Compiler Optimizations for Code Density of Variable Length Instructions

Heikki Kultala, Timo Viitanen, Pekka Jääskeläinen, Janne Helkala, Jarmo Takala
Department of Pervasive Computing, Tampere University of Technology, Finland
Email: {heikki.kultala, timo.2.viitanen, pekka.jaaskelainen, janne.helkala, jarmo.takala}@tut.fi

***Abstract*—Variable length encoding can considerably decrease code size in VLIW processors by decreasing the amount of bits wasted on encoding *No Operation(NOP)s*. 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: prioritizing function units that have fewer implicit NOPs associated to them, and 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. The post-pass optimizer saved an average of 2.5 % and at best of 9.1 % instruction memory, without performance loss. Prioritizing function units gave best case instruction memory savings of 12.7 % but the average savings were only 1.0 % and there was in average 5.7 % slowdown for the program.**

I. Introduction

One of the main pitfalls of *Very Long Instruction Word (VLIW)* and *Transport Triggered Architectures (TTAs)* [1] is large code size which is caused by the long instruction word and *No Operation(NOP)s* that have to be inserted into the program code. When the processor has several computational resources to achieve high performance on critical tight unrolled and software-pipelined loops, it also has wide instruction word with several execution slots, and often there is lots of rarely-executed more control-oriented helper code outside these critical loops consuming lots of instruction memory. This code cannot exploit the parallel execution units, this the code contains large number of instructions where most of the execution slots are NOPs.

To save instruction memory and instruction fetch power, this code should be encoded with instruction encoding where most of the execution slots are encoded in such way that most of execution slots implicitly NOPs, and the instruction word is much narrower. The processor may have different instruction templates to encode such instructions where some execution slots are NOPs. However, supporting all the combinations of operations in wide instructions might make the instruction decoding logic too complicated especially in cases where the execution slots in the instruction encoding may have different bit widths. In a reasonable implementation there are only few different instruction templates where different execution slots are implicitly NOPs. [2] [3]

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. These templates can be used in three instructions to remove a majority of the NOP operations in the program code. Considerable instruction memory savings can be achieved by simply scheduling the instructions for maximum performance without optimizing the code for the implicit NOP slots and just using the shorter instructions when they can be used. 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: Prioritizing function units that have implicit NOP slots associated to them in fewer instruction templates, and executing a post-scheduler NOPOptimizer which utilizes slack of the schedule by spatially moving operations which have slack 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.

II. Related work

In [4] a post-scheduler optimization algorithm is introduced 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 same instructions. This method however does not consider the NOP usage and does not try to optimize the code size.

In [5], a variable-length encoding for VLIW is proposed. 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

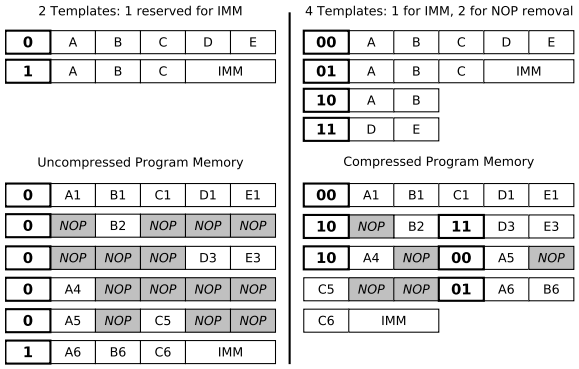

Fig. 1. A short program before (left) and after (right) assigning two new instruction formats, which define two execution slots to be used out of the five in the processor. Most of the NOP operations are removed by using the shorter instruction formats in the 2nd, 3rd and 4th instruction.

instructions, and uses the ordinary version of the operation if possible, to maximize performance. In case it cannot schedule any useful operations to the 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 the instructions, resulting in lower performance but eliminating all NOPs due to delay slots and long-latency operations.

In [6], 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 length problem.

The approach in [7] 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.

In [8], a method to minimize code size with global scheduling is introduced. Their approach, however, does not consider optimization the code size for variable-length instructions.

In [9] compiler optimizations for EPIC architecture are discussed. EPIC architecture is a variation of VLIW where one group of instructions can execute in parallel and one instruction word may contain operations from different instruction groups, and one instruction group can span over multiple instruction words. The instruction groups are separated by stop bits which are specified by the instruction template used for the instruction words. They 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 alforithm 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 and their method can be only used for EPIC-type instruction templates.

