Code Selection for Media Processors with SIMD Instructions

Rainer Leupers*

Department of Computer Science 12
University of Dortmund
44221 Dortmund, Germany
email: Rainer.Leupers@cs.uni-dortmund.de

Abstract – Media processors show special instruction sets for fast execution of signal processing algorithms on different media data types. They provide SIMD instructions, capable of executing one operation on multiple data in parallel within a single instruction cycle. Unfortunately, their use in compilers is so far very restricted and requires either assembly libraries or compiler intrinsics. This paper presents a novel code selection technique capable of exploiting SIMD instructions also when compiling plain C source code. It permits to take advantage of SIMD instructions for multimedia applications, while still using portable source code.

1. Introduction

In order to support the fast execution of computation-intensive multimedia application programs, dedicated *media processors* are available on the semiconductor market. These machines provide architectural support for efficiently processing different data types on the same data path. Examples are the Texas Instruments C62xx, the Philips Trimedia, and Intel's Pentium MMX architecture.

Many media processors show a 32-bit data word length. However, applications in the audio or video domain normally require only a precision of 16 or 8 bits, respectively, resulting in a potential waste of computational resources. Therefore, media processors show a special kind of machine instructions, that permit to virtually split each full data register into multiple *subregisters* and to perform identical computations on the subregisters in parallel. These instructions are now commonly called *SIMD (single instruction, multiple data) instructions*. SIMD instructions are very powerful for computations on media data. However, a major problem with SIMD instructions is the missing support by C compilers. Standard code generation techniques are not capable of detecting opportunities for SIMD instructions due to a limited exploration of the search space. Using plain C code for programming media processors thus potentially results in huge losses in code quality with respect to code size, performance and/or power consumption.

One way to circumvent this problem is the use of "compiler intrinsics", i.e. calls to compiler-known functions which are expanded into specific assembly instructions. Another method is the use of hand-optimized assembly libraries. Unfortunately, both methods result in highly machine-specific code, so that porting an application to a new target processor requires significant programming effort.

The contribution of this paper is a new code selection technique, capable of exploiting SIMD instructions when compiling plain ANSI C code. This allows to take full advantage of the media processor capabilities while still using machine-independent source code. The paper is structured as follows. The next section mentions previous work on code selection for embedded processors. After exemplifying the use of SIMD instructions in section 3, we explain the details of the proposed code selection technique in sections 4 and 5. Experimental results for existing media processors are given in section 6.

2. Related work

Most compilers use tree pattern matching with dynamic programming [1] for code selection. This technique uses an intermediate program representation consisting of *data flow trees* (DFTs). The problem of code selection is mapped to a problem of covering DFTs by available instruction patterns. Tree pattern matching with dynamic programming is capable of computing an optimal covering of each DFT linear time. It is used in code generator generators like "twig" and "olive" [1] which generate code selectors from a tree grammar description of the machine instruction set. A tree grammar consists of terminals, nonterminals (including a designated start symbol), and tree-like, cost-attributed rules. Rules are used to describe the behavior of assembly instructions, while rule costs induce a metric such as the execution time of an instruction.

However, tree pattern matching with dynamic programming is not directly applicable to generation of SIMD instructions, since this in general requires to simultaneously cover multiple DFTs, instead of processing one DFT after another. This means that code selection has to be performed on full data-flow graphs (DFGs) instead of only DFTs as in traditional compiler technology. This will become obvious in the next section, which exemplifies the use of SIMD instructions.

Some work on code selection for DFGs has also been performed in the DSP area [2, 3, 4, 5]. Such techniques have been designed to cope with the irregular data path structure of DSP processors. However, code selection with SIMD instructions has so far not been addressed. As a result, current compilers for media processors cannot directly exploit SIMD instructions when compiling C source code.

*This work has been supported by Agilent Technologies, USA. Publication: DATE 2000,

3. SIMD instructions

We call an instruction a SIMD instruction, if it performs a manipulation (arithmetic or logic operation, load or store) of data stored subregisters instead of full registers. For the use of SIMD instructions, the 32-bit data registers are considered to be composed of either two 16-bit subregisters or four 8-bit subregisters. Thus, in terms of the C programming language, any full register may store either four "char" data, two "short" data, or a single "int" at a time. In the following, for sake of simplicity, we will emphasize SIMD instructions on 16-bit data, although (as outlined later) the proposed technique equally applies to 8-bit data.

