Low-Energy DSP Code Generation Using a Genetic Algorithm*

Markus Lorenz, Rainer Leupers, Peter Marwedel
Dept. of Computer Science 12
University of Dortmund, Germany
email: {lorenz, leupers, marwedel}@LS12.cs.uni-dortmund.de

Thorsten Dräger, Gerhard Fettweis
Mannesmann Mobilfunk Chair For Mobile Communications Systems
Technische Universität Dresden, Germany
email: {draeger, fettweis}@ifn.et.tu-dresden.de

Abstract

This paper deals with low-energy code generation for a highly optimized digital signal processor designed for mobile communication applications. We present a genetic algorithm based code generator (GCG), and an instruction-level power model for this processor. Our code generator is capable of reducing the power dissipation of target applications by means of two techniques: First, GCG minimizes the number of memory accesses by using a special list-scheduling algorithm. This technique makes it possible to perform graph based code selection and to take into account the high interdependencies of the subtasks of code generation by phase coupling. In addition, GCG optimizes the scheduling of processor instructions with respect to the instruction-level power model based on a gate level simulation. Experimental results for several benchmarks show the effectiveness of our approach[1].

1 Introduction

In the recent years the field of mobile communication became more and more a growing market. Especially the battery lifetimes have a great impact on whether a mobile device is successful on the market. Until now most research effort in reducing power consumption has focused on the field of low-power design of integrated circuits (for an overview see e.g. [18], [19]). But there is also a need for low-power *software* optimizations because of the increasing trend towards mapping embedded applications to programmable processors. Recent results indicate that power consumption can in fact be reduced at the software level. Known techniques include reordering of processor instructions [22, 13], special instruction and data encoding [22], and avoiding power intensive processor instructions like memory accesses [7]. An overview of low-power optimizations is given in [23, 16].

Embedded processor based systems have to meet real-time constraints while minimizing area and energy[2] consumption. Thus designers try to meet the given timing constraints by adding instruction level parallelism to processors [11]. Unfortunately, the use of heterogeneous register files for reducing chip area and power consumption leads to irregular processor architectures (our example in this paper is the M3-DSP [6]) which can be rarely handled by traditional compilers. This forces the industry to develop new compilers for each processor family supporting their particular features.
Compilation is done by transforming the given source program into an intermediate representation (IR) and performing machine independent standard optimizations. The source program consists of several basic blocks which can be represented as a set of DFGs (data flow graphs). After this transformation the code generator has to map the IR to assembly code by solving the following subtasks:

- CS (*code selection*) covers the nodes of a DFG using suitable processor instructions.
- IS (*instruction scheduling*) determines the execution order of the processor instructions.
- RA (*register allocation*) determines which variables have to reside in registers or have to be spilled to memory.

Due to the strong interdependencies of CS, RA and IS it is important for efficient code generation (particularly for DSPs) to perform all subtasks simultaneously by means of a complete *phase coupling*.

Optimization should be possible for different criteria like execution time, energy dissipation or code size. A big challenge concerns the handling of the trade-off between these objectives. For instance, in the field of mobile communication it is essential to generate assembly code which meets the real-time constraints and minimizes the energy consumption. However, it is often difficult to find a suitable compromise between these two goals because energy consumption

*Publication: ICCD, Austin, Texas (USA), September 2001, ©IEEE

[1] This work has been sponsored by the German Research Foundation (DFG) and Agilent Technologies, USA.

[2] In contrast to the term "power", "energy" additionally considers the number of clock cycles of the program, and is more interesting for battery-driven embedded systems.

not only depends on the number of processor instructions but also on their schedule. Thus, it is important that low-energy optimizations are performed under given real-time constraints.

In order to compare different instruction schedules with respect to the energy consumption it is very important to have a suitable cost model that allows for a quick evaluation. For instance, Lee et al. published an instruction-level power model based on measurement of the energy consumption of a single instruction (*base energy cost*) and of the switching activities of successive instructions (*overhead energy cost*) [13]. Further power models based on that work can be found e.g. in [21, 20].

