to determine an optimized ordering of available groups in the memory which allows to minimize the code size by choosing suitable address generation instructions. However, the following facts cause problems in solving this task: First, the number of possible memory layouts is exponential with respect to the number of groups which have to be taken into account. Thus, testing all memory layouts is not practical in most cases. Second, computing the quality of a given memory layout means to solve the *general offset assignment* (GOA) problem (see e.g. [8, 14, 15, 16] for details). Here, a set of address and modify registers are given, which are used to address variables in the memory. Unfortunately, this is also an NP-complete problem. In addition, we need to take into account the VLIW cache (fig. 2) in our optimization technique. Thus, we can reduce the code size by maximizing the reuse of address generation instructions which reside in a VLIW of the instruction cache.
For the example memory layout in fig. 9, a *group access sequence* GAS indicating the order of addresses which have to be generated by address instructions, and three AGU instruction sequences are given. The costs below the sequences are dependent on the number of instructions which must be stored to the instruction memory (instructions in bold in fig. 9).

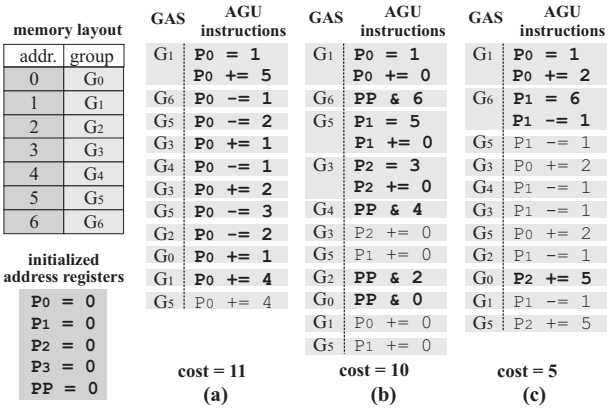

Fig. 9: Example of address assignment techniques

For sequence a) we assume that addressing is done only with address register P_0 and an offset. Without making use of the instruction cache this will result in 11 instructions to be stored. In this case, code size reduction is only possible if the same address instructions are used in successive instructions. Sequence b) is generated by determining those three groups (or addresses) which are accessed most frequently. Then, addressing for these groups is done by loading the group address into an unused address register which is used subsequently for addressing this group with a zero offset. All other (non-addressed) groups are addressed by the page pointer register *PP* and a suitable offset. Sequence c) shows that the cost can be further reduced by a dedicated address generation technique which makes better use of the instruction cache.
We can represent this task as a graph problem where the graph nodes are given by the group set accessed in GAS.

Definition: The *Group Access Graph* GAG $= (V, E)$ is a directed acyclic graph with node set $V = \{g_1, \ldots, g_k\}$. Each node g_i represents one group access in GAS. If $i < j$ we insert an edge e_{ij} from node g_i to node g_j. The weight w_{ij} of an edge e_{ij} is given by the address offset of the corresponding groups G of g_i and g_j.

The corresponding GAG of the group access sequence of fig. 9 is depicted in fig. 10. Obviously, finding long paths (with respect to the number of nodes) along edges with same offset tends to minimize the code size.

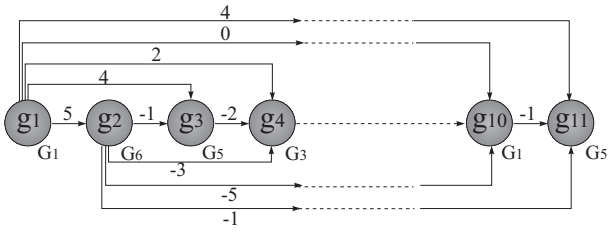

Fig. 10: Problem graph representation (GAG)

B. Heuristical approach

Our heuristic algorithm for vertical address assignment is as follows. As long as the graph contains unmarked nodes perform the following steps:

1. Take the first/next unmarked node g_i.
2. If (cache size - 1) is equal to the number of currently used address instructions in the instruction cache goto step 3d.
3. Determine the longest path starting from node g_i only by using edges with same weights. These nodes can all be addressed by the same address generation instruction, if the used address register is not modified between the first and last access. Choose the address generation instruction in the following order, taking into account that it is not allowed to choose address registers which are still needed for addressing of subsequent already marked nodes:
   (a) Try to reuse one of the available address generation instruction in the instruction cache. Candidates are auto-increment or auto-modify instructions.
   If success goto step 4.
   (b) Try to address all nodes on the path with an (new) auto-increment instruction.
   If success goto step 4.
   (c) Address these nodes with an auto-modify instruction. If necessary load the correct constant into the modify register.
   If success goto step 4.
   (d) Address the current node by the page pointer register PP and an offset. Choose always the same instruction cache entry.
4. Mark all addressed nodes on this path as addressed and delete all edges adjacent to these nodes. If there are unmarked nodes goto step 1 else stop.

Applying this algorithm to the group access sequence of fig. 9 would result in the AGU instruction sequence c).

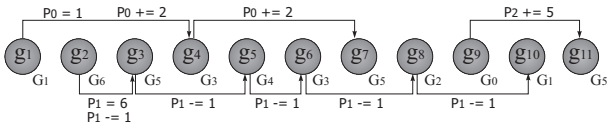

Fig. 11: Graph representation of sequence c) in fig. 9

The described algorithm optimizes the address generation instructions for **one** given memory layout while minimizing code size. It can be shown that the complexity is $O(|GAS|^3)$. Usually, it is possible to find better solutions by comparing different memory layouts with respect to the resulting code size. But, testing all memory layouts will in most cases not be practical. Thus, we use a modified genetic algorithm of the horizontal ordering step to search for a suitable memory layout. Evaluation of the different memory layouts is done using the described heuristic algorithm.