Fig. 1 gives an example of the SIMD instruction "ADD2" of the TI C62xx. It performs two 16-bit additions in parallel and writes two results into the two subregisters of the destination register. While arithmetic SIMD instructions require

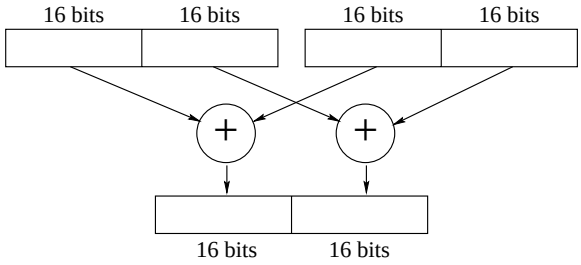

Figure 1. *TI C62xx instruction "ADD2"*

special hardware support, such as the suppression of carry propagation, there are also "trivial" SIMD instructions like those performing logic operations (AND, OR, XOR, NOT).

In order to take full advantage of SIMD instructions, it is necessary, that the 16-bit or 8-bit data to be manipulated are efficiently loaded from and stored into memory. Under certain conditions, one can use 32-bit instructions to load operands and store results of SIMD instructions. As an example, consider the piece of C code in fig. 2, which describes a vector addition on short data. In this example, the loop body

```
void f(short* A,short* B,short* C)
{ int i;
  for (i = 0; i < N; i += 2)
  { A[i]   = B[i] + C[i];
    A[i+1] = B[i+1] + C[i+1];
  }
}
```

Figure 2. *Source code for vector addition*

has been unrolled once, so as to reveal the potential parallelism.

Using the above "ADD2" instruction, the two additions in the loop body could be executed in parallel. However, this requires that the operand pairs B[i], C[i] and B[i+1], C[i+1] are loaded into the lower and upper halves of the argument registers, respectively[1]. Therefore, B[i] and B[i+1] must be loaded by a single 32-bit load instruction instead of two separate 16-bit loads, and the same applies to C[i] and C[i+1]. Since adjacent array elements are stored in adjacent memory locations, this can be accomplished with two 32-bit load instructions. After execution of "ADD2", the results A[i] and A[i+1] are located in the lower and upper halves of the destination register, and a single 32-bit store operation suffices to write the two results back to memory. This is illustrated in fig. 3. In total, the number of instructions required to execute the vector addition can be reduced by 50 % when using SIMD instructions.

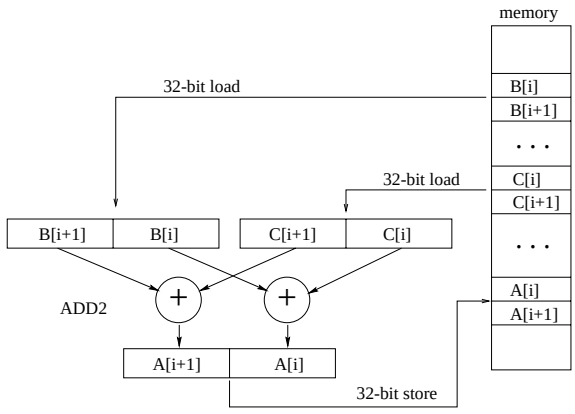

Figure 3. *Parallelization of vector addition with SIMD instructions*

There are two major difficulties in exploiting SIMD instructions. First, parallel loading or storing of values located in subregisters from/to memory requires to establish that the memory address difference is correct. In our C compiler, we apply a standard data flow analysis technique to pointers in order to determine those sets of operands that qualify for parallel loading and storing with 32-bit instructions.

A more difficult task is to correctly pack potentially parallel instructions together during code generation, so as to form SIMD instructions. Generating SIMD instructions on-the-fly only during the instruction scheduling and register allocation phases, although possible, would be very difficult, because a large number of constraints need to be obeyed. If, for instance, multiple values share a single 32-bit register, then their live ranges are tightly coupled. As a consequence, standard register allocation techniques, such as graph coloring [6], cannot be applied.

Instead, we prefer to generate SIMD instructions already early in the code generation process during the *code selection phase*, which maps the machine-independent intermediate representation of a program into machine-specific instructions. The generated code afterwards only operates on symbolic 32-bit registers, so that existing instruction scheduling and register allocation techniques can still be used.