It is already known (and is also confirmed by our power estimation for the M3-DSP in section 3) that the number of memory accesses has a more significant contribution to power consumption than other processor instructions. However, most traditional code generation techniques are based on tree based code selection algorithms [24]. These are runtime-efficient, but one of the main disadvantages is that using a tree based technique usually results in superfluous memory accesses. This is due to the decomposition of graphs into trees and performing separate code selection for each tree. Spill code is then added in a subsequent phase. We briefly compare the graph based and the tree based code selection for the following example: $d = (a + b) * c * (a + b)$

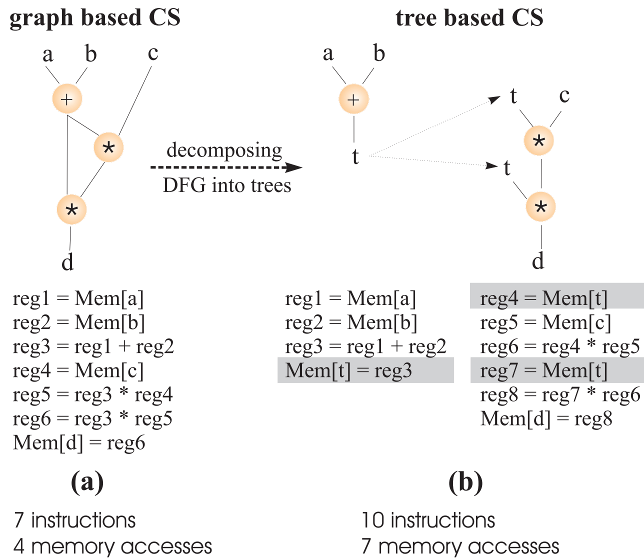

Fig. 1: Graph based and tree based code generation

Fig. 1 represents the corresponding DFG (a) and its decomposition into two trees (b). The resulting instructions are listed below. In this case splitting into trees is done after identifying the common subexpressions (CSE) (here: the "+" node).

On general-purpose processors with large homogeneous register files, CSEs would normally be kept in registers, and the register allocation phase in the compiler would aim at avoiding register spills and reloads. However, this approach usually does not work for DSPs with an irregular, special-purpose register architecture. The reason is that DSP register files show a very small storage capacity (frequently only a single value), so that live values cannot be kept in registers during several instruction cycles. Sometimes DSP registers even cannot be spilled at all, so that from a compiler viewpoint, the memory is the only "safe" resource for storing CSEs. Therefore it is very common in DSP compilers that CSEs are stored in memory right from the beginning and are reloaded into registers only at the time of further CSE uses [15, 1].

Only very recently techniques have been proposed that are capable of keeping (at least some) CSEs in special-purpose registers [4, 14] by means of graph based (instead of tree based) code selection. The code generation technique described in this paper uses a similar approach, but in contrast to earlier work focuses on energy optimization.

Phase coupled code generation techniques based on integer linear programming (ILP) (e.g. [25, 12]) allows generating optimal solutions for the given model. However, the complexity of the ILP-solver allows computation of optimal solutions only for small benchmarks or only for some code generation subtasks. A heuristic phase coupled code generation technique (AVIV) for VLIW-architectures is presented in [8].

Genetic algorithms have been proven very effective in finding optimal or near optimal solutions in huge search spaces. For this reason we are using a special list-scheduling algorithm in combination with a genetic algorithm. In contrast to earlier work using genetic algorithms for scheduling problems (e.g. [5, 26]) we solve the code generation subtasks CS, IS and RA.

The remainder of this paper is organized as follows: The next section gives an overview of the architecture of the M3-DSP. In section 3 we introduce the instruction-level power model that we use for low-energy optimization. The new low-energy code generation algorithm GCG is described in section 4. We demonstrate the effectiveness of our approach in section 5 and conclude the paper with a summary.

2 Architecture of the M3-DSP

The M3-DSP is an instance of the scalable DSP platform for mobile communication applications described in [6]. In order to ensure constraints with respect to real-time processing, chip area, and energy dissipation, the M3-DSP architecture has some special features (fig. 2): It consists of 16 data paths (slices) which allow for processing either on a single data path or on all 16 data paths in parallel according to the SIMD (single instruction multiple data) principle. For demonstrating the basic technique of GCG and our instruction-level power model it is sufficient to focus on code generation for the single slice mode (shaded area in fig. 2).

In order to enable an effective use of all data paths in parallel the memory is organized as a *group memory*. Hence, each memory address refers to group of 16x16-bit words. The addressed group is loaded into the intermediate (buffer) register M from which the values are routed to the registers in the data paths by an application specific interconnection network. The use of the special-purpose registers A, B, C, D and Accu is allowed for the following operations[3]:

[3]For the sake of simplicity we assume that subtractions are realized by adding with complemented registers.

