J Sign Process Syst (2016) 84:435–446
DOI 10.1007/s11265-015-1081-6

Improving Code Density with Variable Length Encoding Aware Instruction Scheduling

Heikki Kultala[1] · Timo Viitanen[1] · Pekka Jääskeläinen[1] · Janne Helkala[1] · Jarmo Takala[1]

Received: 30 January 2015 / Revised: 21 July 2015 / Accepted: 30 October 2015 / Published online: 7 December 2015

Abstract Variable length encoding can considerably decrease code size in VLIW processors by reducing the number of bits wasted on encoding *No Operations(NOPs)*. A processor may have different instruction templates where different execution slots are implicitly NOPs, but all combinations of NOPs may not be supported by the instruction templates. The efficiency of the NOP encoding can be improved by the compiler trying to place NOPs in such way that the usage of implicit NOPs is maximized. Two different methods of optimizing the use of the implicit NOP slots are evaluated: (a) prioritizing function units that have fewer implicit NOPs associated with them and (b) a post-pass to the instruction scheduler which utilizes the slack of the schedule by rescheduling operations with slack into different instruction words so that the available instruction templates are better utilized. Three different methods for selecting basic blocks to apply FU priorization on are also analyzed: always, always outside inner loops, and only outside inner loops only in basic blocks after testing where it helped to decrease code size. The post-pass optimizer alone saved an average of 2.4 % and a maximum of 10.5 % instruction memory, without performance loss. Prioritizing function units in only those basic blocks where it helped gave the best case instruction memory savings of 10.7 % and average savings of 3.0 % in exchange for an average 0.3 % slowdown. Applying both of the optimizations together gave the best case code size decrease of 12.2 % and an average of 5.4 %, while performance decreased on average by 0.1 %.

Keywords Code density · Variable length instructions · vliw · tta · Instruction scheduling · Code optimization · Instruction templates

✉ Heikki Kultala
heikki.kultala@tut.fi

Timo Viitanen
timo.2.viitanen@tut.fi

Pekka Jääskeläinen
pekka.jaaskelainen@tut.fi

Janne Helkala
janne.helkala@gmail.com

Jarmo Takala
jarmo.takala@tut.fi

[1] Tampere University of Technology, Tampere, Finland

1 Introduction

One of the main pitfalls of *Very Long Instruction Word (VLIW)* and *Transport Triggered Architectures (TTAs)* [1] is the poor code density which is caused by the long instruction word and *No Operation (NOP)s* that have to be inserted into the program code. A VLIW or TTA processor has several execution units in order to achieve high performance on computationally intensive inner loops, often with aid of unrolling and software pipelining. Each of the execution units has an execution slot in the wide instruction word. Often there is a large amount of control-oriented code outside the inner loops which cannot exploit the parallel execution units, and is, therefore, represented with instructions where most of the execution slots are NOPs. In a straightforward encoding, this helper code consumes a large amount of instruction memory.

In order to save instruction memory and fetch power, control code should be encoded in a more compact form than full-size VLIW instructions. One common approach

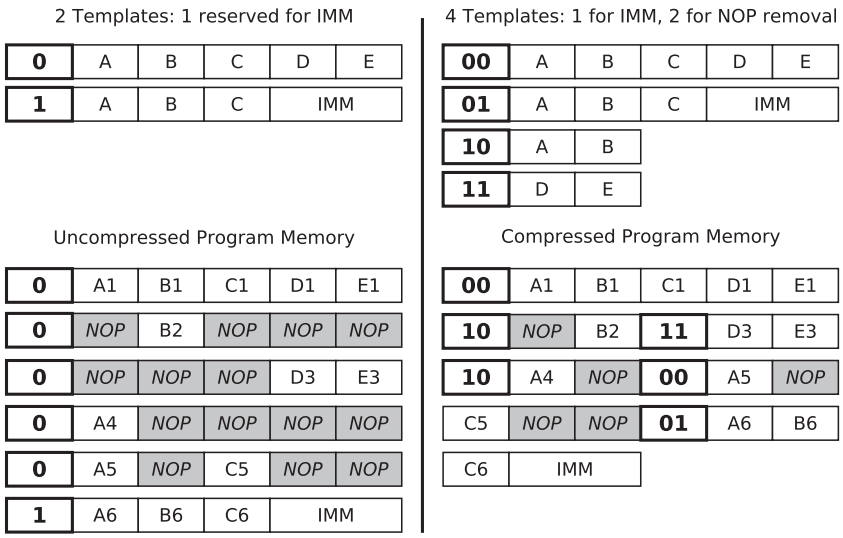


Figure 1 A short program before (*left*) and after (*right*) assigning two new instruction formats, which define two execution slots to be used out of the five in the processor. Most of the NOP operations are removed by using the shorter instruction formats in the 2nd, 3rd and 4th instruction. IMM means a long immediate value which is transferred to the data path of the processor by encoding it into two execution slots of an instruction word in a instruction template where these two executions slots cannot be used for ordinary operations.

is to structure instructions in a series of operations, with, e.g., *start* and *stop* bits to delimit operations that can be executed in parallel. Dynamic hardware then detects bundles and routes operations to appropriate execution units. Conte et al. discuss I-fetch and cache design tradeoffs in this type of encoding [2]. However, the hardware to support any combination of NOPs in each instruction is complex. Other processors simplify the decode apparatus by specifying a limited number of instruction templates: each template specifies some execution slots as implicit NOPs that do not consume space, but other NOPs have to be explicitly encoded. In a reasonable implementation, there are only few different instruction templates. [3, 4]

An example of template selection and NOP removal for a 5-issue processor is shown in Fig. 1. In this example, a large amount of NOPs is seen in four instruction words. Two new instruction formats are assigned to the templates *'10'* and *'11'*, which only use the execution slots A, B and D, E. The rest of the execution slots in these two formats are considered as NOP slots. If NOPs are seen in the NOP slots, they are removed from the instruction. Three instructions can be encoded in these templates to remove a majority of the NOP operations in the example program. Considerable instruction memory savings can be achieved by simply scheduling the instructions for the maximum performance without optimizing the code for the implicit NOP slots and simply using the shorter instructions when the NOP patterns match. This is, however, suboptimal, as the usage of the short instructions can be increased by compiler optimizations.