4. DFG covering

Our code selection technique operates on data flow graph (DFG) representations of basic blocks. Using full DFGs (instead of separate DFTs) is necessary, since exploitation of

[1] On some processors this requires a memory alignment to word boundaries. We assume that an appropriate alignment can be ensured, either by the source code itself or by assembler directives.

SIMD instructions frequently requires to pack together operations located in different DFTs. Each DFG node n represents an operation (arithmetic, logic, load, or store), a constant, or a variable, and each DFG edge (n, m) denotes a data dependence between nodes n and m. Note that a DFG in general may contain common subexpressions (CSEs) (nodes with fanout greater than one) and may consist of several non-connected subgraphs. A given DFG is partitioned into multiple DFTs by cutting the DFG at the CSE edges and computing optimal covers for each single DFT. Since media processors tend to show a regular data path architecture, this does not incur significant losses in code quality. However, this "traditional" approach is not directly capable of generating SIMD instructions, since this in general requires the consideration of multiple DFTs at a time.

We overcome this problem by permitting the generation of *alternative solutions* during tree pattern matching. Instead of annotating only a single optimal rule to each DFG node, we annotate all optimal rules, including those for SIMD instructions, and only later determine the best rules globally for the whole DFG. In order to achieve this, we introduce dedicated nonterminal symbols in the tree grammar, which denote the different possibilities of using a register, i.e. either as a full 32-bit register or as two separate 16-bit registers. As an example, consider instructions for addition on a C62xx processor (all other arithmetic and logic SIMD instructions are modeled in the same way). Instruction "ADD" adds two 32-bit registers and also writes the result to a 32-bit register. This can be expressed by the following rule, where nonterminal symbol "reg" denotes a full register and "PLUS" is a terminal symbol.

```
reg: PLUS(reg,reg)
```

The SIMD instruction "ADD2" (fig. 1) simultaneously performs two 16-bit additions. We use two separate rules for modeling the behavior of "ADD2":

```
reg_lo: PLUS(reg_lo,reg_lo)
reg_hi: PLUS(reg_hi,reg_hi)
```

The nonterminals "reg_lo" and "reg_hi" denote the lower and upper 16-bit subregisters of a full register. Both rules are assigned the same cost value as the 32-bit version. As a consequence, there exist three alternative optimal covers for all DFG nodes representing a PLUS operation. Note that the rule costs for SIMD instructions are not counted twice. Rule costs are only considered during the DFG covering phase in order to obtain alternative optimal DFT covers. During the subsequent code selection phase (section 5), which aims at maximizing the use of SIMD instructions, the rule costs are no longer required.

As mentioned earlier, 32-bit load/store instructions can also be used to simultaneously load/store two 16-bit values. For instance, a 32-bit load into a full register from an address pointed to by some other register is described by the rule

```
reg: LOAD_INT(reg)
```

Similar to the above "ADD2" instruction, we use two additional rules to describe the use of a 32-bit load for SIMD instructions:

```
reg_lo: LOAD_SHORT(reg)
reg_hi: LOAD_SHORT(reg)
```

Note that all rules representing operations on subregisters must not be considered as "stand-alone" instructions, since this would result in invalid code. The constraint system presented in section 5 ensures, that such rules can only be used for covering *pairs* of DFG nodes, in which case only a single SIMD assembly instruction is emitted.

The overall DFG covering process works as follows. The DFG is partitioned into DFTs by assigning each CSE to a symbolic register and replacing all uses of CSEs by read operations on that register. Then, all DFTs are separately covered by means of tree pattern matching with dynamic programming. We use a modification of olive [1] to generate the required tree pattern matcher from an instruction set description. Since olive in the original version only computes a single optimal solution (with ties broken arbitrarily) for each DFT and thus only annotates a single rule at each DFT node, we have modified olive in such a way, that alternative optimal covers are retained during DFT covering. Our modified olive version annotates *all* minimum cost derivations *for each nonterminal* at the DFT nodes. Whether or not SIMD instructions can be selected is decided only later globally for the entire DFG.

