Variable Length Instruction Compression on Transport Triggered Architectures

Variable Length Instruction Compression on Transport Triggered Architectures

Timo Viitanen · Janne Helkala · Heikki Kultala · Pekka Jääskeläinen · Jarmo Takala · Tommi Zetterman · Heikki Berg

Abstract The memories used for embedded microprocessor devices consume a large portion of the system's power. The power dissipation of the instruction memory can be limited by using code compression methods, which may require the use of variable length instruction formats in the processor. The power-efficient design of variable length instruction fetch and decode is challenging for static multiple-issue processors, which aim for low power consumption on embedded platforms. The power saved using compression is easily lost on inefficient fetch unit design design. We propose an implementation for instruction template-based compression and two instruction fetch alternatives for variable length instruction encoding on transport triggered architecture, a static multiple-issue exposed data path architecture. With applications from the CHStone benchmark suite, the compression approach reaches an average compression ratio of 44% at best. We show that the variable length fetch designs reduce the number of memory accesses and often allow the use of a smaller memory component. The proposed compression scheme reduced the energy consumption of synthesized benchmark processors by 15% and area by 33% on average.

T. Viitanen · H. Kultala · P. Jääskeläinen · J. Takala
Department of Pervasive Computing
Tampere University of Technology, Finland
E-mail: { timo.2.viitanen, heikki.kultala, pekka.jaaskelainen, jarmo.takala}@tut.fi

J. Helkala
Nokia Networks, Finland
E-mail: janne.helkala@nsn.com

T. Zetterman · H. Berg
Nokia Technologies, Finland
E-mail: {tommi.zetterman, heikki.berg}@nokia.com

1 Introduction

Modern systems-on-a-chip are becoming more and more advanced as an increasing amount of CMOS transistors can be fit on a single integrated circuit. Larger programs can be stored on the on-chip memories of devices, which consume a significant portion of the system's power and chip area. This makes it important to focus on reducing the memory size and accesses to reach a better power consumption level on the whole.

The power consumption of a circuit is divided into two categories: dynamic power and static power. The majority of the power dissipated in an integrated circuit is due to dynamic activity: net switching power, internal cell power and short-circuit power during logic transitions in the transistors [15]. However, the proportion of static power, i.e. leakage power dissipation is quickly growing towards half of all power consumed as the deep submicron technology nodes continue to decrease in size [2].

The program code, which is often stored on on-chip memory for embedded microprocessors, is an important aspect to consider for power savings. If *High Performance* (HP) SRAM is used on the chip, a substantial amount of current leakage is present [12]. Slower *Low Standby Power* (LSTP) SRAM can be used to avoid large leakage, but LSTP memory cells have higher on-currents, consuming more dynamic power as a trade-off. For either technology used, reducing the size of the memory via program code compression is beneficial: HP SRAM leaks less current when the memory module is smaller, while less dynamic power is used on expensive LSTP memory read-accesses if multiple instructions can be read per cycle.

Static multiple-issue architectures such as *Very Long Instruction Word* (VLIW), *Explicitly Parallel Instruction Computing* (EPIC) [13] and *Transport Triggered Architecture* (TTA) [9] can gain a lot of power savings from program code compression due to their long and loose instruction formats, which require large on-chip memories for the program code. The challenge brought by some code compression approaches, such as instruction template-based compression, is the requirement of variable length instruction fetch and decode units. They are especially difficult to design power-efficiently on embedded devices employing static-scheduled data paths, which have fairly simple fetch and decode hardware as the starting point. If a low-power variable length encoding support can be designed for the processor, power can be saved through sufficient minimum memory size reduction.

We propose an instruction template-based compression method for TTA processors, which is used for NOP removal, and implement power-efficient variable length instruction encoding fetch and decode stages required. Two alternative fetch unit designs are synthesized and benchmarked on a 40 nm ASIC technology for area and power consumption measurements. The efficiency of
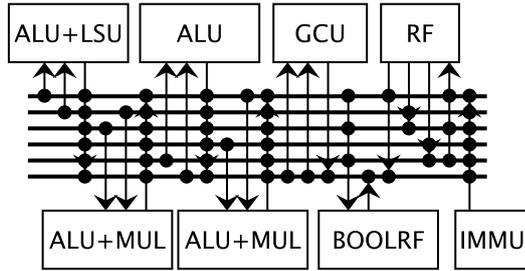
Fig. 1: Example TTA processor with 6 *interconnection buses*. Computational resources include four *arithmetic-logic units* (ALU), two *multipliers* (MUL), a *load-store unit* (LSU), a *general control unit* (GCU), a 64-entry *register file* (RF) and 2-entry *boolean register file* (BOOLRF), and finally an *immediate unit* (IMMU). Some resources are paired to form larger ALU+LSU and ALU+MUL function units. [8]

the code compression is measured by creating custom processors tailored for the applications in the CHStone [4] test suite and compressing each test program's code for the processors using four and eight different instruction templates. Feasibility of the implementation is evaluated by comparing the power consumption of each test's program memory pre- and post-compression with CACTI [16] and comparing the savings with the instruction fetch units' power consumption. LSTP SRAM cells are used for the program memory power estimation as they function at the 600 MHz clock frequency of the synthesized TTA processor. Finally, the implementations are benchmarked in complete TTA processors synthesized in a 28nm FD-SOI process technology.

This paper is an extended version of a conference article [8]. Adding to the original work, this paper includes an algorithm for template customization, and a more comprehensive evaluation with system-level benchmarks where the proposed fetch units are integrated into TTA processor test cases.

This paper is structured as follows. Section II is an overview of TTA. Section III introduces the compression approach and variable length instruction encoding. Section IV describes the hardware implementation. In section V, the proposed method is evaluated in terms of area, compression ratio and power consumption. Section VI discusses related work. Section VII concludes the paper.

2 Transport Triggered Architecture

TTA is a highly modular exposed-datapath relative of VLIW [1]. The main benefit of TTA comes from *software bypassing* operation results as inputs to other operations without going through the *register file* (RF). As many as 80% of RF accesses may be eliminated through bypassing [5]. Since the many-port RF is a major power sink in a VLIW, this allows significant power savings.