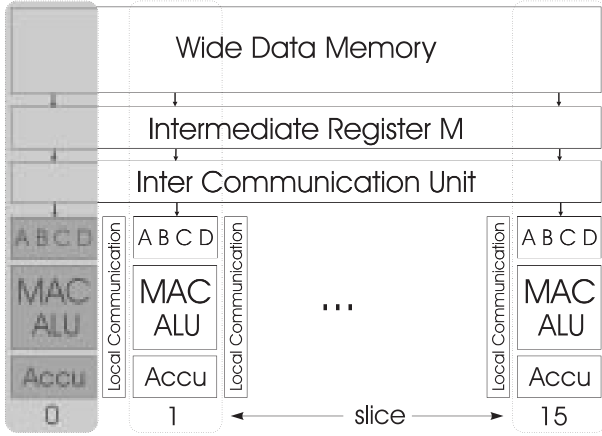

Fig. 2: Coarse architecture of the M3-DSP

`ADD`: Accu = {A, B, Accu} + {A, C, D, Accu}
`MUL`: Accu = {A, B} * {A, C, D, Accu}
`MAC`: Accu = {A, B, Accu} + {A, B} * {A, C, D, Accu}

For instance, Accu = B + A is a valid instruction whereas Accu = D * A is invalid. Further processor instructions are load and store instructions for memory (Mem) accesses and immediate load for constants (Int) and also data transports between registers.

`LD`: M = Mem
`ST`: Mem = {A, B, Accu}
`LDI`: {A, B, C, D } = Int
`DT`: {A, B, C, D } = M
`DT`: {A, B} = {Accu}

3 Instruction-level power model

An accurate cost model is needed in order to obtain power or energy optimized code. Such a model is used to estimate the consumed power or energy of a program. It is represented by values of average power dissipation of certain combinations of instructions. In our case the power dissipation is not physically measured but simulated at the gate level with a zero delay model using Synopsys Power Tools. According to the reference manual [10], the accuracy is within 10 - 25 % of SPICE simulation.

Following Lee et al. [13] the cost model is based on results obtained by simulating pairs of instructions. In contrast, however, our power model does not rely on base and overhead costs. Just using the simulation results, as described in the following, yields much better approximations in our case.

Gate-level simulations are very time consuming. Therefore it is preferred to simulate short instruction sequences. The M3-DSP comprises a four-stage-pipeline. Hence, simulating a pair of instructions is done after filling the complete pipeline with this pair to avoid transitions other than those between the two instructions. All instructions of the M3-DSP run through the same pipeline stages, each stage requiring one clock cycle. Hence, the pipeline is filled after four clock cycles and a pair of instructions always takes two clock cycles.

We do not distinguish between *asymmetric* transitions (transitions where the power dissipation of the transition I_1 to I_2 is not equal to the dissipation of I_2 to I_1). Differentiation is only possible with a high effort since only *vertical* simulation in time can be performed, i. e. the status of the processor is examined in each clock cycle. Thus, symmetric transitions are assumed. Fig. 3 illustrates the pipeline during the vertical simulation process and the mixed transitions between two clock cycles. U indicates an undefined instruction.

	Pipeline Filling				Simulation					
Clock Cycle	i-4	i-3	i-2	i-1	i		i+1		i+2	i+3
Pipeline Stages										
Fetch	I_1	I_2	I_1	I_2	I_1	$\xrightarrow{1,2}$	I_2	$\xrightarrow{2,1}$	I_1	I_2
Decode I	U	I_1	I_2	I_1	I_2	$\xrightarrow{2,1}$	I_1	$\xrightarrow{1,2}$	I_2	I_1
Decode II	U	U	I_1	I_2	I_1	$\xrightarrow{1,2}$	I_2	$\xrightarrow{2,1}$	I_1	I_2
Execute	U	U	U	I_1	I_2	$\xrightarrow{2,1}$	I_1	$\xrightarrow{1,2}$	I_2	I_1

Fig. 3: Pipeline during simulation

As a first step a cost model is developed which does not consider state dependencies, i.e. during the simulation the instructions should not change the state of the processor. For sake of simplicity, all operand values of an instruction have been set to zero. Table 1 shows a part of the obtained cost model. The upper half of table 1 comprises costs if the current instruction I_1 is equal to the subsequent one I_2. Here the high power dissipation of memory accesses (LD, ST) is remarkable. In the lower half of the table both instructions differ ($I_1 \neq I_2$). Comparing the upper and the lower half supplementary costs for transitions to a different instruction are evident.