III. Proposed Optimizations

A. Baseline

In this work, the cycle-based list scheduler [10] is used as baseline and the proposed optimizations are compared against this method. The baseline 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 same cycle, the smallest function unit that can execute the operation is selected. If there are multiple units with an equal amount of operations, then the connectivity of the function unit is considered and the unit with the least connectivity is selected. The reason for this logic is that if some operation, for example addition, can be executed on both function unit A and function units B, but another operation, for example multiplication, can only be executed on unit A, the add operations will mostly be scheduled onto unit B, so that unit A will be free to execute the multiplications. The optimizations could also be used with different instruction schedulers.

B. 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 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.

C. 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, the data dependencies do not limit the rescheduling; In 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 this example.

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. Then the main algorithm is iterated as long as the queue contains elements. An operation can be in the queue only once. 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, there are no wasted bits in those instructions. Rescheduling operations in these moves might sometimes ease up the dependencies of other operations and

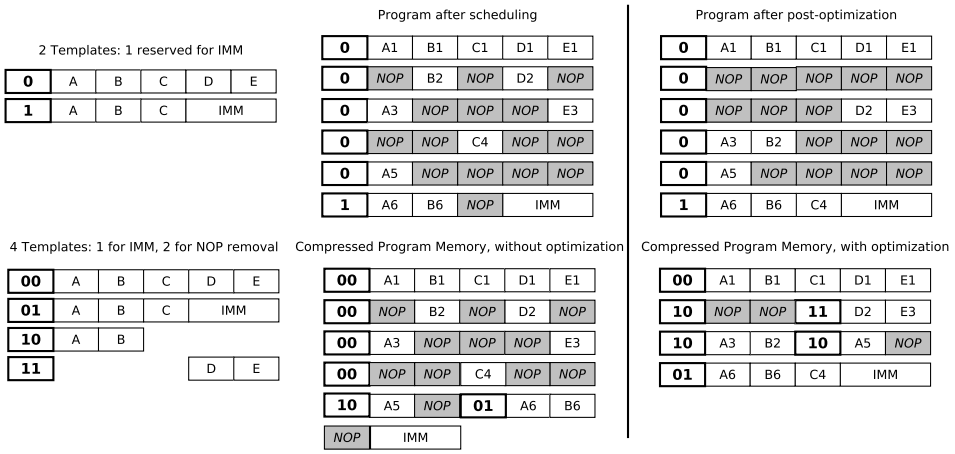

Fig. 2. A short program without (middle) and with (right) the post-scheduler rescheduling. Left side shows the instruction templates used. Upper row shows the operations in full-width instructions and bottom row shows the instructions after the NOP-slot compression is applied. Without the optimization (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.

allow more optimal placement of those other operations, but these situations are very rare and rescheduling also operations in already full instructions would mean that the algorithm would never finish naturally. Jump and call operations are not moved since moving them would affect the length of the basic block and, therefore, the performance of the code. This is shown in Fig. 4, lines 1-5

Figure 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 tried to be scheduled 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 later reverting of inefficient reschedules without special backtracking logic. When an operation is scheduled into a new instruction, its predecessors are requeued if the operation was moved forward, and successors are requeued if the operation was moved backward. This is shown in Fig. 4, lines 19-39 and Figure 5

When taking new operations 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

```
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
```

Fig. 3. The NOPOptimizer outer loop routine

IV. Evaluation

A. Benchmarks

We evaluate the performance of our methods with a subset of the CHStone [11] 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.

B. 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)* [12], 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

```
originalCycle = m.cycle
ins = m.instruction
if ins full or m call or jump then
    return false
end if
if datadeps.earliestCycle(m) == datadeps.latestCycle(m) then
    return false
end if
if datadeps.earliestCycle(m) == 0 then
    earlyLimit = ∞
else
    earlyLimit = datadeps.earliestCycle(m)
end if
if datadeps.latestCycle(m) == ∞ then
    latestLimit = -1
else
    latestLimit = datadeps.latestCycle(m)
end if
ec = m.cycle
lc = m.cycle
unschedule(m)
repeat
    lc = lc + 1
    if lc <= latestLimit then
        if tryMoveToCycle(m, lc) then
            queue ← predecessors(m)
            return true
        end if
    end if
    ec = ec - 1
    if ec >= earlyLimit then
        if tryMoveToCycle(m, ec) then
            queue ← successors(m)
            return true
        end if
    end if
until ec < earlyLimit and lc > latestLimit
  {Could not reschedule, revert to original}
schedule(m, originalCycle)
return false
```

Fig. 4. The tryPushNode helper routine for the NOPOptimizer

```
ins = instruction(cycle)
sizeBefore = size(ins)
if fitsIntoCycle(m, cycle) then
    schedule(m, cycle)
    if size(ins) > sizeBefore then
        unschedule m
        return false
    else
        return true
    end if
end if
return false
```

Fig. 5. The tryToMoveToCycle helper routine for the NOPOptimizer

takes advantage of this by rescheduling individual moves instead of whole operations.