In this paper, two solutions to this are presented and compared: (a) prioritizing function units that have implicit NOP slots associated with them in fewer instruction templates and (b) executing a post-scheduler NOP-optimizer which utilizes slack of the schedule. In post-scheduler approach, operations, which have slack, are moved spatially into different instruction words so that the available instruction templates are better utilized to their maximum capacity. Prioritizing function units may decrease the performance of the code as this may conflict with other, more performance-critical methods of function unit priorization. The post-scheduler optimizer should have minimal effect on performance.

We have already reported some preliminary results in [5] and, in this paper, we propose two additional improvements: performing prioritization only for basic blocks outside inner loops so as to decrease the performance penalty, and speculatively scheduling basic blocks with both heuristics and selecting the one with the better code density. These improvements make FU priorization a viable optimization in practise, unlike the original algorithm which uses same heuristics for all basic blocks in the program. Also some of the technical content is presented in more detail. In addition, an advanecd version of the instruction scheduler is used which explains the differences in some results compared to [5].

This paper is organized as follows. Section 2 discusses previous research related to the proposed methods. Subsection 3.2 explains the function unit selection heuristics. Subsection 3.3 introduces the proposed post-scheduler optimizer algorithm. Section 4 contains evaluation of the two proposed methods. Section 5 discusses potential future research related to the topic. Section 6 concludes the paper.

2 Related Work

Lee et al. [6] introduce a post-scheduler optimization algorithm to minimize the instruction fetch and control logic transitions between successive instructions. In this method,

horizontal and vertical rescheduling of operations is performed, moving operations both between instructions and between execution slots in the same instructions. Their method, however, does not consider the NOP usage and does not try to optimize the code size, but their work has been an inspiration to the post-optimizer pass presented in this paper.

Hahn et al. [7] propose a variable-length encoding for VLIW. This method has "protected" versions of many long-latency operations and control operations. These versions of the operations add pipeline stalls after the operations, so that there is no need to add subsequent instruction words containing only NOPs. Their instruction scheduler fills the delay slots and instruction words after a long-latency operation with usable instructions, and uses the ordinary version of the operation if possible, to maximize performance. In case the scheduler cannot place any useful operations in delay slots or instructions after some long-latency operation, it replaces the operation with the "protected" version of the operation. When optimizing for minimal code size, the compiler always uses the protected versions of instructions, resulting in lower performance but eliminating all NOPs due to delay slots and long-latency operations.

In [8], a method of collapsing the prolog and the epilog of software pipelined loop is introduced. This optimization can be combined with the proposed methods as they attack different parts of the code size problem.

The approach of Jee et al. [9] eliminates many NOP operations by encoding only data dependencies and enabling different execution slots to execute operations from different instruction words. This, however, requires considerable changes to the processor architecture and adds complexity to the processor's control logic, and requires some support for dynamic scheduling in the processor.

Some studies approach the code density problem at a lower level by compressing code with, e.g., Huffman coding. Ros et al. [10] propose compiler optimization passes that improve the compression ratio by reassigning registers. Larin and Conte [11] have the compiler generate an optimal Huffman coding and decompression hardware to go with each program. These approaches might be applied on top of a template-based instruction format.

Haga et al. [12] introduce a method to minimize code size with global scheduling. Their approach, however, does not consider optimizing the code size for variable-length instructions.

Haga et al. [13] discuss compiler optimizations for *explicitly parallel instruction computing (EPIC)* architectures. EPIC is a variation of VLIW where instructions divided into regions that can be executed in parallel by means of conceptual stop bits. They are fetched in fixed-size bundles, each of which contains a template field indicating the placement of stop bits within each bundle. Parallel execution groups can span multiple bundles. Haga et al. introduce an algorithm to create schedules which are optimal in both performance and code size for infinitely wide EPIC processors which are never resource constrained. The optimal algorithm becomes slow when large basic blocks are scheduled. The authors propose non-optimal heuristics to overcome this problem. The instruction templates in EPIC architectures are, however, quite different than the variable-width templates discussed in this work and their method can be only used for EPIC-type instruction templates.

Most of the related work [7, 9–11, 13] concentrates on instruction encodings which are more complex than simple variable-length instruction templates, and the techniques cannot be applied on simple variable-length templates. Some of the related work [8, 12] proposes code size optimizations generic enough to be used together with the techniques proposed in this paper.

3 Proposed Optimizations

3.1 Baseline

In this work, the cycle-based list scheduler [14] is used as baseline and the proposed optimizations are compared against this method. The scheduler used in this paper schedules instructions one basic block at time, but also includes a postpass delay slot filler which performs some inter-basic block code motion. The baseline method schedules operations to the best possible cycle but if multiple execution units can execute the operation in the same cycle, the function unit with least operations that can execute the operation is selected. If there are multiple units with the same number of operations, the connectivity of the function unit is considered and the unit with the least connectivity is selected. The rationale for this is that if, e.g., addition can be executed on both function units A and B, but only unit A supports multiplication, add operations should be preferentially scheduled to unit B, so that unit A is be free to execute the multiplications. The optimizations could also be used with different instruction schedulers.

3.2 Function Unit Selection

A *prioritize NOP-slots* option was added to the function unit selection. This option works by calculating on how many instruction templates a function unit can be encoded as an implicit NOP, and deprioritizing those function units with the highest implicit NOP count. If two or more function units have the same NOP slot value, then the instruction scheduler reverts to the old performance-optimized mechanism when selecting between those function units.

We experiment with three modes of operation for the function unit selection optimization. The first mode schedules all code with the size-optimized function unit heuristic. The second mode uses the size-optimized function unit heuristic only for code outsize inner loops. The third mode is to compile every non-inner-loop basic block with both heuristics and select the code that has the smaller size, as sometimes the original heuristic may generate denser code. Inner loops are scheduled with the baseline heuristics to minimize performance degradion just like in the second mode. Even the third mode can result a small performance degradation as code outside inner loops can simultaneously grow larger in terms of instruction count while actually decreasing in code size.

The idea in the second mode is to minimize the execution time penalty of the optimization by not performing it in the parts of the code that are executed the most. There is sufficient code outside inner loops to still allow considerable size savings.

3.3 Post-scheduler Optimizer Algorithm

The main motivation of the post-scheduler optimizer is to make better use of NOP slots without decreasing performance; decisions taken during the actual instruction scheduling phase would affect the schedule and might decrease the performance of the program.