5. Code selection

After DFG covering, the actual code selection phase determines the detailed DFG node covers to be selected from the available alternatives. In this phase, the goal is to maximize the use of SIMD instructions across the entire DFG. We solve this problem by transforming the code selection problem into an Integer Linear Program (ILP) formulation. For each DFG node n_i, the DFG covering phase returns a set of alternative rules $R(n_i)$, which match n_i at minimum costs. We use Boolean variables x_{ij} to express that node n_i is (or is not) covered by rule $r_j \in R(n_i)$. A valid code selection requires that each node is covered by exactly one rule. Therefore, for each node n_i, we impose the constraint

$$\sum_{r_j \in R(n_i)} x_{ij} \quad = \quad 1$$

Selecting a certain rule r_j for some node n_i has implications on the covering of its children nodes in the DFT. If, for instance, node n_i is covered by rule

```
reg_lo: PLUS(reg_lo,reg_lo)
```

then it must be ensured that the first and second child of n_i are derived to nonterminal "reg_lo", i.e., the arguments of the PLUS operation reside in lower 16-bit subregisters. More generally, let $r_j \in R(n_i)$ be the rule selected for node n_i, let n_k be the m-th child of n_i in a DFT, and let $r_l \in R(n_k)$ be the rule selected for n_k. Since n_k is the m-th child of n_i, the nonterminal on the left hand side (LHS) of r_l must be equal to the m-th nonterminal, say nt_m, on the right hand side of r_j. Let $R_m(n_k) \subset R(n_k)$ denote the set of rules r_l for n_k, such that LHS(r_l) = nt_m. Then, the following constraint expresses the dependence between n_i and n_k:

$$x_{ij} \quad \leq \sum_{r_l \in R_m(n_k)} x_{kl}$$

The next class of constraints concerns code selection for common subexpressions (CSEs) in the DFG. As already mentioned, each CSE is strictly assigned to a register, and we insert register read/write nodes (using dedicated grammar terminals) into the DFG so as to replace the CSE edges. Since we are dealing with general-purpose registers, it is not necessary to commit to certain physical registers during code selection, but the use of symbolic registers, which can be mapped to physical registers only later, is sufficient. However, there still may be alternatives for storing 16-bit "short" CSEs, since these may reside in either full registers or subregisters. This should not be neglected, since SIMD instructions can sometimes also be exploited for parallel computation of CSEs. In our tree grammar model, 16-bit CSEs can be written to and read from either "reg", "reg_lo", or "reg_hi". The corresponding rules are ("S" is the grammar start symbol):

```
S: WRITE_SHORT_CSE(reg)     /* r1 */
S: WRITE_SHORT_CSE(reg_lo)  /* r2 */
S: WRITE_SHORT_CSE(reg_hi)  /* r3 */

reg:    READ_SHORT_CSE      /* r4 */
reg_lo: READ_SHORT_CSE      /* r5 */
reg_hi: READ_SHORT_CSE      /* r6 */
```

A correct code selection requires that the locations for a CSE definition and its uses are identical across the entire DFG. If, for instance, rule r_2 is selected for some node n_i defining a CSE, then all uses n_j of that CSE have to be covered by rule r_5. Conversely, the selection of r_5 for n_j enforces the selection of r_2 for n_i. In general, for any "short" CSE definition/use pair (n_i, n_j) the following three constraints[2] have to be specified:

$$x_{i1} = x_{j4}, x_{i2} = x_{j5}, x_{i3} = x_{j6}$$

Another class of constraints ensures a valid packing of instructions to SIMD instructions. For this purpose, we introduce the notion of *SIMD pairs*. A pair (n_i, n_j) of DFG nodes is called a SIMD pair, if the following conditions are satisfied:

- There is no scheduling precedence between n_i and n_j
- n_i and n_j have the same operator
- According to the tree grammar rules, n_i may be located in an upper subregister and n_j may be located in a lower subregister.
- If n_i and n_j represent load or store operations of 16-bit values, where a_i and a_j are the corresponding memory addresses, then the difference $a_j - a_i$ equals the number of memory words occupied by a 16-bit value (e.g. 2 for a byte-addressable memory).

The latter condition ensures, that parallel loads and stores of subregisters implemented by SIMD instructions actually refer to adjacent data in memory.

The set P of all SIMD pairs can be computed from the information generated by DFG covering. The required runtime is quadratic in the number of DFG nodes. Any DFG node contained in a SIMD pair can potentially be mapped to a SIMD instruction. However, it must be guaranteed that any selected SIMD instruction actually covers a *pair* of DFG nodes and that any DFG node is covered by *at most* one SIMD instruction. In order to express these conditions in terms of ILP constraints, we introduce one auxiliary Boolean variable y_{ij} for each SIMD pair (n_i, n_j). The setting of $y_{ij} = 1$ denotes that n_i and n_j are packed into a single SIMD instruction, i.e., n_i operates on the upper subregister and n_j operates on the lower subregister of the same full register.