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 6 buses, each of which have their own slot in the instruction encoding. The first two buses are connected into a combined *Load-Store-Unit (LSU)* and *Arithmetic-Logical Unit (ALU)*. Buses three and four are connected to a combined ALU and multiplier, and buses five and six are connected to ALU and control unit.

The processor has 4 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 mostly used when the adpcm of the CHStone benchmark was compiled without any compiler optimizations for the NOP slot usage. The 48-bit instruction template was selected in similar manner, finding all combinations that can be encoded in 48 bits and selecting the one that is mostly used when the adpcm of the CHStone benchmark was compiler without any compiler optimizations for the NOP slot usage.

Another processor architecture, *fourway*, is a 4-issue processor with a 128-bit instruction word. The processor architecture has 8 buses, each of which have their own slot in the instruction encoding. The organization of the *fourway* processor architecture 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 combined ALU and multiplier, and 5th and 6th bus are connected to another combined ALU and multiplier. 7th and 8th bus are connected an ALU. This organization of the function units is reletively close to the default configuration of HP's *VLIW EXample (VEX)* processor architecture [13], with the difference that in VEX the control unit is combined with the last ALU.

Instruction templates in *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 32-bit immediate value in addition to the two first buses. The short templates in this processor are bigger 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.

C. Evaluation Results

Tables I and II show the performance and instruction counts and code sizes of the CHStone benchmark with different optimization methods on the two processors. The baseline "No opt" in these results means that the shorter instructions are used when the compiler happens to generate instructions which can be encoded with smaller instructions, but the compiler does not perform any optimizations which encourage their usage.

On more high-ILP workloads such as *adpcm*, *blowfish* and *aes* prioritizing the function units had a considerable negative effect on the performance, and also the number of instructions. In these cases, the increase in instruction count usually caused bigger consumption of instruction memory than what was saved by the better usage if the smaller instructions, and the total program memory size increased by 2.8 - 4.7 %. The worst slowdown occured with the *blowfish* benchmark where the program slowdown was 15.8 %. On more control-oriented low-ILP workloads such as *gsm* and *mips* prioritizing function units caused smaller slowdown on performance on both processors, and with *fourway* processor decreased the program memory size by 10.0 - 12.7 %. The *sha* benchmark behaved in similar fashion than *gsm* and *mips* benchmarks, even though it has more ILP, benefiting 6.2 % from the function unit prioritizing. On the *threeway* processor the results of function

TABLE I

INSTRUCTION TEMPLATE USAGE ON CHSTONE BENCHMARK WITH AND WITHOUT THE PROPOSED OPTIMIZATIONS ON 3-ISSUE, 4-TEMPLATE PROCESSOR. *No optimization* PRIORITIZES FUNCTION UNITS BASED ON SUPPORTED OPERATIONS AND SELECTS THE UNIT WITH THE FEWEST OPERATIONS. *Prioritize FUs* PRIORITIZES FUNCTION UNITS BASES ON THE IMPLICIT NOP SLOTS. *Post-optimize* RUNS THE POST-OPTIMIZER AFTER INSTRUCTION SCHEDULING. *Both* PRIORITIZES FUNCTION UNITS BASED ON THE NOP SLOTS AND RUNS THE POST-OPTIMIZER. CODE SIZE IS IN BITS.