Figure 2 shows an example on how rescheduling can improve the usage of the shorter instruction templates. In this example, there are no data dependencies limiting the rescheduling while, in a real situation, the data dependencies would usually not allow all the operations to be rescheduled, and the benefit from the optimization would be smaller than in the example.

```
1: for all m in operations do
2:     queue ← m
3: end for
4: while queue not empty do
5:     m = queue.pop()
6:     if counter[m] < limit then
7:         if tryPushNode(m) then
8:             counter[m]++
9:         end if
10:    end if
11: end while
```

Figure 3 The NOPOptimizer outer loop routine.

The algorithm is run for every basic block after that basic block has been scheduled, but before the inter-basic block delay slot filler.

Figure 3 shows how the algorithm first pushes all moves or operations into a queue. After this the main algorithm is iterated as long as the queue contains elements. Only one instance of each operation can be in the queue.

In the main loop, the first element in the queue is popped and processed. If the instruction where the operation belongs is already full, nothing is done for that operation; these instructions are already optimally coded and contain no wasted bits. Rescheduling operations in these moves may sometimes ease up the dependencies of other operations and allow more optimal placement of those other operations, but

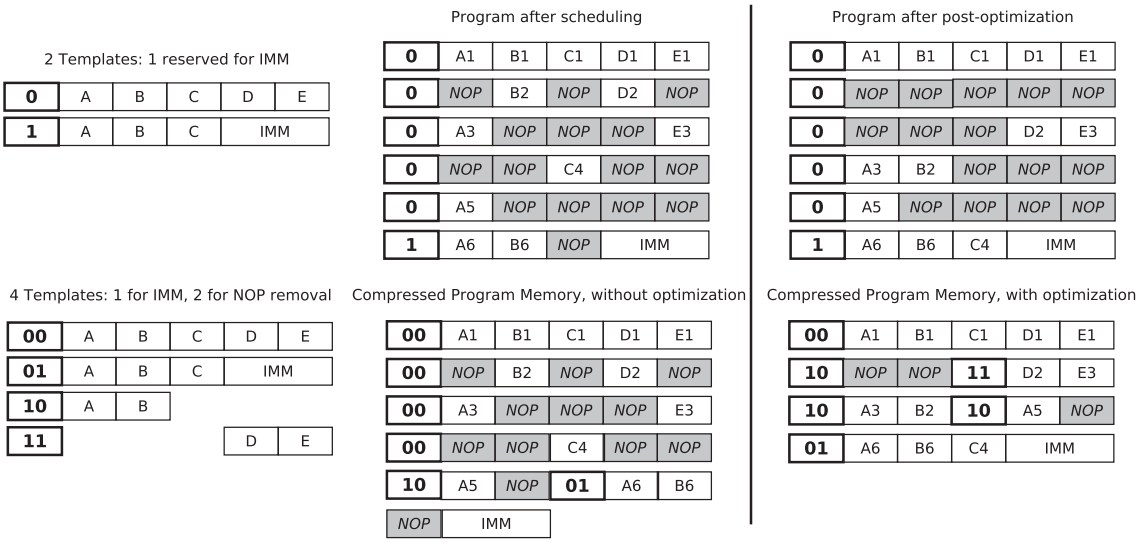


Figure 2 A short program without (*middle*) and with (*right*) the post-scheduler rescheduling. Left side shows the instruction templates used. Upper row shows the operations in full-width instructions and bottom row shows the instructions after the NOP-slot compression is applied. Without the optimization (*middle column*) the usage of both B and D slots simultaneously prevents usage of any short template in instruction 2, usage of both A and E slots simultaneously prevents usage of any short template in instruction 3, and usage of slot C prevents usage of any short template in instruction 4. On the optimized version D2 is moved to instruction 3, A3 and B2 are moved to instruction 4 and C4 is moved to instruction 6. This allows instruction template 10 to be used for instructions 2 and 4 and instruction template 11 to be used for instruction 3.

these situations are rare and rescheduling operations in full instructions prevents the algorithm to finishi naturally. Jump and call operations are not moved since this would affect the length of the basic block and, therefore, the performance of the code. This is illustrated in Fig. 4, lines 1-5.

In Fig. 4, lines 6-18 show how the slack of the schedule is considered; the data dependencies of the operation are checked and the latest and earliest possible time for the operation are calculated based on the data dependencies. If an operation has no data producers limiting how early it can be scheduled, it is not moved earlier, and if it has no data consumers limiting how late it can be scheduled, it is not moved later. This is to guarantee that the basic block cannot get longer and that later inter-basic-block code motion optimizations do not lose optimization opportunities due to the NOP optimization.

The operation is then unscheduled and rescheduling is attempted to both earlier and later cycles, until either it is scheduled to an instruction which will not grow longer due the rescheduling, or the data dependence limits are reached. Moving operations to both directions allow the same algorithm to be used for both top-down and bottom-up scheduled code, and also allows inefficient reschedules to be reverted later without special backtracking logic. When an operation is scheduled into a new instruction, its predecessors are

```
1: originalCycle = m.cycle
2: ins = m.instruction
3: if ins full or m call or jump then
4:    return false
5: end if
6: if datadeps.earliestCycle(m) == datadeps.latestCycle(m) then
7:    return false
8: end if
9: if datadeps.earliestCycle(m) == 0 then
10:   earlyLimit = ∞
11: else
12:   earlyLimit = datadeps.earliestCycle(m)
13: end if
14: if datadeps.latestCycle(m) == ∞ then
15:   latestLimit = -1
16: else
17:   latestLimit = datadeps.latestCycle(m)
18: end if
19: ec = m.cycle
20: lc = m.cycle
21: unschedule(m)
22: repeat
23:   lc = lc + 1
24:   if lc <= latestLimit then
25:      if tryMoveToCycle(m, lc) then
26:         queue ← predecessors(m)
27:         return true
28:      end if
29:   end if
30:   ec = ec - 1
31:   if ec >= earlyLimit then
32:      if tryMoveToCycle(m, ec) then
33:         queue ← successors(m)
34:         return true
35:      end if
36:   end if
37: until ec < earlyLimit and lc > latestLimit
    {Could not reschedule, revert to original}
38: schedule(m, originalCycle)
39: return false
```

Figure 4 The tryPushNode helper routine for the NOPOptimizer.

```
1: ins = instruction(cycle)
2: sizeBefore = size(ins)
3: if fitsIntoCycle(m, cycle) then
4:    schedule(m, cycle)
5:    if size(ins) > sizeBefore then
6:       unschedule m {If worsened the target instruction, undo}
7:       return false
8:    else
9:       return true
10:   end if
11: end if
12: return false
```

Figure 5 The tryToMoveToCycle helper routine for the NOPOptimizer.

requeued if the operation was moved forward, and its successors are requeued if the operation was moved backward. This is shown in Fig. 4, lines 19-39, Figs. 5 and 5.