For any n_i let $R_{hi}(n_i) \subset R(n_i)$ and $R_{lo}(n_i) \subset R(n_i)$ denote the sets of rules for n_i operating on an upper or a lower subregister, respectively. If n_i is covered by some rule in $R_{hi}(n_i)$, then there must be a node n_j, such that $(n_i, n_j) \in P$, and n_j is covered by a rule in $R_{lo}(n_j)$. Conversely, if n_i is covered by some rule in $R_{lo}(n_i)$, then there must be a node n_j, such that $(n_j, n_i) \in P$, and n_j is covered by a rule in $R_{hi}(n_j)$. For any n_i contained in a SIMD pair, this is modeled by two constraints:

$$\sum_{j:(n_i,n_j)\in P} y_{ij} = \sum_{r_k \in R_{hi}(n_i)} x_{ik}$$

$$\sum_{j:(n_j,n_i)\in P} y_{ji} = \sum_{r_k \in R_{lo}(n_i)} x_{ik}$$

Since the right hand sides of the equations are always less or equal to 1, it is also guaranteed, that any node n_i is packed into at most one SIMD instruction.

The last class of constraints is required for avoiding code selection decisions leading to scheduling deadlocks. For any DFG node n_i, let $pred(n_i)$ denote the set of nodes that must be scheduled before n_i (e.g. due to data or output dependence), and let $succ(n_i)$ be the set of nodes to be scheduled after n_i. Whenever a SIMD pair (n_i, n_j) is covered by a SIMD instruction I_1, and there is a SIMD pair $(n_k, n_l) \in P$ with $n_k \in succ(n_i)$ and $n_l \in pred(n_j)$, or vice versa, then it must be ensured, that n_k and n_l are *not* packed into another SIMD instruction I_2. Otherwise, the resulting code could not be scheduled, since I_2 would need to be executed both before and after I_1. For any SIMD pair (n_i, n_j), let $X_{ij} \subset P$ denote the set of SIMD pairs (n_k, n_l), such that $n_k \in succ(n_i)$ and $n_l \in pred(n_j)$, or $n_k \in pred(n_i)$ and $n_l \in succ(n_j)$. Then, for (n_i, n_j) and any $(n_k, n_l) \in X_{ij}$, we specify the following constraint to avoid scheduling deadlocks:

$$y_{ij} + y_{kl} \leq 1$$

For an optimized code selection under the above correctness constraints, the number of selected SIMD instructions must be maximized across the entire DFG G. For any node n_i, let $S(n_i) = R_{hi}(n_i) \cup R_{lo}(n_i) \subset R(n_i)$ denote the subset of rules for n_i operating on a subregister. Then, we maximize the following objective function:

$$f = \sum_{n_i \in G} \sum_{r_j \in S(n_i)} x_{ij}$$

This task can be performed with any ILP solver. For our experiments (section 6) we have used "lp_solve" [9]. The 0/1 binding of the x_{ij} solution variables accounts for the detailed

[2] These can also be replaced by unifying the variables on the left and right hand sides in the ILP.

code selection and thus allows to emit assembly code for the DFG.

The code selection technique described above can be easily scaled to SIMD instructions operating on 4 subregisters of 8 bit each. The necessary changes mainly concern the nonterminals for subregisters and the definition of ILP variables. For instance, we have to use *SIMD quadruples* instead of SIMD pairs, and the decision variables y_{ij} have to be replaced by four-index variables y_{ijkl}. Naturally, due to the larger number of variables, code selection for 8-bit subregisters requires more runtime than in the case of 16 bits.