| 0 | slot 1 (18) | slot 2 (13) | slot 3 (18) | slot 4 (12) | slot 5 (18) | slot 6 (14) |

| 1 | slot 1 (18) | slot 2 (13) | slot 3 (18) | slot 4 (12) | 32–bit long immediate |

| guard(3) | | source(8) | | | destination(7) |

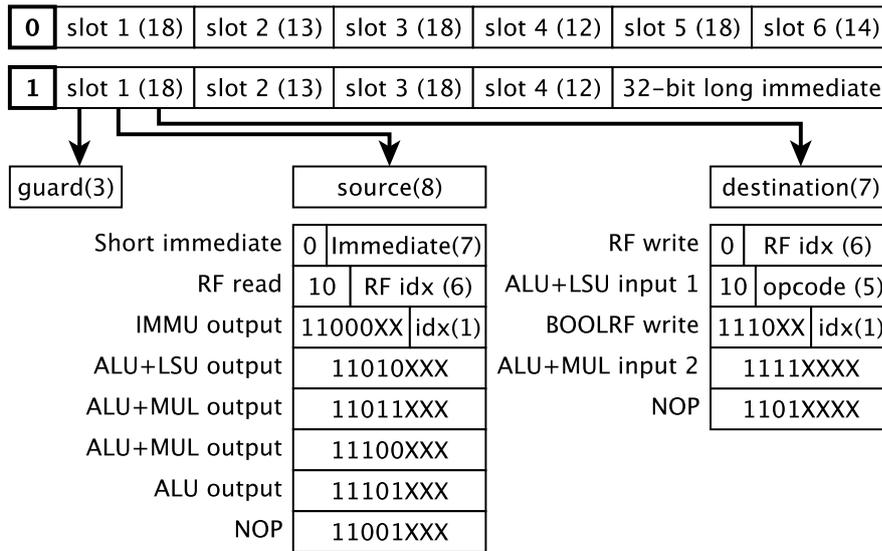| Short immediate | 0 | Immediate(7) | | RF write | 0 | RF idx (6) |
RF read	10	RF idx (6)		ALU+LSU input 1	10	opcode (5)
IMMU output	11000XX	idx(1)		BOOLRF write	1110XX	idx(1)
ALU+LSU output		11010XXX		ALU+MUL input 2		1111XXXX
ALU+MUL output		11011XXX		NOP		1101XXXX
ALU+MUL output		11100XXX				
ALU output		11101XXX				
NOP		11001XXX				

Fig. 2: Instruction encoding for the example TTA processor, using two instruction templates. [8]

Figure 1 shows an example TTA processor, comprised of several *function units* (FU), two register files and a 6-bus *interconnection network*.

The original work on TTA proposed various instruction encodings for TTAs including connection encoding, socket encoding and bus encoding [1]. All contemporary TTAs use bus encoding, in which each bus has a corresponding *move slot* in the instruction word which contains either a move instruction or a NOP. A partial bus encoding for the example processor is shown in Fig. 2. The move instruction consists of a *source* field, a *destination* field, and an optional *guard field* for predicated execution. Opcodes and RF indices are encoded in the source and destination fields. Moreover, our TTA processor design toolset [3] supports long immediate encoding using instruction templates, which replace some move slots with immediate values. The example encoding has two templates, toggled by a one-bit *template field* which is located at the instruction's MSB. This is the minimal template amount for encoding the move operations and the long immediate for this particular processor. The *template 0* is a *base template* which has a move operation for each of the six buses, while *template 1* replaces moves 5 and 6 with a special long immediate move.

3 Variable Length Instruction Compression

Instruction template-based compression removes a part of information in instructions, which can lead to a variable length format. The instruction formats

defined by the templates can be used, e.g., for NOP removal. Superscalar processors need to decode each incoming instruction and search for instruction level parallelism simultaneously, whereas in static architectures the operations are fetched as a bundle, readily scheduled for function units. The effective design of fetching, decompressing and decoding remains as the challenge for the static architectures, as well as the optimal selection of instruction templates for particular processor configurations.

3.1 Template-Based Compression

Instruction template-based compression approach re-encodes the processor's instruction set by adding a template field to the instructions. This template field is used for defining instruction formats which contain information for only a subset of the available fields in the architecture's instruction encoding. On TTA this compression can be employed by considering the available move slots in the processor as the information which to include in the different instruction formats. A template defines which move slots are included in the instruction format, hence the instruction's size is also tied to the template. The move slots that are left out of the selection of a template are implicitly assigned NOPs in the decoding stage, therefore called *NOP slots*.

The problem becomes the optimal selection of such instruction templates that the majority of the NOP operations can be removed from the program code with a minimal amount of templates, as the hardware complexity and power usage of the decoder grows with each additional template. There is a large design space of possible instruction template encodings and their compression ratio depends on the workload. For example, a template which can encode loads and stores is more efficient for data copying than branch-heavy control logic.

An example of template selection and NOP removal for a 5-bus TTA is displayed in Fig. 3. In this example, a large amount of NOPs are seen in four instructions. Two new instruction formats are assigned to the templates '10' and '11', which only use the buses A, B and D, E. The rest of the buses 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.

As seen in the example above, we merged the template previously used only for long immediate unit selection to be used for NOP removal as well. This means that in addition to the necessary base template which defines a move for each bus, at least one template is required by immediate unit selection if an immediate FU is available in the machine, reducing the amount of templates that can be used for NOP removal by 2. Due to the binary representation of templates in the template field, the amount of templates for each machine is optimally a power-of-two number.

2 Templates: 1 reserved for IMM

0	A	B	C	D	E
1	A	B	C	IMM	

4 Templates: 1 for IMM, 2 for NOP removal

00	A	B	C	D	E
01	A	B	C	IMM	
10	A	B			
11	D	E			

Uncompressed Program Memory

0	A1	B1	C1	D1	E1
0	NOP	B2	NOP	NOP	NOP
0	NOP	NOP	NOP	D3	E3
0	A4	NOP	NOP	NOP	NOP
0	A5	NOP	C5	NOP	NOP
1	A6	B6	C6	IMM	

Compressed Program Memory

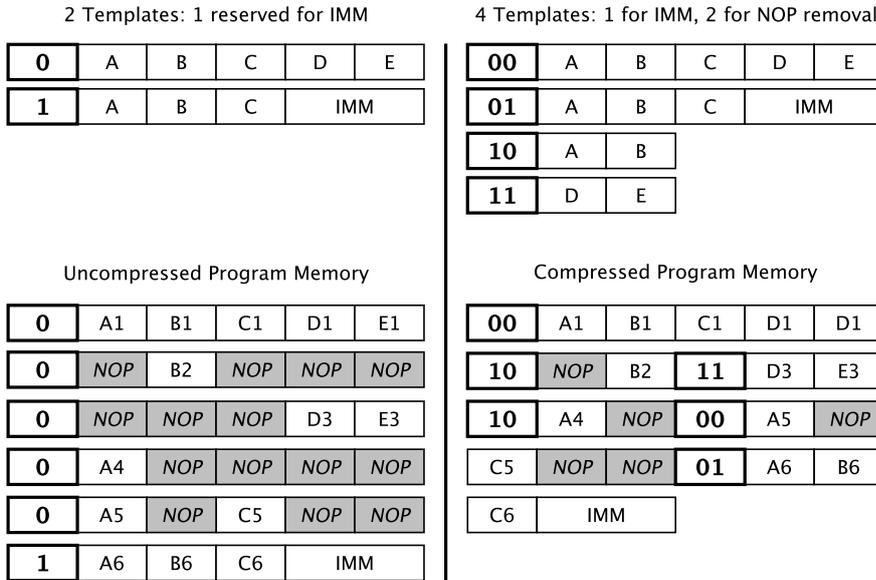| 00	A1	B1	C1	D1	D1
10	NOP	B2	11	D3	E3
10	A4	NOP	00	A5	NOP
C5	NOP	NOP	01	A6	B6
C6	IMM				

Fig. 3: A short program before (left) and after (right) assigning two new instruction formats, which define two move 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. [8]

The actual use of the templates for compression happens during program scheduling. Each instruction is attempted to match to the list of defined templates starting with the template with most NOP slots used, resulting in best compression. If an instruction can be matched with a NOP slot template, the given template is assigned to the instruction and the bits for each of the matched NOP moves are removed by a compressor during program image generation. The instruction template field is read during run-time in the decoder and the instruction is pieced back together from the variable length representation to the processor's maximum instruction length by inserting the missing NOP bits to the NOP slots. The complexity of the re-assembly depends on the amount of slots in the processor, number of templates, maximum instruction width and the bus widths [10].