When new operations are popped from the queue, there is a counter limiting how many times one operation can be rescheduled; this is to prevent the algorithm from going into an infinite loop scheduling some two consecutive operations back and forth. This is shown in Fig. 3, lines 6-10.

4 Evaluation

4.1 Benchmarks

We evaluate the performance of our methods with a subset of the CHStone [15] benchmark. This benchmark is selected since it contains a range of real-world routines, not microbenchmarks, with varying amounts of control code and instruction-level parallelism. Tests *adpcm*, *gsm*, *mips*, *jpeg*, *aes*, *blowfish* and *sha* are used. The software floating-point tests *dfadd*, *dfmul*, *dfdiv* and *dfsin* are omitted since they are microbenchmarks with a very small code footprint so they are not good benchmarks for code size measurement.

4.2 Processor Architectures

In order to measure the efficiency of the optimizations in practice, two *Transport Triggered Architecture (TTA)* type VLIW processors were developed using the *TTA Codesign Environment (TCE)* [16], and the compiler for the TCE toolset was modified to include the proposed optimizations.

TTA-type VLIW gives the compiler extra freedom to transfer some operands to earlier instructions than the execution starts and to read results later than they are produced. The implemented version of the post-scheduler optimizer algorithm takes advantage of this by rescheduling individual moves instead of whole operations.

The processor interconnect was developed with a method resembling the method in [17] to have reasonable interconnect and register file structure to the processor.

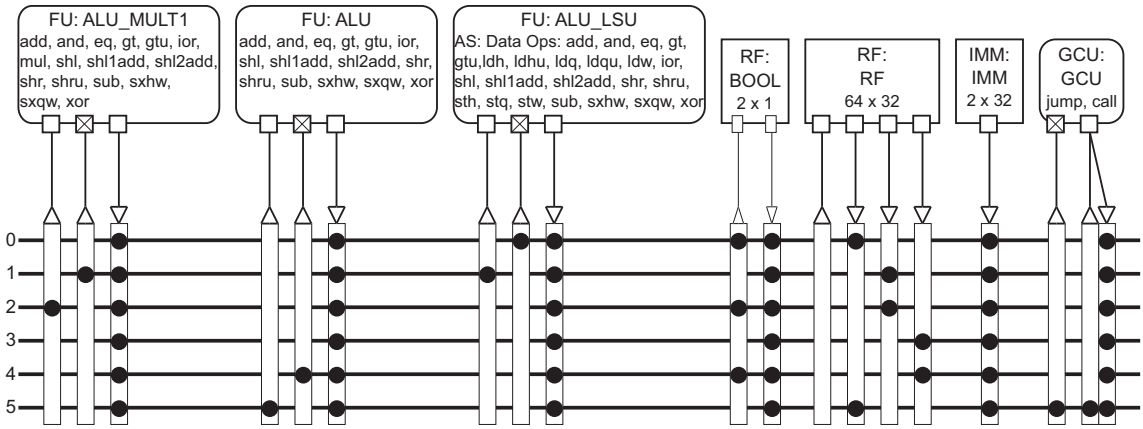


Figure 6 Organization of the threeway Processor used in the evaluations. Function units and register files on top, interconnect buses ans sockets on bottom. Little boxes on the function units and register files are ports, X indicates the trigger port on the function units.

The first processor architecture, *threeway*, is a 3-issue processor with a 96-bit instruction word. The overall processor architecture was designed to give relatively good performance on the CHStone test while keeping the instruction width at 96 bits, to have good balance between performance and instruction size even without variable-length instruction encoding. The processor has six buses, each of which has its own slot in the instruction encoding. The first two buses are connected into a combined *Load-Store-Unit (LSU)* and *Arithmetic-Logical Unit (ALU)*. Third and fourth buses are connected to a combined ALU and multiplier while the fifth and sixth bus are connected to an ALU and the control unit. This processor has three register read ports and one register file write port. Figure 6 shows the organization of the processor.

The processor has four different instruction templates; one with 32-bit length, one with 48-bit length and two full-length ones: one with long immediate value and one without long immediate value. The 32-bit instruction template was selected by first finding all bus combinations that can be encoded in 32 bits and then selecting the one that is used the most often when the adpcm code from the CHStone benchmark was compiled without any compiler optimizations for NOP slot usage. The 48-bit instruction template was selected in a similar manner, finding all the combinations that can be encoded in 48 bits and selecting the one that is mostly used when the adpcm was compiled without any compiler optimizations for NOP slot usage.

Another processor architecture, *fourway*, is a 4-issue processor with a 128-bit instruction word. The processor has 8 buses, each of which have their own slot in the instruction encoding. The organization of the *fourway* processor is such that most code with low level of *instruction-level-parallelism (ILP)* can execute using only the two first buses and the first function unit, as these are connected to both combined ALU and LSU and also separate control unit. The 3rd and 4th bus are connected to a combined ALU and multiplier, and the 5th and 6th bus are connected to another combined ALU and multiplier. The 7th and 8th bus are connected to an ALU. This organization of the function units is reletively close to the default configuration of HP's *VLIW EXample (VEX)* processor architecture [18], with the difference that in VEX the control unit is combined with the last ALU, though the *fourway* processor has considerably fewer number of register file ports(3 read, 2 write) due TTA-specific optimizations such as software bypassing and operand sharing decreasing the need for ports.

Instruction templates in the *fourway* processor architecture are such that in all instruction templates, the first and second bus can always contain moves. In the 40-bit template, all the other slots are implicit NOPs, and in 72-bit template, there is also a 32-bit immediate value in addition to the two first buses. The short templates in this processor are wider than the templates in the *threeway* processor because of the requirement to be able to have the moves in the first two buses in all of the instruction templates.