Ins. 1	Ins. 2	Average Power Dissipation [mW]		
MAC	MAC		$C_{DT,DT}$ =	17.66
LD	LD		$C_{DT,DT}$ =	25.76
ST	ST		$C_{DT,DT}$ =	21.72
NOP	NOP		$C_{DT,DT}$ =	17.51
DT	DT		$C_{DT,DT}$ =	17.56
DT	ADD	$C_{DT,ADD}$ =	$C_{ADD,DT}$ =	24.72
DT	MUL	$C_{DT,MUL}$ =	$C_{MUL,DT}$ =	24.85
DT	MAC	$C_{DT,MAC}$ =	$C_{MAC,DT}$ =	24.73
DT	LD	$C_{DT,LD}$ =	$C_{LD,DT}$ =	25.70
DT	ST	$C_{DT,ST}$ =	$C_{ST,DT}$ =	38.16
DT	NOP	$C_{DT,NOP}$ =	$C_{NOP,DT}$ =	20.18

Table 1: Part of the power cost model

In order to estimate the energy dissipation of a sequence of instructions the sequence is separated into pairs. For each pair the power costs are added. But then the first and the last instruction have been considered only once whereas all other instructions have been considered twice. Thus, one half of the costs of the instruction at the start and at the end are added. This causes a small error because the transition costs for the first and the last instruction are inprecise. This error is small for longer instruction sequences, the predicted power

cost for a sequence of n instructions is:

$$PowerCost_{pred.} = \Big(\frac{1}{2}C_{I_0,I_0} + \sum_{i=0}^{n-1} C_{I_i,I_{i+1}} + \frac{1}{2}C_{I_n,I_n}\Big)/n$$

The particular costs C_{I_i,I_j} of the instruction pair I_i and I_j can be found in the cost model (table 1).

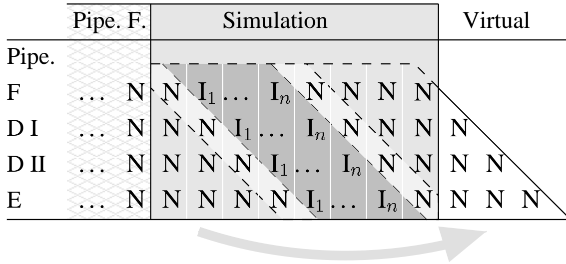

Fig. 4: Simulating an instruction sequence to compare with predicted power costs

To compare the predicted costs with the simulation result the pipeline has to be considered. In fig. 4 the inner trapezium comprises the sequence for which a prediction is to be made. Unfortunately, it is not possible to simulate just this trapezium. Since the pipeline is filled with undefined instructions before and after the sequence, the instructions at the beginning and at the end cannot be predicted properly. In order to avoid prediction inaccuracies due to vertical simulation, we add defined instructions. Initially the pipeline is filled with the same instruction e. g. the NOP (N) instruction and at the end of the sequence the same instruction is added as often as the pipeline depth defines (in our case four times). If we cut out the inner trapezium and merge the two outer triangles as fig. 4 illustrates, we would get another trapezium consisting of NOPs only. Thus, no inaccuracies are calculated at the beginning or at the end anymore.

In order to exemplify the use of the power model, fig. 5 shows the graph based example from fig. 1 using two different schedules. The difference is marked gray. Using the power cost model to calculate the average power consumption of the sequence leads to the energy dissipation by dividing by the clock frequency f_{clock} and multiplying with the number of instructions n:

$$EnergyCost_{predicted} = PowerCost_{predicted} * n \; / \; f_{clock}$$

In our case the clock frequency is equal to 20 MHz. Fig. 5 shows the difference of energy consumption between schedule (a) and schedule (b). Comparing the predicted values and the simulation result of this example shows that the differences are less than 0.5%. Several comparisons of other sequences confirmed this error range, which indicates sufficient accuracy of sequence prediction for code generation purposes.