In order to simplify the fetch unit, it is useful to constrain instruction templates to be multiples of a *quantum q*. The q and maximum instruction size I_{max} define the size of the shifter network generated. This q can be increased from instruction template bit field width $IT_w + 1$ to I_{max} for the least complex shifter network, but worst decompression ratio of the instruction template compression. If q equals I_{max}, the instructions become fixed length. In practice we use power-of-two values for q and I_{max} in order to further simplify fetch logic.

3.2 Template Customization Algorithm

This section describes an algorithm for optimizing instruction template selection for small to medium-sized TTAs based on a given target program. The algorithm first compiles the program and computes a histogram of NOP patterns found in the machine code, proceeds to evaluate all possible templates, and selects the template which gives the best compression ratio. Templates are added in this manner until an user-set target is reached.

Algorithm 1 Template Customization

1: program ← compile(test_program)
2: Initialize slot_histogram[2^{num_buses}] to 0.
3: **for all** instuctions in program **do**
4:     slot_histogram[slotMask(instruction)] ++
5: **end for**
6: Initialize instruction_sizes[2^{num_buses}] to full instruction size.
7: **for** tmpl= 1 **to** target_nof_templates **do**
8:     best_pattern ← 0
9:     best_template_size ← 0
10:    best_program_size ← ∞
11:    **for** pattern = 0 **to** $2^{num\_buses} - 1$ **do**
12:        template_size ← computeTemplateSize(pattern);
13:        program_size ← 0
14:        **for** pattern2 = 0 **to** $2^{num\_buses} - 1$ **do**
15:            n ← slot_histogram[pattern2]
16:            **if** $(pattern \land \neg pattern2) = 0$ **then**
17:                program_size + = n × min(instruction_sizes[pattern2], template_size)
18:            **else**
19:                program_size + = n × instruction_sizes[pattern2];
20:            **end if**
21:        **end for**
22:        **if** program_size < best_program_size **then**
23:            best_pattern ← pattern
24:            best_template_size ← template_size
25:            best_program_size ← program_size
26:        **end if**
27:    **end for**
28:    **for** pattern2 = 0 **to** $2^{num\_buses} - 1$ **do**
29:        **if** $(best\_pattern \land \neg pattern2) = 0$ **then**
30:            instruction_sizes[pattern2] ← best_template_size
31:        **end if**
32:    **end for**
33: **end for**

On line 4, the function *slotMask* constructs a binary number whose bit-pattern corresponds to active move slots. Bits corresponding to NOP slots are set to 0. On line 12, the function *computeTemplateSize* computes how many bits are needed to encode a template with the given NOP pattern, rounded up to the next multiple of the user-defined quantum. The bitwise logic on line 16 tests whether the template described by the bitmask *pattern* can encode an

instruction with the mask *pattern2*. Multiple target programs can be supported by computing a weighted average of their NOP pattern histograms.

The algorithm has an unfavorable complexity class $O(2^b)$ with respect to the number of transport buses b in the target architecture, but with most practical TTAs, the exponent is small enough that compiling the test programs takes more time than the iteration. However, some proposed TTAs have tens of move slots that program buses of very limited connectivity. These machines would require a different exploration method.

3.3 Variable Length Instruction Encoding

Variable length instruction encoding's main purpose is to encode some instructions in a smaller amount of bits than others to save memory required to represent the program. This immediately introduces a problem: since instructions become tightly packed in the memory, they might no longer be aligned at the beginning of memory words for convenient fetching and execution.

The incoming instructions must be found from the memory words being fetched and expanded back to the full instruction length before decoding. In order to be able to splice the bit patterns into decipherable instructions while guaranteeing continuous execution, a buffer is required in the fetch unit. The design of this buffer is crucial, because its complexity can grow rapidly on the logic level if an inefficient implementation is used, consuming more dynamic power than is saved by the reduced instruction bits. The decoder's complexity must be taken into account as well, as its size will increase undesirably unless constraints are set on the design.

Finally, a method for handling random access support is required, i.e. how to execute control flow operations such as *jump* or *call*, which require finding an instruction to execute from the misaligned memory. Especially the execution of calls is complicated, because the return address of the program flow must be recorded. In a fixed length instruction architecture, saving the return address is as simple as saving the program counter's value, because each instruction is neatly aligned in the memory. In a variable length fetch design, there's an unknown amount of instructions with unknown sizes remaining in the buffer when a call is detected. A pure hardware solution to return address calculation requires knowing which memory address each instruction comes from, the track-keeping of which bloats the hardware.

We investigated two alternative fetch units to estimate the power consumption of different buffer architectures: *ring buffer fetch* and *shift register fetch*. The fetch units are capable of continuous instruction splicing from the memory words and both handle the execution of jump instructions. On-hardware return address calculation was implemented for the former design. Due to the additional complexity from recording the return addresses, we found out that it is easier to implement function calls and their return stack completely in software, therefore, the shift register design was designed without a *call* operation.

Our solution for random access support on TTA is addressed partially by the compiler and partially inside the fetch unit. Jumps are supported by aligning all control flow operation targets in the program code at memory addresses. This means that the code is divided into blocks which are mostly misaligned due to variable length instructions, but occasionally aligned again at the locations of jump targets. An issue with jump target alignment is that the instruction prior to an aligned instruction may contain redundant information, *padding bits*, which are not to be executed. In our implementation, we append a padding indication bit to the MSB-end of each instruction, indicating whether the current instruction contains padding bits in the memory word after the actual instruction bits. This bit is *'1'* if padding bits exist.

4 Implementation of Hardware Unit

The major changes to TTA processor micro architecture required for variable length instruction support are in the decoder and the fetch units. The changes in the decoder are generated per-processor according to NOP instruction templates. The decoder contains a look-up table -based re-assembly network for the instruction template decompression.

The names of the two fetch alternatives, *Ring Buffer* (RB) and *Shift Register* (SR), describe how the fetch unit handles the incoming memory words. RB uses a multiplexer network which targets different parts of the buffer for writing and reading. SR uses a shifter network to store and consume the instructions in the buffer in a *First In First Out* manner. The former is a minimalist approach that has a buffer width of only $2I_{max}$, while the latter's buffer width is $3I_{max}$ and has more relaxed control logic.