4.3 Evaluation Results

Tables 1 and 2 show the performance and instruction counts and code sizes of the CHStone benchmark with different optimization methods on the two processors. The baseline "No optimization" in these results means that the shorter instruction templates are used when the compiler happens to generate instructions which can be encoded with them, but no optimizations are performed to encourage their usage.

On higher-ILP workloads such as *adpcm*, *blowfish* and *aes* prioritizing the function units on all basic blocks had a considerable negative effect on the performance, and also the number of instructions. In these cases, the increase in instruction count usually resulted in larger instruction memory than was saved by the better usage of the smaller

Table 1 Instruction template usage on CHStone benchmark with and without the proposed optimizations on 3-issue, 4-template processor.

Test	Strategy	Instr. count	Full -width	48 bits	32 bits	Code size	Cycle count	Size saved	Slowdown
adpcm	No optimization	1600	1145	151	304	126896	64138		
	Post-optimize	1570	1044	182	344	119968	64010	5.5 %	−0.2 %
	Prioritize FUs mode1	1707	1159	164	384	131424	65960	−3.6 %	2.8 %
	Prioritize FUs mode2	1699	1161	161	377	131248	65865	−3.4 %	2.7 %
	prioritize FUs mode3	1602	1130	153	319	126032	64354	0.7 %	0.3 %
	Both mode1	1705	1075	190	440	126400	65821	0.4 %	2.6 %
	Both mode2	1696	1079	190	427	126368	65764	0.4 %	2.5 %
	Both mode3	1572	1029	184	359	119104	64226	6.1 %	0.1 %
jpeg	No optimization	8172	3351	2394	2427	514272	9192740		
	Post-optimize	8166	3249	2461	2456	508624	9197800	1.1 %	0.1 %
	Prioritize FUs mode1	8584	3257	2478	2849	522784	9513412	−1.7 %	3.5 %
	Prioritize FUs mode2	8451	3358	2351	2742	522960	9260249	−1.7 %	0.7 %
	prioritize FUs mode3	8188	3302	2397	2489	511696	9213743	0.5 %	0.2 %
	Both mode1	8576	3167	2524	2885	517504	9518490	−0.6 %	3.5 %
	Both mode2	8442	3263	2407	2772	517488	9265327	-0.6 %	0.7 %
	Both mode3	8184	3198	2465	2521	506000	9218824	1.6 %	0.3 %
aes	No optimization	2131	1377	456	298	163616	24666		
	Post-optimize	2111	1301	500	310	158816	24478	2.9 %	−0.8 %
	Prioritize FUs mode1	2272	1362	482	428	167584	27966	−2.4 %	13.4 %
	Prioritize FUs mode2	2210	1395	456	359	167296	25944	−2.4 %	5.2 %
	prioritize FUs mode3	2131	1373	457	301	163376	24666	0.1 %	0.0 %
	Both mode1	2252	1274	540	438	162240	27778	0.8 %	12.6 %
	Both mode2	2190	1301	517	372	161616	25756	1.2 %	4.4 %
	Both mode3	2111	1297	501	313	158576	24478	3.1 %	-0.8 %
sha	No optimization	644	432	119	93	50160	418703		
	Post-optimize	643	419	128	96	49440	418446	1.4 %	-0.1 %
	Prioritize FUs mode 1	685	452	123	110	52816	458506	−5.3 %	9.5 %
	Prioritize FUs mode 2	656	442	120	94	51200	420761	−2.0 %	0.5 %
	prioritize FUs mode 3	646	429	120	97	50048	418705	0.2 %	0.0 %
	Both mode 1	684	432	133	119	51664	458249	−3.0 %	9.4 %
	Both mode 2	655	423	133	99	50160	420504	0.0 %	0.4 %
	Both mode 3	645	416	129	100	49328	418448	1.7 %	0.1 %
blowfish	No optimization	1185	772	211	202	90704	591302		
	Post-optimize	1185	749	231	205	89552	591297	1.2 %	0.0 %
	Prioritize FUs mode 1	1265	827	203	235	96656	634285	−6.6 %	7.3 %
	Prioritize FUs mode 2	1246	820	197	229	95504	618318	−5.3 %	4.6 %
	prioritize FUs mode 3	1187	769	213	205	90608	591496	0.1 %	0.0 %
	Both mode 1	1265	805	217	243	95472	634285	−5.3 %	7.3 %
	Both mode 2	1246	801	209	236	94480	618313	−4.2 %	4.6 %
	Both mode 3	1187	749	230	208	89600	591491	1.2 %	0.0 %
mips	No optimization	554	217	233	104	35344	34858		
	Post-optimize	535	163	255	117	31632	34619	10.5 %	−0.7 %
	Prioritize FUs mode 1	582	232	235	115	37232	35584	−5.3 %	2.1 %
	Prioritize FUs mode 2	582	232	235	115	37232	35584	−5.3 %	2.1 %
	prioritize FUs mode 3	555	211	238	106	35072	34977	0.8 %	0.3 %
	Both mode 1	563	166	271	126	32976	35345	6.7 %	1.4 %
	Both mode 2	563	166	271	126	32976	34345	6.7 %	1.4 %
	Both mode 3	536	157	260	119	31360	34738	11.3 %	−0.3 %

Table 1 (continued)

Test	Strategy	Instr. count	Full -width	48 bits	32 bits	Code size	Cycle count	Size saved	Slowdown
gsm	No optimization	1823	1027	473	323	131632	12652		
	Post-optimize	1828	1001	492	335	130432	12762	0.9 %	0.9 %
	Prioritize FUs mode 1	1943	1084	474	385	139136	12757	−5.7 %	0.8 %
	Prioritize FUs mode 2	1910	1077	465	368	137488	12990	−4.4 %	2.7 %
	prioritize FUs mode 3	1822	1018	478	326	131104	12712	0.4 %	0.5 %
	Both mode 1	1958	1010	521	427	135632	12898	−3.0 %	1.9 %
	Both mode 2	1925	1012	505	408	134448	13131	−2.1 %	3.8 %
	Both mode 3	1828	991	499	338	129904	12850	1.3 %	1.5 %