Prediction

schedule a)	average power [mW]	schedule b)	average power [mW]
	0.5 * 17.51		0.5 * 17.51
Nop		Nop	
	23.46		23.46
M = Mem[a]		M = Mem[a]	
	25.70		25.70
C = M		C = M	
	25.70		25.70
M = Mem[b]		M = Mem[b]	
	25.70		25.70
B = M		B = M	
	25.70		24.72
M = Mem[c]		Accu = B + C	
	25.70		28.08
A = M		M = Mem[c]	
	24.72		25.70
Accu = B + C		A = M	
	24.72		17.56
B = Accu		B = Accu	
	24.85		24.85
Accu = B * A		Accu = B * A	
	17.66		17.66
Accu = B * Accu		Accu = B * Accu	
	40.69		40.69
Mem[d] = Accu		Mem[d] = Accu	
	36.00		36.00
Nop		Nop	
	17.51		17.51
Nop		Nop	
	17.51		17.51
Nop		Nop	
	17.51		17.51
Nop		Nop	
	0.5 * 17.51		0.5 * 17.51
Σ =	390.64	Σ =	385.86
average power =	24.415 mW	average power =	24.116 mW
energy =	19.532 * 10^{-6}J	energy =	19.293 * 10^{-6}J

Simulation

energy =	19.614 * 10^{-6}J	energy =	19.382 * 10^{-6}J

Fig. 5: Energy consumption for different schedules

4 Genetic code generator (GCG)

The code generation process of GCG is started after the source program is transformed into a machine independent IR. At this level of abstraction the source program is given by a set of DFGs which are then separately mapped to assembly code. This is done by performing the subtasks IS, CS, and RA of code generation. One of the main problems is that there are strong interdependencies between these subtasks. For instance, finding an optimal instruction schedule does not necessarily result in optimal code with respect to all subtasks together. Hence, GCG is capable of revising (poor) decisions made in an early optimization step.
As we have seen in the last section, memory accesses have a more significant contribution to the energy consumption than other processor instructions. In order to minimize these power intensive instructions GCG uses a graph based code selection technique. Unfortunately, the problem of mapping graphs to optimal assembly code means solving an NP-hard problem. Thus, there is a need for an optimization algorithm capable of finding optimal or near optimal solutions in polynomial time.

Genetic algorithms (GA) have proven to solve complex optimization problems by imitating the natural evolution process (see e.g. [9, 2] for an overview). A population of a GA consists of several individuals, each of them representing a potential solution of 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 of 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.
An overview of the main steps of the optimization process of genetic algorithms is given in fig. 6.

At first all individuals of the population are initialized (1) and evaluated (2). Individuals which should inherit their genes to the next generation are selected probabilistically in the following step (3). The crossover operator (4) performs a recombination of the genetic information by choosing two individuals and swapping genes between these individuals. Afterwards mutation (5) creates new gene material by changing alleles. The resulting individuals are evaluated (6) again.

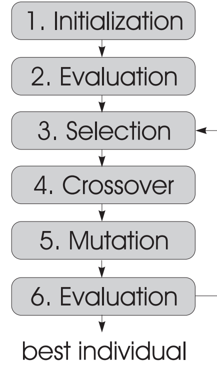

Fig. 6: Optimization steps of a GA

The optimization process is iterated until a termination condition (e.g. maximum number of generations or convergence) is met. It is a very important characteristic of genetic algorithms that suitable gene material is passed to the subsequent generations. This permits one to revise unfavorable decisions made in a previous optimization phase. For this reason genetic algorithms are adequate for solving non-linear optimization problems like phase-coupled code generation. However, one of the main problems using genetic algorithms is finding a suitable representation of the underlying optimization problem and using genetic operators which can be executed very quickly (because they are used several times). In the next section we first describe the coding mechanism and then the initialization, evaluation, crossover, and mutation steps of GCG in more detail.

Chromosomal representation

The goal of the chromosomal representation is to encode all information which is essential for code generation. We assume that the given source program is decomposed into a set of basic blocks, each represented by a DFG. In our approach the graph nodes of the intermediate representation are mapped to special genes of the chromosome (fig. 7) which are stored in a sequential order.

Thus, each gene of the chromosome represents an operation like a load or an addition. The values of a gene (allele) express all information which are necessary for code generation. These are for example used registers, performed processor instruction, execution cycle, and binding of arguments to special ports of the functional unit[4]. Swapped input variables are depicted in fig. 7 by crossing edges (inputs of the add node).