The RB fetch unit's basic buffer structure during execution is displayed in Fig. 4. It was designed based on the constraint that without needing to stall during execution, a minimum buffer width of $2I_{max}$ is required for continuous instruction fetching. Its internal logic cycles a *Read Pointer* (RP) to point at the MSB of the current instruction being read from the buffer, and *Write Pointer* (WP) to define whether the next memory word is to be assigned to the upper or lower half of the buffer. The granularity of the RP and the complexity of the internal multiplexer structure are directly affected by the minimum instruction size q. The content of the buffer is stored in a variable and rotated each cycle by a *rot_r* function with $RP + 1$ amount to align the instruction being read to the buffer's MSB for output. The *rot_r* operation is needed when the content inside the buffer becomes misaligned, causing instructions to wrap around from the LSBs of the buffer to the MSBs.

Because of the buffer's limited size, need for uninterrupted execution and the memory read latency of one cycle, the implementation needs to check whether the current instruction pointed by RP is large enough to free the buffer half targeted by WP for the next cycle. The size of the variable length instruction is decoded with the help of a look-up table during the same cycle as an instruction is read out to determine buffer fullness, next RP calculation and
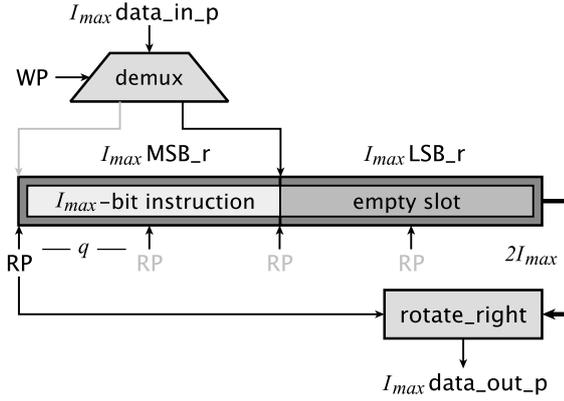
Fig. 4: The structure of the ring buffer fetch unit for $I_{max} = 2q$. An instruction has been written to the MSB register on the previous cycle and is currently being read out of the buffer, indicated by the RP. WP is assigning the next memory word to the LSB register. The RB's contents are rotated with a *rot_r*-function by $RP + 1$ for output. Other possible RP locations are defined by the q. The data goes directly to fetch output port from the buffer. [8]

WP selection. Due to RB's minimal size, the cycle-accurate internal control logic becomes complicated. The WP and RP synchronization after a control flow instruction ultimately required the use of one stall cycle to simplify the control logic and to flush the buffer.

The shift register design was created to simplify the internal logic of the fetch unit and to address the RB's stall cycle. The buffer width was increased by one instruction to $3I_{max}$ to alleviate the need for complex control logic, using the data_in port of the unit as one of the buffer slots. The wider buffer simplifies checking buffer fullness and reduces the critical path inside the fetch unit, allowing higher clock frequencies to be reached. However, an extra register with the width of I_{max} is required at the output, as otherwise data_in would be routed directly to data_out, disrupting the processor's pipeline.

An example of the SR unit is displayed in Fig. 5. The memory words are always read to the LSB-end of the buffer. Instead of a RP, the SR tracks the current instruction to be forwarded with a *Shift Amount* (SA) value. This can be imagined as a virtual RP with the conversion: $SA = I_{max} - (RP + 1)$. For output, the entire buffer's contents are stored into a variable which is shifted left by SA, aligning the current instruction pointed by RP to MSB.

Every cycle the buffer's contents are shifted left by I_{max} bits, and every cycle an instruction is consumed from the buffer. If only instructions with the size of I_{max} are written in and read out, the buffer stays at equilibrium. The buffer begins to fill up when smaller instructions are written in, with the RP approaching the left side of the buffer. When an instruction the size of I_{max}
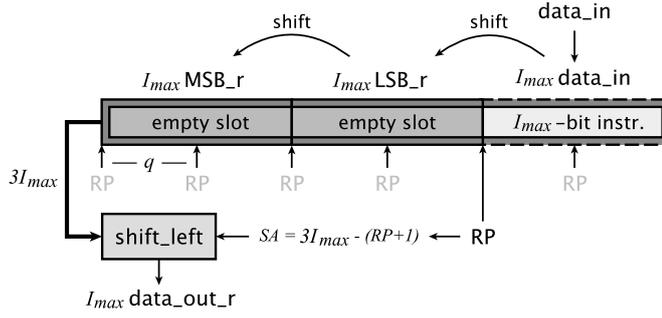
Fig. 5: The shift register fetch structure for $I_{max} = 2q$. An instruction is seen in the data_in port and treated as a part of the buffer. The incoming instruction is I_{max} length and is shifted left to the MSB for output, indicated by the SA value, which is calculated from a virtual RP value. Other possible SA amounts are defined by the q. The data propagates through a data_out register before output. [8]

would no longer fit in, the fetching is stalled and instructions are read out from the buffer until an instruction the size of I_{max} fits to the buffer again.

5 Evaluation

Two TTA processors with I_{max} of 128 and 256 bits were customized for a subset of the CHStone test suite to measure the compression efficiency of two different instruction template compression configurations with four and eight templates. In these configurations, two and six templates were used for NOP removal, respectively. The power consumption of the program memory was estimated with web-based CACTI 5.3 (rev 174) pre- and post-compression. The power consumption of the two fetch designs were measured with three different quanta using synthetic tests to scope out the worst case power dissipation. Additionally, the used chip area of the designs are provided.

5.1 Compression Efficiency

We used the CHStone C-based high-level synthesis test bench for measuring the compression ratio of the instruction template-based compression. A TTA machine with an I_{max} of 256 was customized for the benchmarks. We started with a 6-issue VLIW equivalent processor architecture and reduced it by combining rarely used buses until a 256-bit instruction length was reached. This process is described in more detail in [17]. The benchmark programs' uncompressed sizes were in the range of 14–50 KB, with the exception of the large *jpeg* test which was approximately 376 KB.
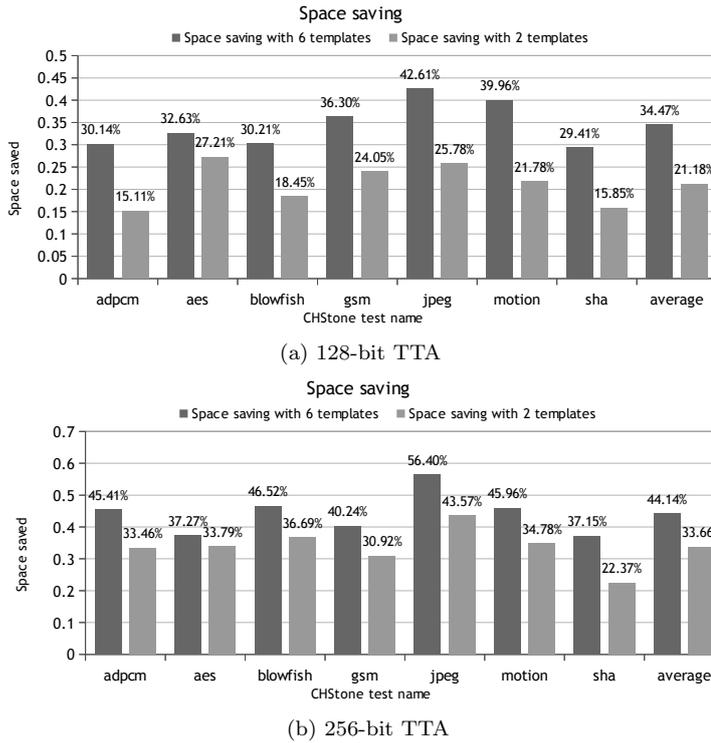
(a) 128-bit TTA

(b) 256-bit TTA

Fig. 6: Compression ratios in the CHStone benchmark.