Post-optimize runs the post-optimizer after instruction scheduling. *Prioritize FUs mode1* prioritizes function units Based on the implicit NOP slots for all basic blocks. *Priorize FUs mode2* prioritizes function units based on the implicit NOP slots only outside inner loops. *Priorize FUs mode3* Schedules basic blocks outside inner loops with both FU selection heuristics and selects the schedule with the better code size for every basic block. *Both mode1* prioritizes function units based on the NOP slots and runs the post-optimizer. *Both mode2* prioritizes function units based on the NOP slots only outside inner loops and runs the post-optimizer for all code. *Both mode3* Schedules basic blocks outside inner loops with both FU selection heuristics and selects the schedule with the better code size for every basic block, and runs the post-optimizer for all code. Code size is in bits.

instructions, and the total program memory size increased by 1.7 - 6.6 %. The worst slowdown occured with the *blowfish* benchmark where the program slowdown was 28.3 %. On more control-oriented low-ILP workloads such as *gsm* and *mips* prioritizing function units in all basic blocks caused smaller slowdown on performance on both processors, and with *fourway* processor decreased the program memory size by 6.9 - 7.1 %. The *sha* benchmark behaved in similar fashion than *gsm* and *mips* benchmarks, even though it has more ILP, benefiting 2.7 % from the function unit prioritizing. On the *threeway* processor the results of function unit prioritizing in all basic blocks were also negative also in the low-ILP cases. On average always applying the FU priorization resulted in 1.2 % increase in code size with an average slowdown of 7.2 %

Applying the function unit priorization only for code outside inner loops usually helped to decrease the slowdown from the function unit selection in many tests, but also in many cases resulted in smaller code size decrease. However, in *threeway* processor, this did not help to make function unit priorization beneficial, as the code size was still larger than the code size without the optimization, applying the optimization for code outside inner loops only helped to make the code size penalty less severe. On *fourway* processor the best result is achieved in *gsm* test where this optimization results in both smallest code size, 9.1 % smaller than the unoptimized version, with only 0.5 % slowdown. In case case always applying the FU priorization resulted only 7.1 % savings, with a big 7.7 % slowdown. Especially on *sha*, applying priorization only outside inner loops helps; in this case, the code has only a 1.5 % slowdown compared to the 16.1 % slowdown when the optimization is applied for all the basic blocks, and code size is actually smaller, 3.8 % decrease versus 2.7 % decrease. On average this optimization was also harmful, causing on average a 0.3 % code size increase and an average slowdown of 4.1 % between all the test cases.

Scheduling every non-inner-loop basic block with both FU selection heuristics and reverting to the baseline function unit selection heuristic for the basic blocks where FU prioritization produced larger code, code gave much better results. This mode practically eliminated the cases where longer code with more instructions caused code size increase. This optimization was even beneficial on the *threeway* processor, with on average code size saving of 0.4 % while causing in average only a 0.2 % slowdown. On *fourway* processor the results were much better, achieving the best case of 10.7 % and an average of 5.6 % saved code size at the cost of an average slowdown of 0.5 %.

The post-optimizer pass mode had a more stable effect on both performance and code size on both processors. The code size decrease was in the range between 0.0 % and 10.5 %. The performance in all cases was very close to the original performance, in average being 0.3 % better than the non-optimized version. The average code size decrease was 3.4 % for the *threeway* processor and 1.5 % for the *fourway* processor, the average of both being 2.4 %. The reason for the weak improvement with the *fourway* processor is that operations in other than the first function unit always required a full-length instruction to be used, while with *threeway* there was also a shorter instruction template that included the final three buses.

Table 2 Instruction template usage on CHStone benchmark with and without the proposed optimizations on a 4-issue, 4-template processor.

Test	Strategy	Instr. count	Full -width	72 bits	40 bits	Code size	Cycle count	Size saved	Slowdown
adpcm	No optimization	1290	935	261	94	142232	58170		
	Post-optimize	1285	854	288	143	135768	58018	4.5 %	−0.3 %
	Priorize FUs mode1	1448	858	352	238	144688	61544	−1.7 %	5.8 %
	Priorize FUs mode2	1409	848	348	213	142120	61232	0.1 %	5.3 %
	Priorize FUs mode3	1299	880	281	138	138392	58662	2.7 %	0.8 %
	Both, mode1	1444	815	368	261	141256	61442	0.7 %	5.6 %
	Both, mode2	1405	805	364	236	138688	61130	2.5 %	5.1 %
	Both, mode3	1295	797	309	189	131824	58560	7.3 %	0.7 %
jpeg	No optimization	7634	4196	1146	2292	711280	8362618		
	Post-optimize	7632	4103	1181	2348	704136	8362356	1.0 %	0.0 %
	Priorize FUs mode1	8625	3456	1836	3333	707880	8759608	0.5 %	4.7 %
	Priorize FUs mode2	8364	3520	1781	3063	701312	8379090	1.4 %	0.2 %
	Priorize FUs mode3	8052	3506	1523	3023	679344	8415760	4.5 %	0.6 %
	Both mode1	8619	3373	1866	3380	701296	8759368	1.4 %	4.7 %
	Both mode2	8363	3451	1808	3104	696064	8378857	2.1 %	0.2 %
	Both mode3	8050	3435	1555	3060	674040	8415239	5.2 %	0.6 %
aes	No optimization	1684	1256	252	176	185952	22430		
	Post-optimize	1683	1225	264	194	183568	22298	1.3 %	−0.6 %
	Priorize FUs mode1	1955	1072	445	438	186776	22386	−0.4 %	−0.2 %
	Priorize FUs mode2	1869	1138	419	312	188312	23354	−1.3 %	4.2 %
	Priorize FUs mode3	1721	1194	308	219	183768	22610	1.2 %	0.8 %
	Both mode1	1952	1028	458	466	183200	22386	1.5 %	−0.2 %
	Both mode2	1867	1090	436	341	184552	23288	0.8 %	3.8 %
	Both mode3	1718	1162	322	234	181280	22468	2.5 %	0.2 %
sha	No optimization	545	401	72	72	59392	316146		
	Post-optimize	542	393	77	72	58728	315886	1.1 %	−0.1 %
	Priorize FUs mode1	596	347	106	143	57768	367023	2.7 %	16.1 %
	Priorize FUs mode2	578	349	103	126	57128	321028	3.8 %	1.5 %
	Priorize FUs mode3	563	347	93	123	56032	317173	5.7 %	0.3 %
	Both mode1	596	337	114	145	57144	367023	3.8 %	16.1 %
	Both mode2	578	346	105	127	56928	321028	4.1 %	1.5 %
	Both mode3	562	343	95	124	55704	316916	6.2 %	0.2 %
blowfish	No optimization	1050	737	168	145	112232	539242		
	Post-optimize	1052	714	180	158	110672	534952	1.4 %	−0.8 %
	Priorize FUs mode1	1235	607	324	304	113184	691743	−0.8 %	28.3 %
	Priorize FUs mode2	1202	604	317	281	111376	690265	0.8 %	28.0 %
	Priorize FUs mode3	1072	658	206	208	107376	547721	4.3 %	1.5 %
	Both mode1	1232	572	340	320	110496	681733	1.5 %	26.4 %
	Both mode2	1204	576	333	295	109504	686233	2.4 %	27.3 %
	Both mode3	1074	642	215	217	106336	539039	5.3 %	0.0 %
mips	No optimization	497	257	102	138	45760	34661		
	Post-optimize	497	257	102	138	45760	34661	0.0 %	
	Priorize FUs mode1	558	173	158	227	42600	34260	6.9 %	−1.2 %
	Priorize FUs mode2	558	173	158	227	42600	34260	6.9 %	−1.2 %
	Priorize FUs mode3	542	163	155	224	40984	34272	10.4 %	−1.1 %
	Both mode1	558	172	159	227	42544	34260	7.0 %	−1.2 %
	Both mode2	558	172	159	227	42544	34260	7.0 %	−1.2 %
	Both mode3	542	163	155	224	40984	34272	10.4 %	−1.1 %