6. Experimental results

Using the techniques described above, we have implemented code selectors for the Texas Instruments C62xx and the Philips Trimedia TM1000. We have compiled ANSI C source codes into assembly code for several signal processing kernel routines, which mainly consist of one finite loop. "vector add" is the example from fig. 2, "image compositing" is taken from [8], and the remaining sources are from the DSPStone suite [7]. In order to ensure that exploitation of SIMD instructions takes place without machine-specific source code constructs, the sets of C source codes were *identical* for both target processors.

source	type	unroll	no SIMD	SIMD	CPU sec
TI C62xx					
vector add	short	1	8	4	0.7
IIR filter	short	0	21	17	2.9
convolution	short	1	8	6	0.6
FIR filter	short	1	15	11	0.9
N complex updates	short	1	20	16	3.0
image compositing	short	1	14	11	3.1
Trimedia TM1000					
vector add	short	1	8	4	0.7
IIR filter	short	0	22	22	5.1
convolution	short	1	8	8	0.9
FIR filter	short	1	15	9	0.9
N complex updates	short	1	20	20	4.7
image compositing	short	1	14	7	3.2
vector add	char	3	16	4	5.0
FIR filter	char	3	36	18	26.5

Table 1. *Experimental results*

The experimental results for the TI C62xx are listed in the upper part of table 1. The unrolling factor specifies the number of duplications of the loop body, which is necessary to exhibit enough parallelism for exploitation of SIMD instructions. Columns 4 and 5 give the number of generated machine instructions for the loop body without and with exploitation of SIMD instructions. Column 6 mentions the required CPU time (Sun Ultra-1, including both DFG covering and ILP solving) when using SIMD instructions.

The TI C62xx shows a comparatively limited support for SIMD instructions, essentially parallel additions and subtractions on 16-bit subregisters. Therefore, all experiments have been carried out with "short" data types. The maximum reduction in the instruction count (50 %) was obtained for the "vector add" example, since using the C62xx SIMD instructions permits to unroll the loop once without increasing the code size. The lower part of table 1 shows the corresponding results for the Trimedia architecture. While for some source codes, such as the "IIR filter" and "convolution", SIMD instructions were not applicable, the code quality gains for "FIR filter" and "image compositing" were more significant as compared to the C62xx. This is due to the more powerful SIMD capabilities of the Trimedia (e.g. special instructions for FIR computations), which become particularly obvious for certain algorithms on 8-bit data. As shown for the "vector add" and "FIR filter" examples, the use of SIMD instructions for 8-bit "char" data types results in a reduction of instruction count of 75 % and 50 %, respectively.

Even though we partially use ILP for code selection, the runtime consumed by our approach is moderate if the DFGs to be compiled are not too large. The largest example (FIR filter on char data), whose DFG comprises 95 nodes, took 26.5 CPU seconds. We believe that this is acceptable for embedded applications and systems-on-a-chip, where code quality is of higher concern than compilation speed. Exhaustive compilation times can be avoided (possibly at the expense of lower code quality) by specifying a threshold value for the maximum size of DFGs passed to the code selector at a time.

7. Conclusions

SIMD instructions are so far not really exploited by compilers for media processors. Taking advantage of such instructions is only possible, if processor-specific assembly routines or compiler intrinsics are used, resulting in low portability of software. The presented code selection technique is capable of exploiting SIMD instructions without the need for processor-specific code. Our approach builds on the classical tree-based code selection paradigm, but it generates alternative covers. The detailed code selection is performed only later, when enough information for generation of SIMD instructions for an entire data flow graph is available. The applicability has been demonstrated experimentally by compiling the same set of C sources for two different media processors and exploiting SIMD instructions in both cases.

References

[1] A.V. Aho, M. Ganapathi, S.W.K Tjiang: *Code Generation Using Tree Matching and Dynamic Programming*, ACM Trans. on Programming Languages and Systems 11, no. 4, 1989, pp. 491-516

[2] S. Liao, S. Devadas, K. Keutzer, S. Tjiang: *Instruction Selection Using Binate Covering for Code Size Optimization*, Int. Conf. on Computer-Aided Design (ICCAD), 1995, pp. 393-399

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

[4] R. Leupers, P. Marwedel: *Instruction Selection for Embedded DSPs with Complex Instructions*, European Design Automation Conference (EURO-DAC), 1996

[5] S. Bashford, R. Leupers: *Constraint Driven Code Selection for Fixed-Point DSPs*, 36th Design Automation Conference (DAC), 1999

[6] G.J. Chaitin: *Register Allocation and Spilling via Graph Coloring*, ACM SIGPLAN Symp. on Compiler Construction, 1982, pp. 98-105

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

[8] A. Peleg, S. Wilkie, U. Weiser: *Intel MMX for Multimedia PCs*, Comm. of the ACM, vol. 40, no. 1, 1997

[9] Eindhoven University of Technology: ftp.es.ele.tue.nl/pub/lp_solve/