We created instruction templates using the algorithm described in Section 3.2. The whole CHStone testbench was used as the target workload. For the machine with 2 NOP templates, we set q to 64, and the algorithm produced two short templates of lengths 64 and 128. For the machine with 6 NOP templates, we reduced q to 32, resulting in a range of templates between 32 and 192 bits, two of which encode a 32-bit long immediate.

When compiling the test programs, we used a compiler pass which attempts to reschedule instructions to utilize available instruction templates better [10]. This improved compression ratios by 2-3%.

Resulting compression ratios are shown in Fig. 6. The 2-template TTA reached an average ratio of 33% and a maximum of 43%, and the 6-template TTA improved to an average of 44% and a maximum of 56%. Some ratios are lower than in [8], which appears to be due to a newer version of the compiler generating up to 16% more compact baseline code, which has fewer NOPs to compress.

5.2 Program Memory Power Consumption

The power consumption of the program memory when assuming the componen is scaled down to the minimum required by the program at hand was estimated with CACTI before and after instruction template compression. ITRS-LSTP was chosen for the SRAM transistor type, interconnect projection type was set to conservative and wire outside of mat as semi-global. Technology node used was 40 nm and temperature was set to 300K for all measurements. The number of bits read out per cycle was matched with the memory width, i.e., the full instruction length of 256 bits. One read/write port was used. For the estimation, SRAM size was set to exactly the size of the program, which is unrealistic as SRAM is typically not manufactured in arbitrary sizes, but gives an estimate of power savings achieved by the instruction template compression. Finally, the total dynamic read power per read port P_{dyn} is calculated with

$$P_{dyn} = \frac{E_{dyn}}{t} = E_{dyn} f_{clk} \tag{1}$$

where E_{dyn} is the dynamic energy per read port estimated by CACTI and f_{clk} is the clock frequency of 600 MHz for the SRAM, which is the target frequency used in the synthesis of the fetch units. Since LSTP SRAM cells were used in the measurements, the portion of leakage power was much less than 0.1% of the total power consumed and could be left out of consideration. The overhead of the instruction template bits and padding bits required by the proposed TTA's variable length instruction format are taken into account in the results, while their effect is minimal ($< 1\%$).

The power savings per CHStone benchmark are presented in Fig. 7b. The difference between 2 and 6 instruction templates used for the NOP removal is also visible in the power results: 6 templates covered much more of the NOP moves, allowing much better compression ratio and smaller SRAM memory size. In order to compensate for the *jpeg* test results, where the benchmark contains a significantly larger instruction count, a geometric mean of the power saved in all the tests is presented: 4.74 mW with six instruction templates and 3.91 mW with just two instruction templates. The power saved was not linear with the amount of bytes reduced from the program code, because the size of the program memory affects the consumption, especially when power of two values are crossed. Despite approximately 21 KB was saved in the *aes* test with six templates, only 3.45 mW less power was consumed, while 6.42 mW of power was saved in the *blowfish* test with 13 KB memory reduction. As examples, the program code for *aes* could be fitted on a 32 KB memory instead of 64 KB after compression, and *blowfish* on 16 KB instead of 32 KB.

Since SRAM memory is typically not manufactured in arbitrary sizes, the power saved when switching to a half smaller memory size was estimated with CACTI with the same parameters as for instruction compression. These results are presented in Fig. 8. The chart shows that a considerable saving is seen each time when a reduction is possible, until 16 KB. This highlights that a good amount of power can be saved even if the program image does
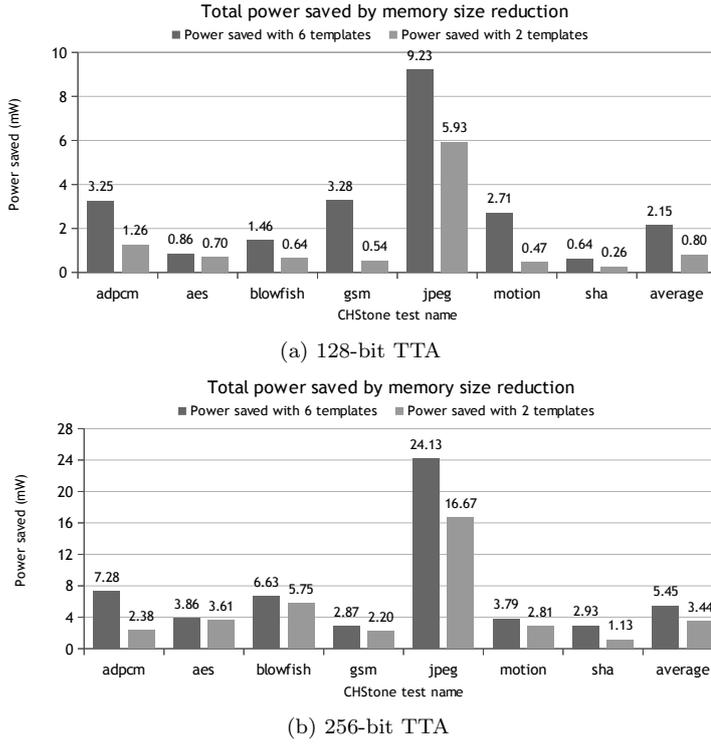
(a) 128-bit TTA

(b) 256-bit TTA

Fig. 7: Power saved with instruction template compression, using 2 and 6 instruction templates.

not compress significantly, but if it compresses sufficiently to fit on a smaller memory module.

5.3 Fetch Unit Power Consumption

The original and the two alternative fetch designs were synthesized on a 40 nm standard cell technology, using quanta of 2, 32, and 128 bits. The target clock speed was set to 600 MHz and each variable length design variant was subjected to three synthetic test cases, which explored the units' worst case power consumption. The three tests cases consisted of a varying degree of $I_{max} = 256$-bit and quantum (q) length instructions: Either all q-length, all I_{max}-length or alternating I_{max}- and q-length instructions.