Table 2 (continued)

Test	Strategy	Instr. count	Full -width	72 bits	40 bits	code size	cycle count	size saved	slowdown
gsm	No optimization	1680	1117	281	282	174488	11159		
	Post-optimize	1680	1084	310	286	172512	11059	1.1 %	−0.9 %
	Priorize FUs mode1	1747	919	353	475	162048	12016	7.1 %	7.7 %
	Priorize FUs mode2	1671	924	328	419	158648	11212	9.1 %	0.5 %
	Priorize FUs mode3	1615	920	318	377	155736	11229	10.7 %	0.6 %
	Both mode1	1745	868	373	504	158120	12018	9.4 %	7.7 %
	Both mode2	1669	884	347	438	155656	11139	10.8 %	0.2 %
	Both mode3	1615	884	340	391	153272	11159	12.2 %	0.0 %

No optimization prioritizes function units based on supported operations and selects the unit with the fewest operations. *Post-optimize* runs the post-optimizer after instruction scheduling. *Prioritize FUs mode1* prioritizes function units based on the implicit NOP slots for all basic blocks. *Priorize FUs mode2* prioritizes function units based on the implicit NOP slots only outside inner loops. *Priorize FUs mode3* Schedules basic blocks outside inner loops with both FU selection heuristics and selects the schedule with the better code size for every basic block. *Both mode1* prioritizes function units based on the NOP slots and runs the post-optimizer. *Both mode2* prioritizes function units based on the NOP slots only outside inner loops and runs the post-optimizer for all code. *Both mode3* Schedules basic blocks outside inner loops with both FU selection heuristics and selects the schedule with the better code size for every basic block, and runs the post-optimizer for all code. Code size is in bits.

The effect of applying both optimizations together usually had a similar result as the sum of the benefits of the optimizations done separately, and the best result was usually obtained by applying together the post-optimizer pass and FU priorization in a mode where it reverts to the old heuristics in basic blocks where the FU priorization would cause larger code size due larger number of instructions. The best case improvement was 12.2 % in the *gsm* test in processor *fourway* while performance stayed exactly the same. The average code size decrease in this mode was 5.4 % while performance decreased by 0.1 %

5 Future Work

In the second mode of the function unit selection heuristic optimization the basic blocks where the optimization is used and when not are statically selected by the compiler by just analysing whether the code is in inner loop or not. Profiling could be used to identify what are actually the most critical inner loops and to leave only those unoptimized.

The post-scheduler optimizer should also be improved to aggressively perform horizontal rescheduling of operations into those execution slots which have fewer implicit NOP slots associated with them, even when the main scheduling is done with function unit selection which favors performance instead of code size. This could increase the code size savings from the post-scheduler optimizer without performance degradion.

6 Conclusions

Two compiler optimizations to better utilize short instruction words in a variable-length instruction coding scheme were presented and analyzed. The introduced post-optimizer pass resulted on average 2.4 % and the best case of 10.5 % code size reduction without performance loss. As the post-scheduler optimizer had a good impact on both code size and performance, it should be always used.

Always prioritizing function units based on the implicit NOP slots gave best case code size savings of 7.1 % while on average the code size increased by 1.2 % while the performance decreased by an average of 7.2 %. Also the processor architecture and the selection of instruction templates had a significant effect on whether the function unit priorization decreased or increased the code size; with the *threeway* processor the function unit selection increased code size, but with *fourway* it decreased the code size. Since it may reduce performance and sometimes even increase code size, the function unit prioritization should be used cautiously, only after testing that it provides benefit for the case at hand and only in cases where the performance penalty is not too severe.

Prioritizing function units based on the implicit NOP slots only outside inner loops gave best case code size savings of 9.1 % but the average code size increased by 0.3 % while performance decreased by average of 4.1 %. In most cases both the performance drawbacks and the code size savings were smaller than when the optimization

was always performed, but compared to always performing the optimization, the performance saved by not doing the optimization was usually better than code size wasted. But compared to the unoptimized version, performance still dropped in most test cases, so even this mode should not be enabled without testing that it is really beneficial.

Because results from the function unit priorization were so often negative, third mode had to be implemented to it. In this mode, for every non-innerloop basic block it is first tested which heuristic gives better code size, and then this heuristic is selected for the final scheduling. This mode gave best case code size saving of 10.7 % and average of 3.0 % code with average slowdown of 0.4 %.

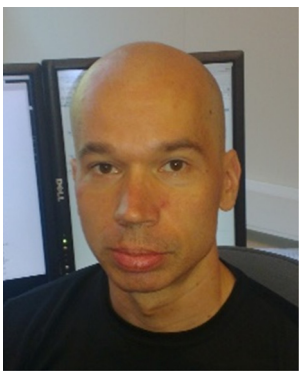
By combining the two optimization methods, a best case of 12.2 and average of 5.4 % code size savings could be achieved, with only a 0.1 % performance loss.