Initialization

The aim of the *initialization* is to establish an initial population from which the optimization process can be started. In

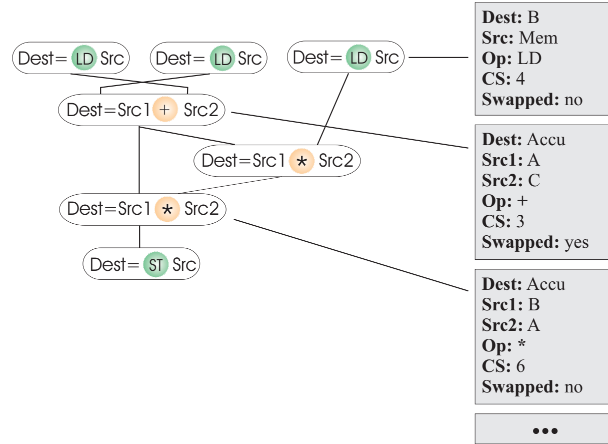

Fig. 7: DFG and chromosomal representation of a possible mapping to machine code

GCG the method of initializing an individual is based on the well-known list-scheduling algorithm [3]. But in contrast to traditional variants performing a heuristic selection we prefer a probabilistic selection of the next executable graph node. Doing this for all individuals of the population we obtain a set of different potential solutions.
The following steps are performed iteratively while there are graph nodes which have not been scheduled:

1. Select the next graph node to be scheduled (*instruction scheduling*).
2. Perform *code selection* for the scheduled graph node. At this time it is possible to take into account complex operations like MAC (multiply-add-accumulate) by pattern matching. If the selected graph node can be covered by more than one processor instruction we perform a probabilistical choice.
3. Determine source and destination locations (*register allocation*). This is done by determining the set of alleles whose selection leads to a valid solution. This set can be determined with knowledge of the actual set of registers in use, actual locations (registers or memory) of arguments and the set of registers which can be used by the processor instruction. If there is need for a data transport of an argument to the selected register or a need for spilling a variable the required instructions are inserted (and will lead to *additional cycles*).

Let $|V|$ and $|E|$ be the number of graph nodes and edges respectively. The complexity of this step is then $O(|V| + |E|)$.

Fig. 8 shows an example for an individual of the given DFG after performing the initialization step[5]. The encoded execution cycles are given by the relative execution order of the genes. Thus, inserted instructions which are not encoded on a separate gene (additional cycles) have no influence of the allocated time steps.

[4] Due to the restricted combination of the special-purpose registers in the data path it is very important for commutative operations to make use of the opportunity of swapping the input variables.

[5] In this representation we denote M = Mem followed by {A, B, C, D} = M as {A, B, C, D} = Mem.

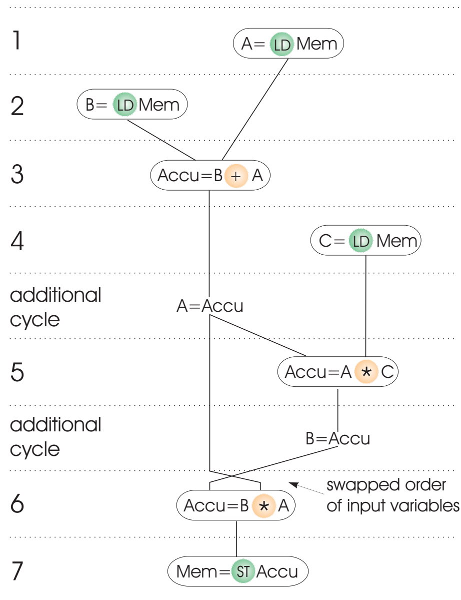

Fig. 8: Example of an initialized individual

Evaluation
The *fitness* function of a genetic algorithm represents the objective function of the underlying optimization problem. The individual with the highest fitness is the best solution. The fitness function has an essential impact on the optimization progress of the genetic algorithm because the fitness values serve as a basis for the subsequent selection step. Hence, GCG allows code generation according to different objectives by specifying a suitable fitness function:

- Minimization of execution time can be done by counting the number of execution cycles. A high number of cycles corresponds to low fitness.
- Low-energy optimization is possible by computing the energy consumption with respect to our instruction-level power model for the respective individual, or simply by counting the number of memory accesses.

Furthermore low-energy optimizations can be performed for given real-time constraints (e.g. maximum number of execution cycles of a basic block) by adding a penalty for every constraint violation. Thus, solutions not meeting the constraints will be assigned a lower fitness than others.