The test result with the highest power consumption for each design variant is displayed in Fig. 9. In most cases, the worst power consumption was seen when the fetch unit had to repeatedly fetch and handle q-length instructions, as its internal multiplexer and shifter structures had to operate on bits. The best results are seen with a q of 128 bits, which is half of the maximum instruction
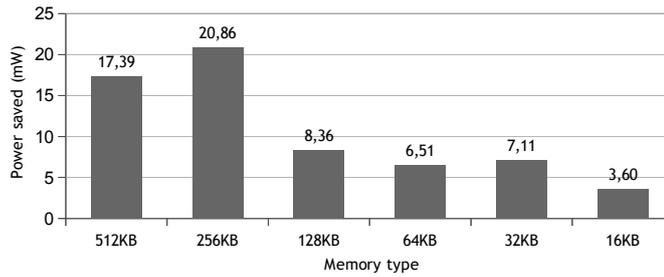
Fig. 8: Memory power saved when LSTP SRAM memory size is reduced by half. [8]
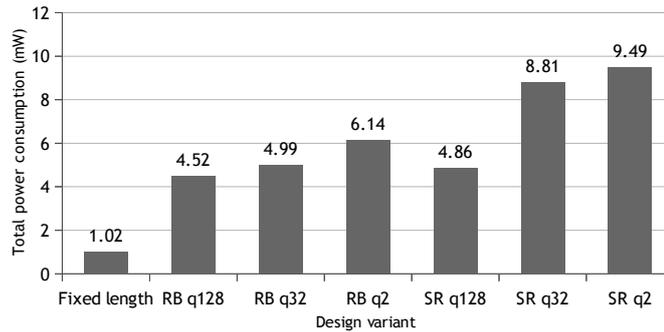

Fig. 9: Fetch units' total power consumption with quanta (q) of 2, 32 and 128 bits, showing worst case test results. [8]

length. On the ring buffer design, the other q-values follow closely, while the power consumption grows rapidly on the SR design.

At best, the variable length fetch unit requires 3.50 mW of extra power to operate at worst case, when the q of 128 bits is used. However, a much better compression ratio is seen with a q of 32 bits, which is the quantum used in the instruction template compression results of this paper. The SR approach for a q of 32 consumes more power than would be saved with 6 instruction templates on average, unless a reduction from a 128 KB memory or larger to a smaller category can be made. The ring buffer is much more efficient, reaching the break-even of average power savings when just two instruction templates are used for compression.

As long as a SRAM memory power saving of approximately 4.0 mW or more is reached with compression, the variable length ring buffer fetch's usage is favorable. These results do not include the overhead from the instruction template decompression which is integrated in the decoder unit, which most likely consumes some more dynamic power to re-assemble the decompressed instructions. This can be projected to be a fairly efficient operation, as it is a
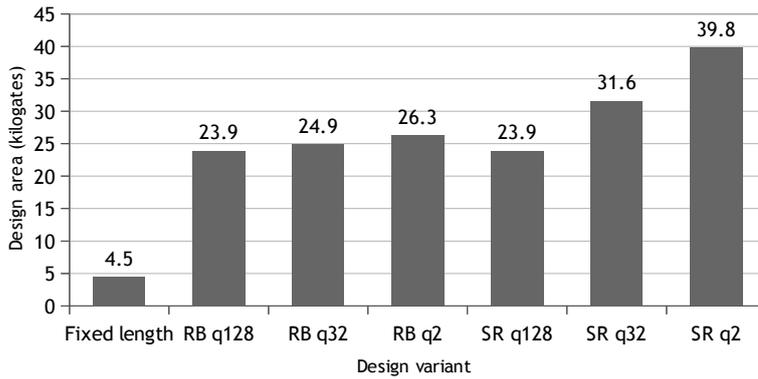
Fig. 10: Fetch units' area in kilogates with quanta (q) of 2, 32, and 128 bits, using target clock speed of 600 MHz. [8]

multiplexer network which is simplified by choosing a reasonably large q and using few instruction templates.

5.4 Chip Area

The area of each of the fetch designs was collected from the 40 nm standard cell synthesis results and is presented in Fig. 10 in kilogates. The fetch designs were synthesized successfully for the performance range of 500–1000 MHz.

A similar trend is seen in the area as in the power consumption: the SR designs with a small q grow rapidly, while the ring buffer stays more compact even when q is increased. Worth noting is that SR design's area exploded when the maximum instruction length of a *power of two value* $- 1$ was used, while the ring buffer's area followed a linear trend with maximum instruction size increase. It is interesting to note that at their simplest form at q of 128, the ring buffer and SR are of similar size. This implies that the extra logic the ring buffer requires to function roughly equals the extra logic required by the SR design's buffer, which is one instruction longer. Finally, with the least logic generated from a q of 128 bits, both of the new designs are 431% larger than the original fetch design, which only handles fixed length instructions.

5.5 System Benchmark

The previous synthetic benchmarks characterize the proposed fetch units in general terms, but excludes some dynamic behavior which may have a large effect on the power consumption of a complete system. For one example, the power consumption of the instruction decoder varies depending on the runtime mixture of templates it has to expand into full instructions. On the other hand,

if many of the executed instructions use small templates, the processor needs fewer memory accesses to fetch them, reducing SRAM power.

In order to gauge the relative importance of these dynamic effects, we performed synthesis and power estimation of processors with different memory sizes and fetch units. The processors were synthesized on a 28nm FD-SOI process technology using Synopsys Design Compiler. We used operating conditions of 1V, 25C, and typical resistance-capacitance models. Basic power optimizations are enabled, including clock gating and poly-bias, which replaces multi-V_{th} optimization in the target process. Design Compiler is run in a topographical mode, which provides estimates of wire delays and clock tree power consumption. Clock frequency is constrained to 1GHz, except for the cases with very large memories, which are set to 0.8GHz.

We concentrate on the 128-bit wide instruction processor, since the 256-bit processor rarely achieves good utilization with this benchmark: there is only an average performance penalty of 10%. The processor is synthesized with the two proposed fetch options and a baseline fetch without variable-length coding. Each option is synthesized with several instruction memory sizes, so that each benchmark program can be run on the smallest memory that can accommodate it, rounded to the next power of two. A memory compiler which prioritizes, in order of decreasing importance, density, speed, leakage power and dynamic power, is used to generate SRAM blocks of sizes 128x512, 128x1024, 128x2048 and 128x4096. The *jpeg* test requires more memory than the largest block, so in this case the memory is constructed in RTL from multiple blocks and a multiplexer. The compiler produces timing and power models which are used in simulation. Data memory is not included in synthesis, since it is unaffected by the choice of instruction compression.

Power estimation is based on switching activity recorded from RTL simulation of the CHStone benchmark on Mentor Modelsim. The tests *blowfish*, *jpeg* and *sha* have long runtimes, so we simulate them for only 10^5 clock cycles. Instruction-level simulation shows that the utilization profile of computational resources after this period is similar to a complete run. The shorter tests are run until completion.

Results are shown in Table 1 and in Figure 11. Our memory components have an approx. 50% higher dynamic power than estimated with CACTI for 32nm, balanced by a one-third smaller area and a shorter access time. This is likely due to design tradeoffs in the memory compiler, which favors density at the cost of dynamic power. Another factor is that our simulated dynamic activity may differ from the statistical model in CACTI that does not account for the switching activity of real application execution. As a result, the instruction memory dominates the power profile of each processor. Since compressed programs have fewer instruction memory accesses, compression gives 5-9% power savings even in the *aes* and *blowfish* tests, where the instruction memory cannot be shrunk since the program size does not fall past a power-of-two boundary.