The fact that results from the function unit prioritization were sometimes negative also weight the importance of good low level instruction scheduler. The quality of the instruction scheduler can sometimes have greater impact on code size than special optimizations to decrease it.

Acknowledgments This work was funded by Academy of Finland (funding decision 253087), Finnish Funding Agency for Technology and Innovation (project "Parallel Acceleration", funding decision 40115/13), and ARTEMIS Joint Undertaking under grant agreement no 621439 (ALMARVI).

References

1. Corporaal, H., & Arnold, M. (1998). Using Transport Triggered Architectures for embedded processor design. *Integrated Computer-Aided Engineering*, *5*(1), 19–38.
2. Conte, T.M., Banerjia, S., Larin, S.Y., Menezes, K.N., & Sathaye, S.W. (1996). Instruction fetch mechanisms for VLIW architectures with compressed encodings. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 201–211).
3. Aditya, S., Mahlke, S.A., & Rau, B.R. (2000). Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats. *ACM Transactions on Design Automation of Electronic Systems*, *5*(4), 752–773.
4. Helkala, J., Viitanen, T., Kultala, H., Jääskeläinen, P., Takala, J., Zetterman, T., & Berg, H. (2014). Variable length instruction compression on transport triggered architectures. In *Proceedings of the International Conference on Embedded Computing Systems: Architectures Modeling and Simulation* (pp. 149–155). Samos, Greece.
5. Kultala, H., Viitanen, T., Jääskelainen, P., Helkala, J., & Takala, J. (2014). Compiler optimizations for code density of variable length instructions. In *Proceedings of the IEEE Workshop on Signal Processing Systems* (pp. 1–6).
6. Lee, C., Lee, J.K., & Hwang, T. (2000). Compiler optimization on instruction scheduling for low power. In *Proceedings of the 13th International Symposium on System Synthesis* (pp. 55–60).
7. Hahn, T.T., Stotzer, E., Sule, D., & Asal, M. (2008). Compilation strategies for reducing code size on a VLIW processor with variable length instructions. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers* (pp. 147–160). Berlin Heidelberg: Springer-Verlag.
8. Stotzer, E.J., & Leiss, E.L. (2012). Co-design of compiler and hardware techniques to reduce program code size on a vliw processor. *CLEI Electronic Journal*, *15*(2), 2–2.
9. Jee, S., & Palaniappan, K. (2002). Performance evaluation for a compressed-VLIW processor. In *Proceedings of the ACM Symposium on Applied Computing* (pp. 913–917).
10. Ros, M., & Sutton, P. (2005). A post-compilation register reassignment technique for improving hamming distance code compression. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (pp. 97–104).
11. Larin, S.Y., & Conte, M.T. (1999). Compiler-driven cached code compression schemes for embedded ilp processors. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 82–92): IEEE.
12. Haga, S., Webber, A., Zhang, Y., Nguyen, N., & Barua, R. (2005). Reducing code size in VLIW instruction scheduling. *Journal of Embedded Computing*, *1*(3), 415–433.
13. Haga, S., & Barua, R. (2001). EPIC instruction scheduling based on optimal approaches. In *Proceedings of the First Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology* (pp. 22–31).
14. Muchnick, S.S. (1997). *Advanced Compiler Design and Implementation*: Morgan Kaufmann.
15. Hara, Y., Tomiyama, H., Honda, S., & Takada, H. (2009). Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, *17*, 242–254.
16. Jääskeläinen, P., Guzma, V., Cilio, A., & Takala, J. (2007). Codesign toolset for application-specific instruction-set processors. In *Proceedings of SPIE Multimedia on Mobile Devices* (pp. 65070X–1 – 65070X–11).
17. Viitanen, T., Kultala, H., Jääskeläinen, P., & Takala, J. (2014). Heuristics for greedy transport triggered architecture interconnect exploration. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (pp. 2:1–2:7).
18. Fisher, J.A., Faraboschi, P., & Young, C. (2005). *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*: Elsevier.



Heikki Kultala (M.Sc) received his M.Sc degree in Computer Science from Tampere University of technology (TUT) in 2010, and is now a graduate student at the Department of Pervasive Computing of TUT. He has worked with exposed datapath processor architectures and programming since 2005. His research interests include compiler backend techniques, low-level code optimizations and processor architecture customization.

Timo Viitanen (M. Sc.) received his M.Sc. degree in embedded systems from Tampere University of Technology (TUT) in 2013, and is now a graduate student at the Department of Pervasive Computing of TUT, where his main work has been on automated design space exploration and hardware implementation of TTA processors. His research interests include processor architecture, floating-point arithmetic and computer graphics.

Pekka Jääskeläinen (Dr. Tech.) has worked with exposed datapath processor architecture customization and programming since 2002. He has led the development work of the TTA-based Co-design environment (TCE), a toolset for rapid customization of VLIW-style parallel processors based on the Transport-Triggered Architecture. He received his master's degree in 2005, and doctor's degree in 2012 from Tampere University of Technology. Both of the thesis involved topics in parallel processor design and TTAs. His current research interests include programmable parallel platforms and compiler techniques for enhancing performance portability of parallel programs.

Janne Helkala (M.Sc.) graduated with Master of Science degree from Tampere University of Technology (TUT) in 2014, where he researched variable length instruction compression and no-operation optimization for Transport-Triggered Architecture (TTA) processors. He currently works at Nokia Networks as a system on chip design engineer.

Jarmo Takala received his M.Sc. (hons) degree in Electrical Engineering and Dr.Tech. degree in Information Technology from Tampere University of Technology, Tampere, Finland (TUT) in 1987 and 1999, respectively. From 1992 to 1995, he was a Research Scientist at VTT-Automation, Tampere, Finland.

Between 1995 and 1996, he was a Senior Research Engineer at Nokia Research Center, Tampere, Finland. From 1996 to 1999, he was a Researcher at TUT. Since 2000, he has been Professor in Computer Engineering at TUT and currently Dean of the Faculty of Computing and Electrical Engineering of TUT. Dr. Takala is Co-Editor-in-Chief for Springer Journal on Signal Processing Systems. During 2007-2011 he was Associate Editor and Area Editor for IEEE Transactions on Signal Processing and in 2012-2013 he was the Chair of IEEE Signal Processing Society's Design and Implementation of Signal Processing Systems Technical Committee. His research interests include circuit techniques, parallel architectures, and design methodologies for digital signal processing systems.