Crossover and mutation
The *crossover* operator deals with generating new individuals by probabilistically swapping genes between two selected individuals. In this case we perform a *uniform crossover* which means that every gene has the same probability to be exchanged. This step depends only on the number of graph nodes (genes) and can be done in $O(|V|)$. The result is two individuals consisting of recombined information of the parents. However, the large number of constraints (e.g. data dependencies, or resource constraints) which have to be handled can lead to invalid solutions. In order to avoid such invalid solutions we combine the subsequent mutation operator with a correctness check.
So, the main tasks of the *mutation* operator are to check the correctness of the actual allele and to generate the new gene material by changing alleles. Performing mutation for an individual is nearly the same task as initializing an individual. In analogy to the initialization step we determine a new allele of a gene by choosing an allele whose selection potentially leads to a valid solution. Thus, the complexity of the mutation step is $O(|V|+|E|)$.

5 Experimental results

Experimental results obtained with our M3 code generator are shown in table 2. The results refer to five real DSP routines (complex multiplication, IIR, lattice filter, FFT and FIR) as well as two pure test programs with large data flow graphs (DFG1, DFG2). Columns 2 and 3 give some characteristics about the number of common subexpressions in the DFGs and their uses, respectively.

Columns 4 and 5 show the number of instructions generated with the traditional tree based code generation technique vs. our graph based technique, while column 6 gives the relative improvement. As already explained in section 1, we compared to the tree based technique, since it is still common in compilers for DSPs with irregular register architectures and there is no reference compiler for the M3-DSP.

Columns 7–9 indicate the differences in the number of memory accesses between the tree and graph based techniques. The reduction in the number of memory accesses ranges between 18 and 58 %.

This is also reflected in the comparison of energy consumption of the generated machine programs in columns 10–12. As compared to the tree based technique, the graph based algorithm generates code with 18–36 % lower energy consumption. Note that this is not only due to the reduction of memory accesses, but also due to the dedicated instruction scheduling for low power.

Finally, columns 13–15 give the number of graph nodes in the test programs, the number of generations (= 20 * number of graph nodes) simulated in the genetic algorithm, as well as the CPU time requirements of our code generator on a 333 MHz Ultra-10 workstation. All results are generated by using a *steady-state* genetic algorithm with the following parameters[6]: population size: 30; number of individuals in the population to be replaced by the offspring: 10; mutation rate: 1/(number of graph nodes); crossover rate: 0.6.

In the worst case, of course, the energy savings obtained may be over-compensated by inaccuracies of the gate-level power simulation that forms the basis of our instruction-level power model. However, the predicted energy savings are typically larger than the expected inaccuracy of gate level power simulation (10–25 %). Therefore, in general real energy savings may strongly be expected. Ongoing work deals with measuring the real energy consumption of the generated machine code for an existing M3-DSP prototype chip.

[6]The implementation of the base genetic algorithm uses the genetic algorithm library PGAPack [17].

source	#CSEs	#CSE uses	#instructions tree	graph	%	#mem tree	graph	%	energy (10^{-6} [J]) tree	graph	%	#graph nodes	#gen	CPU[s]
cmult	4	8	23	15	35	10	6	40	35.06	26.63	24	12	240	3
IIR	3	7	34	24	29	16	10	38	52.65	36.39	31	19	380	8
lattice	8	16	55	34	38	24	10	58	84.03	53.81	36	22	440	13
FFT	10	26	76	48	37	36	22	39	109.75	76.57	30	32	640	29
FIR	15	30	156	128	18	78	64	18	238.01	196.29	18	80	1600	256
DFG1	14	43	127	78	39	56	28	50	181.89	116.97	36	41	820	57
DFG2	23	80	245	166	32	106	60	43	351.89	252.95	28	69	1380	192

Table 2: Comparison of tree based and graph based code generation results

6 Conclusions

This paper focuses on a relatively new topic: compilation for low energy consumption. For our driver application, the M3-DSP, we have first developed an instruction-level power model that has been shown to be very accurate (within 0.5 %) as compared to an underlying gate-level power simulation. Although there still may be inaccuracies w.r.t. the energy consumption of the real hardware, using such a high-level power model is very useful in compilers due to the need for efficient estimations. The main contribution of this paper is a phase-coupled, genetic algorithm based code generation technique that aims at minimizing energy consumption of the generated machine programs. This is achieved by reducing the number of memory accesses and a dedicated low-power scheduling technique. Experimental results indicate that this is a promising approach. Energy savings between 18 and 36 % as compared to a previous code generation technique have been observed, which is very significant for our target domain, mobile embedded DSP systems. This also justifies higher compilation times than in compilers for general-purpose systems. For demonstrating the basic technique of GCG we have focused on code generation for the special single slice mode of the M3-DSP. Future work will deal with exploitation of all data paths, optimizing global data transfers between basic blocks, extension of our power model to non-zero operand values and verifying the approach by physical measurements using an M3-DSP evaluation board.