The RB and SR designs both reduce instruction SRAM power by an average of 32% and area by an average of 30%, resulting in overall power savings
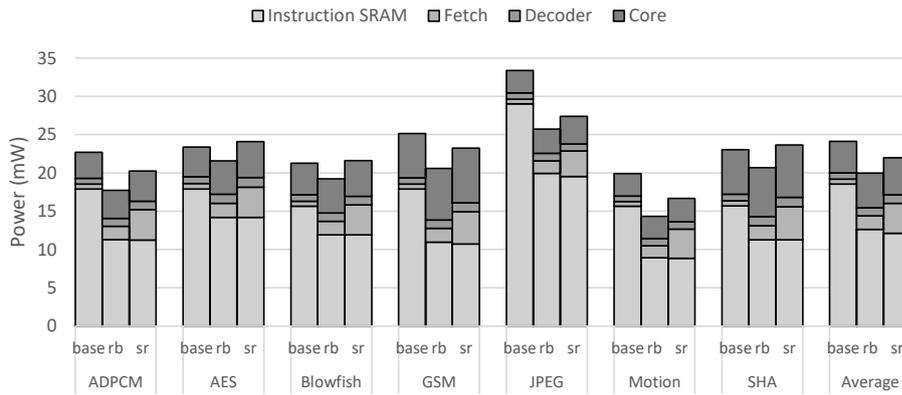
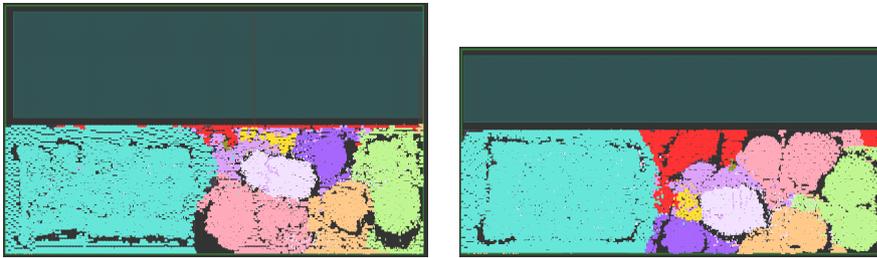Fig. 11: Power estimation results with different fetch units on the 128-bit TTA and the CHStone benchmark.

Fig. 12: Processor designs after routing: baseline fetch with 1024-word memory (left) and RB fetch with 512-word memory (right). The RB design has a prominent fetch unit (red), but takes up less area overall.

of 16.9% and an area reduction of 32%. However, the shift register incurs a considerable power penalty in the instruction fetch. Due to stalls on jump instructions, the circular buffer design incurs a slowdown of 0.7-4.1%. Consequently, the shift register design may be preferable in performance-critical applications. Taking the slowdown into account, the average energy savings per program run for the RB design are 14.6%. In the *jpeg* case, instruction compression allowed the ring buffer design to reach a higher clock frequency than the baseline. Howerer, this is likely an unrealistic scenario, since a memory of this size would be accessed through a cache hierarchy.

In order to improve confidence in the synthesis results, we took two of the designs through a rudimentary place and route flow in Synopsys IC Compiler and ran power estimation with the *sha* case. We targeted an utilization of 70% and used congestion-placed routing. The total power of each design was approx. 5% higher after routing and clock tree synthesis. Figure 12 shows plots of the routed designs with the design hierarchy highlighted.

	ADPCM			AES		
	base	rb	sr	base	rb	sr
Power(mW)						
SRAM	17.90	11.32	11.25	17.91	14.19	14.21
Fetch unit	0.65	1.71	3.97	0.70	1.86	3.93
Decoder	0.76	1.04	1.10	0.91	1.19	1.25
Core	3.40	3.67	3.92	3.86	4.34	4.71
Total	22.70	17.73	20.24	23.37	21.58	24.10
Power diff.		-21.9%	-10.8%		-7.7%	3.1%
Energy diff.		-20.2%	-10.8%		-5.6%	3.1%
Stall rate	0.0%	2.1%	0.0%	0.0%	2.3%	0.0%
Access rate	100.0%	71.3%	70.7%	100.0%	78.4%	78.1%
Memory size (kB)	32	16	16	32	32	32
	BLOWFISH			**GSM**		
	base	rb	sr	base	rb	sr
Power(mW)						
SRAM	15.68	11.91	11.91	17.90	10.95	10.74
Fetch unit	0.64	1.79	3.93	0.67	1.82	4.21
Decoder	0.83	1.06	1.11	0.81	1.08	1.17
Core	4.14	4.46	4.68	5.77	6.74	7.12
Total	21.28	19.22	21.62	25.15	20.60	23.23
Power diff.		-9.7%	1.6%		-18.1%	-7.6%
Energy diff.		-9.0%	1.6%		-14.7%	-7.6%
Stall rate	0.0%	0.8%	0.0%	0.0%	4.1%	0.0%
Access rate	100.0%	75.2%	75.0%	100.0%	68.9%	67.3%
Memory size (kB)	16	16	16	32	16	16
	JPEG			**MOTION**		
	base	rb	sr	base	rb	sr
Power(mW)						
SRAM	29.01	19.94	19.52	15.67	8.91	8.86
Fetch unit	0.64	1.65	3.34	0.61	1.60	3.81
Decoder	0.79	0.96	0.92	0.71	0.95	0.97
Core	2.94	3.17	3.62	2.92	2.87	3.06
Total	33.38	25.72	27.40	19.91	14.33	16.69
Power diff.		-23.0%	-17.9%		-28.0%	-16.2%
Energy diff.		-19.1%	-17.9%		-25.8%	-16.2%
Stall rate	0.0%	5.1%	0.0%	0.0%	3.1%	0.0%
Access rate	100.0%	69.0%	67.7%	100.0%	60.8%	60.1%
Memory size (kB)	256	128	128	16	8	8
	SHA			**Average**		
	base	rb	sr	base	rb	sr
Power(mW)						
SRAM	15.69	11.29	11.29	18.54	12.64	12.12
Fetch unit	0.68	1.82	4.31	0.66	1.75	3.93
Decoder	0.88	1.18	1.21	0.81	1.07	1.10
Core	5.80	6.42	6.85	4.12	4.52	4.85
Total	23.04	20.70	23.66	24.12	19.98	21.87
Power diff.		-10.1%	2.7%		-16.9%	-6.5%
Energy diff.		-8.1%	2.7%		-14.6%	-6.5%
Stall rate	0.0%	2.3%	0.0%	0.0%	2.8%	0.0%
Access rate	100.0%	77.2%	76.7%	100.0%	71.5%	70.8%
Memory size (kB)	16	8	8			

Table 1: Synthesis and power estimation results.

6 Related Work

Program code compression has been vastly researched and eventually adopted in many instruction set architectures. Similarly, variable length instructions are used in many architectures such as ARM Thumb [14], EPIC[13] and x86, not only for NOP-removal, but also for executing other instructions of varying sizes. Some papers introducing new compression methods based on variable length instruction encoding list impressive compression ratios, but do not show either performance, area, or power consumption results, such as in [11].

Heikkinen evaluated instruction template-based compression in [7, ?]. The compression was performed with 2–32 templates on DSP tests with different processor configurations. In his benchmarks, the designs consumed more power than saved, despite a compression ratio of 46,5% was reached with maximum templates for the processors. The results in this paper are more favorable for several reasons: The quantum in [6] is limited to 16 bits, while we explored the effect of quanta of 2, 32, and 128 bits, which are all power of two factors of the selected maximum instruction length of 256 bits. The two processor configurations in [6] had instruction widths of 127, and 192 bits, which are not a power of two, causing more complex hardware structures to be generated in synthesis. Also both of the fetch designs in this work contained less registers due to more optimized design.