Test	Strategy	Instr. count	Full-width	48 bits	32 bits	code size	cycle count	code size saved	slowdown
adpcm	No opt	1535	1215	78	242	128128	69780		
	Prioritize FUs	1642	1215	92	335	131776	70274	-2.8 %	0.7 %
	Post-Optimize	1533	1124	106	303	122688	69680	4.2 %	-0.1 %
	Both	1639	1085	130	424	123968	70124	3.2 %	0.5 %
jpeg	No opt	7914	2837	2330	2747	472096	8443457		
	Prioritize FUs	8422	2665	2530	3237	479904	9540437	-1.7 %	13.0 %
	Post-Optimize	7898	2798	2331	2769	469104	8448877	0.6 %	0.1 %
	Both	8405	2610	2536	3259	476576	9542282	-0.9 %	13.0 %
aes	No opt	1959	1235	388	336	147936	25555		
	Prioritize FUs	2128	1248	389	491	154192	28833	-4.2 %	12.8 %
	Post-Opt	1938	1144	440	354	142272	25366	3.8 %	-0.7 %
	Both	2115	1183	419	513	150096	28773	-1.4 %	12.6 %
blowfish	No opt	1162	770	203	189	89712	587828		
	Prioritize FUs	1266	788	189	289	93968	680532	-4.7 %	15.8 %
	Post-Opt	1162	720	231	211	86960	578593	3.1 %	-1.6 %
	Both	1266	744	228	294	91776	671302	-2.3 %	14.2 %
gsm	No opt	1721	1050	420	251	128992	12437		
	Prioritize FUs	1793	1066	393	334	131888	12629	-2.2 %	1.5 %
	Post-Opt	1722	1021	439	262	127472	12410	1.2 %	-0.2 %
	Both	1794	1033	414	347	130144	12689	-0.9 %	2.0 %
mips	No opt	562	213	226	123	35232	34637		
	Prioritize FUs	567	217	221	129	35568	35300	-1.0 %	1.9 %
	Post-Opt	543	169	240	134	32032	34398	9.1 %	-0.7 %
	Both	548	171	236	141	32256	35061	8.4 %	1.2 %
sha	No opt	643	430	126	87	50112	406368		
	Prioritize FUs	669	427	129	113	50800	417165	-1.4 %	2.7 %
	Post-Opt	644	408	141	95	48976	406366	2.3 %	0.0 %
	Both	667	409	138	120	49728	421017	0.8 %	3.6 %

TABLE II

INSTRUCTION TEMPLATE USAGE ON CHSTONE BENCHMARK WITH AND WITHOUT THE PROPOSED OPTIMIZATIONS ON A 4-ISSUE, 4-TEMPLATE PROCESSOR. *No optimization* PRIORITIZES FUNCTION UNITS BASED ON SUPPORTED OPERATIONS AND SELECTS THE UNIT WITH THE FEWEST OPERATIONS. *Prioritize FUs* PRIORITIZES FUNCTION UNITS BASES ON THE IMPLICIT NOP SLOTS. *Post-optimize* RUNS THE POST-OPTIMIZER AFTER INSTRUCTION SCHEDULING. *Both* PRIORITIZES FUNCTION UNITS BASED ON THE NOP SLOTS AND RUNS THE POST-OPTIMIZER. CODE SIZE IS IN BITS.

Test	Strategy	Instr. count	Full-width	72 bits	40 bits	code size	cycle count	code size saved	slowdown
adpcm	No opt	1254	993	174	87	143112	64035		
	Prioritize FUs	1407	878	295	234	142984	65645	0.1 %	2.5 %
	Post-Opt	1249	954	197	98	140216	63884	2.0 %	-0.2 %
	Both	1403	835	308	260	139456	65544	2.6 %	2.4 %
jpeg	No opt	7573	4032	1129	2412	693864	8379890		
	Prioritize FUs	8460	3398	1357	3705	680848	8035700	1.9 %	-4.1 %
	Post-Opt	7562	3938	1159	2465	686112	8374541	1.1 %	-0.6 %
	Both	8458	3314	1390	3754	674432	8034580	2.8 %	-4.1 %
aes	No opt	1682	1275	181	226	185272	23912		
	Prioritize FUs	1890	1144	340	406	187152	24602	-1.0 %	2.8 %
	Post-Opt	1682	1226	215	241	182048	23891	1.7 %	-0.1 %
	Both	1886	1112	351	423	184528	24492	0.4 %	2.4 %
blowfish	No opt	1041	726	143	172	110104	511568		
	Prioritize FUs	1166	656	226	284	111600	636276	-1.4 %	24.4 %
	Post-Opt	1039	696	167	176	108152	511308	1.8 %	-0.5 %
	Both	1166	612	241	313	108208	636276	1.7 %	24.4 %
gsm	No opt	1634	1132	238	264	172592	11455		
	Prioritize FUs	1663	908	281	474	155416	11811	10.0 %	3.1 %
	Post-Opt	1626	1101	251	274	169960	11647	1.5 %	1.7 %
	Both	1667	869	290	508	152432	11818	11.7 %	3.2 %
mips	No opt	498	274	99	125	47200	34742		
	Prioritize FUs	556	164	142	250	41216	33830	12.7 %	-2.6 %
	Post-Opt	498	268	99	131	46672	34742	1.1 %	0.0 %
	Both	556	164	142	250	41216	33830	12.7 %	-2.6 %
sha	No opt	585	440	73	72	64456	380379		
	Prioritize FUs	628	355	128	145	60456	400955	6.2 %	5.4 %
	Post-Opt	584	431	78	75	63784	380122	1.0 %	-0.1 %
	Both	627	344	132	151	59576	399931	7.6 %	5.1 %

unit prioritizing were also negative, but the increase in code size was smaller than with *fourway*, in the range of -1.0 to -2.2 %