References

[1] G. Araujo, S. Malik, and M. Lee. Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. In *33rd Design Automation Conference (DAC)*, 1996.

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

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

[4] S. Bashford and R. Leupers. Constraint driven Code Selection for Fixed-Point DSPs. In *36th Design Automation Conference (DAC)*, 1999.

[5] S. J. Beaty. *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Department of Mechanical Engineering, Colorado State University, Fort Collins, Colorado, Fall 1991.

[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] C. H. Gebotys. Low Energy Memory and Register Allocation Using Network Flow. In *Proc. 34th Design Automation Conference, Anaheim California USA*. ACM, June 1997.

[8] S. Hanono and S. Devadas. Instruction Selection, Resource Allocation, and Scheduling in the Aviv Retargetable Code Generator. In *Proceedings of the 35th DAC'98*, 1998.

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

[10] Synopsys Inc. *Power Products Reference Manual*, 1998.

[11] Texas Instruments. *TMS320C60 Instruction Set Manual*, 1997.

[12] D. Kästner and M. Langenbach. Integer Linear Programming vs. Graph-Based Methods in Code Generation. Technical Report Technical Report A/01/98., Universität des Saarlandes, 1998.

[13] M. Lee, V. Tiwari, S. Malik, and M. Fujita. Power Analysis and Low-Power Scheduling Techniques for Embedded DSP Software. In *Proceedings of the International Symposium on System Synthesis*, Sept. 1995.

[14] R. Leupers. Register Allocation for Common Subexpression in DSP Data Paths. In *Asia and South Pacific Design Automation Conference*, 2000.

[15] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction Selection Using Binate Covering for Code Size Optimization. In *Int. Conf. on Computer-Aided Design (ICCAD)*, 1995.

[16] E. Macii, M. Pedram, and F. Somenzi. High-Level Power Modeling, Estimation, and Optimization. In *Trans. on CAD of ICs and Systems*. IEEE, November 1998.

[17] *PGAPack Parallel Genetic Algorithm Library*. http://www-fp.mcs.anl.gov/CCST/research/reports_pre1998/-comp_bio/stalk/pgapack.html.

[18] J. M. Rabaey and M. Pedram, editors. *Low Power Design Methodologies*. Kluwer, 1996.

[19] K. Roy, A. Raghunathan, and S. Dey. VLSI-Design-Tutorial: 12th International Conference on VLSI Design: Low-Power Design Methodologies for Systems-on-Chips - Full Day Tutorial. January 1999.

[20] A. Sama, M. Balakrishnan, and J. F. M. Theeuwen. Speeding up Power Estimation of Embedded Software. In *Proceedings of International Symposium on Low Power Electronics and Design*, 2000.

[21] G. Sinevriotis and T. Stouraitis. Power Analysis of the ARM 7 Embedded Microprocessor. In *Proc. 9th Int. Workshop Power and Timing Modeling, Optimization and Simulation (PATMOS), Oct. 6-8 1999*, October 1999.

[22] C. Su, C. Tsui, and A. M. Despain. Low Power Architecture Design and Compilation Techniques for High-Performance Processors. In *IEEE COMPCON*, Februar 1994.

[23] V. Tiwari, S. Malik, and A. Wolfe. Compilation Techniques for Low Energy: An Overview. In *Proceedings of the 1994 IEEE Symposium on Low Power Electronics, San Diego*, October 1994.

[24] R. Wilhelm and D. Maurer. *Compiler Design*. Addison Wesley, 1995.

[25] T. Wilson, G. Grewal, B. Halley, and D. Banerjii. An Integrated Approach to Retargetable Code Generation. In *Proceedings of the 7th International Symposium on High-Level Synthesis*, 1994.

[26] T. Zeitlhofer and B. Wess. Operation scheduling for parallel functional units using genetic algorithms. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, 1999.