V. Experimental results

First, we compare the quality of the Kernighan-Lin (KL) and the genetic partitioning algorithm from section III with optimal solutions obtained by a time-intensive CLP (constraint logic programming) partitioning algorithm. The results for four test sequences are depicted in table 1. Column 1 contains the number of different variables occurring in the test sequence. For

group size	genetic #mem	genetic cycles	genetic AGU ops	genetic cpu[s]	KL #mem	KL cycles	KL AGU ops	KL cpu[s]
4	17	82	6	8	21	91	10	< 1
8	10	71	6	8	15	79	8	< 1
12	10	71	4	8	10	71	4	< 1
16	6	65	4	8	7	67	4	< 1

Table 2: Results for the IIR example

#var	CLP (optimal) #mem	CLP (optimal) cpu[s]	genetic #mem	genetic cpu[s]	KL #mem	KL cpu[s]
5	71	< 1	71	12	71	< 1
10	122	7	122	17	126	< 1
15	134	4140	134	22	139	< 1
20	136	> 2 weeks	136	26	144	< 1

Table 1: Comparison of optimal, genetic, and KL partitioning algorithm with $|\text{VAS}| = 200$ and partition size = 4 (for the horizontal address assignment step)

each partitioning technique also the number of resulting memory accesses (#mem), which are important from a performance or power consumption viewpoint, and the respective runtimes (on a 333 MHz Sun Ultra-10) are mentioned. The main result here is that the genetic algorithm computes the optimal solution[3] (for these sequences) in all cases within acceptable runtime. For 20 variables the result of the KL heuristic is 5,8 % worse than the optimal solution[4]. Further experimental results have shown that using an optimized variable grouping can save up to approximately 60 % of the number of memory accesses in contrast to an arbitrary (unoptimized) grouping.

Table 2 gives results for a real DSP code example: an infinite impulse response (IIR) filter. In addition to the pure address assignment results, also the total effect on code quality is reflected. For this purpose, the results of the address assignment phase have been propagated to an existing M3 DSP code generator, so as to obtain a complete assembly program for the IIR example. Again, we compared the proposed genetic algorithm approach to the use of the KL heuristic. Column 1 gives the group size, i.e., the number of 16-bit data words per memory word, which is a parameter to the M3 DSP platform. Columns labeled "#mem", like in table 1, denote the resulting number of memory (i.e. group) accesses. Column "cycles" denotes the number of instructions cycles required by the assembly programs generated based on the given address assignment information. Column "AGU ops" denotes the number of different address computation instructions needed to be stored in VLIW instructions, thereby reflecting the code size. Finally, the required CPU times are given in seconds.

Naturally, the KL heuristic is faster than the genetic algorithm, but the latter generates significantly better code both in terms of memory accesses and performance. Also w.r.t. the number of required AGU operations, the genetic algorithm performs at least as good as KL.

VI. Conclusions

New high performance DSP processors need to be supported by advanced compiler algorithms. In this paper we have described memory allocation algorithms designed for the M3 DSP, high performance scalable SIMD architecture. In a first phase, variables are partitioned into variable groups, reflecting the M3's SIMD-like group access mechanism. A newly designed optimization based on a genetic algorithm has been shown to be capable of outperforming the traditional Kernighan-Lin algorithm. In a second phase, variable groups are allocated to memory. This is done by exploiting the M3's tagged VLIW instruction cache architecture.

Results have shown that optimizing the variable grouping can save a significant amount of memory accesses and increase code performance. So far, the technique is mainly tuned for the M3 architecture. However, adaptations for other SIMD DSP processor architectures would be relatively straightforward.

[3] For this and the following results of the genetic partitioning algorithm we have performed 10.000 generations with a population size of 50 and a replacement rate of 10 individuals.

[4] Note that the KL heuristic is also adapted to our cost function.

References

[1] Motorola. *DSP56000, Digital Signal Processor, User's Manual*, 1986.

[2] *GEPARD - Familiy of Embedded Software Programmable DSP Cores*. http://asic.amsint.com/databooks/digital/gepard.html.

[3] A. Sudarsanam and S. Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPs. In *Proceedings of the International Conference on Computer Aided Design*, pages 388–392, 1995.

[4] M. A. R. Saghir, P. Chow, and C. G. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operation Systems*, pages 234–243, 1996.

[5] Craig Hansen. MicroUnity's MediaProcessor Architecture. *IEEE Micro*, 16(4):34–41, Aug 1996.

[6] G. Fettweis, M. Weiss, W. Drescher, U. Walther, F. Engel, and S. Kobayashi. Breaking new grounds over 3000 MOPS: A broadband mobile multimedia modem DSP. In *Proc. of ICSPAT'98*, pages 1547–1551, Toronto, Canada, 1998.

[7] M. H. Weiss and G. P. Fettweis. Dynamic Codewidth Reduction for VLIW Instruction Set Architectures in Digital Signal Processors. In *3rd International Workshop on Image and Signal Processing*, pages 517–520. IEEE, 1996.

[8] S. Liao and S. Devadas. Storage Assignment to Decrease Code Size. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, New York, 1979.

[10] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. In *Bell System Technical Journal*, volume 49, pages 291–307, 1970.

[11] J. H. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, 1992.

[12] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[13] K. R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, New York, 1974.

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

[15] R. Leupers and F. David. A Uniform Optimization Technique for Offset Assignment Problems. In *Proceedings of the 11th International Symposium on System Synthesis*, 1998.

[16] B. Wess and M. Gotschlich. Optimal DSP Memory Layout Generation as a Quadratic Assignment Problem. In *Proceedings IEEE International Symposium on Circuits and Systems*, volume 3, pages 1712 – 1715, Hong Kong, 1997.