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 0.6 % and 9.1 %. The performance in all cases was very close to the original performance, in average being 0.4 % better than the non-optimized version. The average code size decrease was 3.5 % for the *threeway* processor, 1.5 % for the *fourway* processor, average of both being 2.5 %. The reason for the weaker 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.

The effect of applying both optimizations together usually had a similar result as the sum of the benefits of the optimizations done separately, but sometimes gave better performance, for example, in the adpcm and aes tests on the *threeway* processor and blowfish test on *fourway* processor.

The best case improvement was same 12.7 % as with the function unit selection in the *mips* test in processor *fourway*. In this test the performance and code size was exactly the same as when only prioritizing function units, so the post-optimizer could not do anything when the function units were prioritized for the NOP slots, even though without the function unit priorization the post-optimizer could decrease the program size by 1.5 % in the same test with the same processor.

V. Future work

As most of the sparse code that benefits most from the variable instruction encoding is outside the critical inner loops of the program, the overall slowdown of the program might be relatively small if the compiler would sacrifice some performance in order to reduce the code size in these parts of the code.

Loop analysis or profiling should be used to identify these non-performance-critical parts of the program and the NOP usage optimization methods that can sacrifice performance should only be applied to these parts of the code, producing smaller code while keeping performance of the critical inner loops at same level.

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

VI. 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 in average 2.5 % and a best case of 9.1 % 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.

Prioritizing function units based on the implicit NOP slots gave best case code size savings of 12.7 % and average savings of 1.0 % while the performance decreased by an average of 5.7 %. Also the architecture had significant effect 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. Due the performance reduction and sometimes even code size increase, the function unit prioritization should be used cautiously, only after testing that it provide benefit for the case and only in cases where the performance reduction is not too severe.

Acknowledgment

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 641439 (ALMARVI).

References

[1] H. Corporaal and M. Arnold, "Using Transport Triggered Architectures for embedded processor design," *Integrated Computer-Aided Eng.*, vol. 5, no. 1, pp. 19–38, 1998.

[2] S. Aditya, S. A. Mahlke, and B. R. Rau, "Code size minimization and retargetable assembly for custom epic and vliw instruction formats," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 4, pp. 752–773, Oct. 2000.

[3] J. Helkala, T. Viitanen, H. Kultala, P. Jääskeläinen, J. Takala, T. Zetterman, and H. Berg, "Variable Length Instruction Compression on Transport Triggered Architectures," in *Proc. Int. Conf. on Embedded Comput. Syst.: Architectures Modeling and Simulation*, Samos, Greece, 2014, pp. 149–155.

[4] C. Lee, J. K. Lee, and T. Hwang, "Compiler optimization on instruction scheduling for low power," in *System Synthesis, 2000. Proceedings. The 13th International Symposium on*, 2000, pp. 55–60.

[5] T. T. Hahn, E. Stotzer, D. Sule, and M. Asal, "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*, ser. HiPEAC'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 147–160.

[6] E. J. Stotzer and E. L. Leiss, "Co-design of compiler and hardware techniques to reduce program code size on a vliw processor," *CLEI Electronic Journal*, vol. 15, no. 2, pp. 2–2, 2012.

[7] S. Jee and K. Palaniappan, "Performance evaluation for a compressed-vliw processor," in *Proc. 2002 ACM Symposium on Applied Computing*, ser. SAC '02. New York, NY, USA: ACM, 2002, pp. 913–917.

[8] S. Haga, A. Webber, Y. Zhang, N. Nguyen, and R. Barua, "Reducing code size in vliw instruction scheduling," *J. Embedded Comput.*, vol. 1, no. 3, pp. 415–433, Aug. 2005.

[9] S. Haga and R. Barua, "Epic instruction scheduling based on optimal approaches," in *In Proc. First Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, 2001, pp. 22–31.

[10] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[11] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *J. Inf. Process.*, vol. 17, pp. 242–254, 2009.

[12] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. of SPIE Multimedia on Mobile Devices*, January 2007, pp. 65 070X–1 – 65 070X–11.

[13] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Elsevier, 2005.