A very similar instruction template-based compression is used in EPIC[13], where two variable length encoding schemes can be used to eliminate NOPs from the program code: MultiTemplate and VariOp. In addition, EPIC's fixed length MultiOp instruction format contains a field for how many full NOP instructions are to be issued after the current instruction, allowing instructions which contain only null data transports to be omitted completely from the program code. MultiTemplate instruction format involves the use of templates, each of which defines a subset of function units to target with the operations of the variable length instruction. The rest of the FUs are implicitly provided with a NOP. The VariOp instruction format is different, as it permits any subset of operation slots to be included within any instruction up to the maximum FU amount. Each operation is explicitly targeted to a FU and the remaining empty operation fields are implicitly filled with NOPs on a per instruction basis.

7 Conclusion

This paper proposes a solution for compressing NOP instructions on TTA architectures using a variable length instruction encoding approach and instruction template-based compression. The compression reduces the required SRAM memory size required for the program code, lowering the total power consumption of the processor. Two instruction fetch designs were proposed: the ring buffer and shift register buffer. The former is a more minimalist design with a buffer of two maximum length instructions. The latter uses one

more buffer slot to reduce the control logic complexity and reach a better clock frequency for the processor. A simple template customization algorithm was also proposed.

The compression achieved 44% program size reduction on average with 6 NOP removal templates and 37% reduction with 2 templates. The fetch designs consume an extra 3.50 mW of power at minimum on a TTA processor with 256-bit maximum instruction length. Even though the savings from the template compression do not always directly surpass the extra power consumed by the fetch unit in our benchmark suite, the target program can often be fitted on a half smaller memory module after compression. For SRAM memory sizes between 32–512 KB and beyond, this is reduction is sufficient to benefit from the variable length architecture.

In a core-level benchmarks, the ring buffer design saved energy even in cases where it was not possible to shrink the memory module, by reducing the number of memory accesses, as multiple instructions could be fetched with a single access. In the cases where the memory module could be shrunk, savings were more substantial. Even a small, high-density memory is comparable in area to the benchmarked processor, therefore, the potential area savings are significant. The shift register design gave less consistent energy savings, but may be preferable in performance-critical designs since it does not incur stall cycles on jumps.

As future work we are investigating compiler techniques to take better advantage of the NOP templates. It would also be interesting to combine the variable-length instruction compression with a small L0 instruction cache or a loop buffer for further power savings.

References

1. Corporaal, H.: Microprocessor Architectures: From VLIW to TTA. Wiley (1997)
2. Deepaksubramanyan, B. S. and Nuñez, A.: Analysis of subthreshold leakage reduction in CMOS digital circuits. In: Proc. Midwest Symp. Circ. Syst., pp. 1400–1404. Montreal, QC (2007). DOI 10.1109/MWSCAS.2007.4488809
3. Esko, O., Jääskeläinen, P., Huerta, P., de La Lama, C., J.Takala, Martinez, J.: Customized exposed datapath soft-core design flow with compiler support. In: Proc. Int. Conf. Field Programmable Logic and Applications, pp. 217–222. Milan, Italy (2010). DOI 10.1109/FPL.2010.51
4. Hara, Y., Tomiyama, H., Honda, S., Takada, H.: Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. J. Inf. Process. **17**, 242–254 (2009). DOI 10.2197/ipsjjip.17.242
5. He, Y., She, D., Mesman, B., Corporaal, H.: MOVE-Pro: A low power and high code density TTA architecture. In: Proc. Int. Conf. Embedded Comput. Syst.: Architectures, Modeling and Simulation, pp. 294–301. Samos, Greece (2011)
6. Heikkinen, J.: Program compression in long instruction word application-specific instruction-set processors. Ph.D. thesis, Tampere Univ. Tech., Finland (2007)
7. Heikkinen, J., Takala, J., Corporaal, H.: Dictionary-based program compression on customizable processor architectures. Microprocessors and Microsystems **2**, 139–153 (2009)
8. Helkala, J., Viitanen, T., Kultala, H., Jääskeläinen, P., Takala, J., Zetterman, T., Berg, H.: Variable length instruction compression on Transport Triggered Architectures. In: Proc. Int. Conf. Embedded Comput. Syst.: Architectures, Modeling and Simulation, pp. 149–155. Samos, Greece (2014)

9. Jääskeläinen, P., Guzma, V., Cilio, A., Takala, J.: Codesign toolset for application-specific instruction-set processors. In: Proc. SPIE Multimedia on Mobile Devices, pp. 65,070X–1 – 65,070X–11. San Jose, USA (2007)

10. Kultala, H., Viitanen, T., Jääskeläinen, P., Helkala, J., Takala, J.: Compiler optimizations for code density of variable length instructions. In: Proc. IEEE Workshop on Signal Processing Systems, pp. 1–6. Belfast, UK (2014)

11. Pan, H., Asanović, K.: Heads and tails: A variable-length instruction format supporting parallel fetch and decode. In: Proc. Int. Conf. Compilers Arch. Synthesis for Embedded Syst., pp. 168–175. Atlanta, Georgia (2001). DOI 10.1145/502217.502244

12. Pilo, H., Adams, C.A., Arsovski, I., Houle, R.M., Lamphier, S.M., Lee, M.M., Pavlik, F.M., Sambatur, S.N., Seferagic, A., Wu, R., Younus, M.I.: A 64Mb SRAM in 22nm SOI technology featuring fine-granularity power gating and low-energy power-supply-partition techniques for 37% leakage reduction. In: Proc. IEEE Int. Solid-State Circ. Conf. Digest Tech. Papers, pp. 322–323. San Francisco, USA (2013). DOI http://dx.doi.org/10.1109/ISSCC.2013.6487753

13. Schlansker, M.S., Rau, B.R.: EPIC: An architecture for instruction-level parallel processors. Tech. rep., Hewlett-Packard (2000)

14. Segars, S., Clarke, K., Goudge, L.: Embedded control problems, thumb, and the ARM7TDMI. IEEE Micro **15**(5), 22–30 (1995). DOI 10.1109/40.464580

15. Sheng, L., Ahn, J.H., Strong, R., Brockman, J., Tullsen, D., Jouppi, N.: McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In: Proceedings of Annual International Symposium on Microarchitecture, pp. 469–480. New York, USA (2009)

16. Thoziyoor, S., Ahn, J.H., Monchiero, M., Brockman, J.B., Jouppi, N.P.: A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In: Proc. ACM/IEEE Int. Symp. Comp. Arch., pp. 51–62. Beijing, China (2008)

17. Viitanen, T., Kultala, H., Jaaskelainen, P., Takala, J.: Heuristics for greedy transport triggered architecture interconnect exploration. In: Proc. Int. Conf. Compilers, Architecture and Synthesis for Embedded Systems, pp. 1–7 (2014)