TAMPERE UNIVERSITY OF TECHNOLOGY

Author(s) Helkala, Janne; Viitanen, Timo; Kultala, Heikki; Jääskeläinen, Pekka; Takala, Jarmo; Zetterman, Tommi; Berg, Heikki

Title Variable Length Instruction Compression on Transport Triggered Architectures

Citation Helkala, Janne; Viitanen, Timo; Kultala, Heikki; Jääskeläinen, Pekka; Takala, Jarmo; Zetterman, Tommi; Berg, Heikki 2014. Variable Length Instruction Compression on Transport Triggered Architectures. 2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS XIV, Samos Island, Greece, July 14-17, 2014 149-155.

Year 2014

DOI http://dx.doi.org/10.1109/SAMOS.2014.6893206

Version Post-print

URN http://URN.fi/URN:NBN:fi:tty-201409161431

Copyright © 2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

All material supplied via TUT DPub is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorized user.

Variable Length Instruction Compression on Transport Triggered Architectures

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

Tommi Zetterman, Heikki Berg
Radio Systems Laboratory
Nokia Research Center
Espoo, Finland
Email: {tommi.zetterman, heikki.berg}@nokia.com

***Abstract*—The SRAM 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 processor 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. The compression approach reaches an average program size reduction of 44% at best. We show that the variable length fetch designs are sufficiently low-power oriented for the system to benefit from the code compression, which reduces the program memory size.**

I. 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. This makes it important to focus on reducing the memory accesses and memory size to reach a better power consumption level on the whole, especially on embedded, battery-powered devices which aim for low power consumption levels.

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 [1]. 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 stored on SRAM 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 [3]. 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) [4] and *Transport Triggered Architecture* (TTA) [5] can gain a lot of power savings from program code compression due to their long 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 processors, 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 memory 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 the code compression is measured by creating a custom processor with 256-bit instructions for the CHStone [6] test suite and compressing each test program's code for the processor using four and eight 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 [7] [8] 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.

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, memory space saving and power consumption. Section VI discusses related work. Section VII concludes the paper.

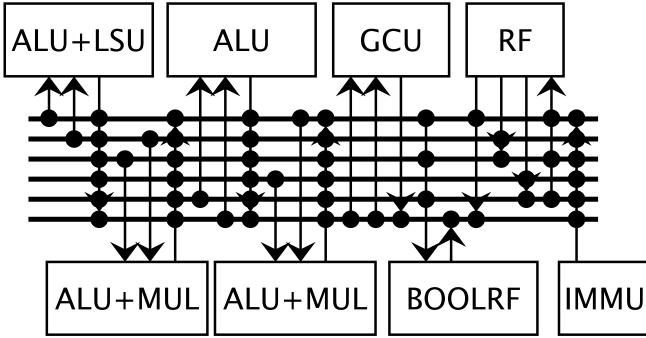

Fig. 1. Example TTA processor with 6 *interconnection buses*. Computational resources include four *arithmetic-logic units* (ALU), two *multipliers* (MUL), a *load-store unit* (LSU), a *general control unit* (GCU), a 64-entry *register file* (RF) and 2-entry *boolean register file* (BOOLRF), and finally an *immediate unit* (IMMU). Some resources are paired to form larger ALU+LSU and ALU+MUL function units.

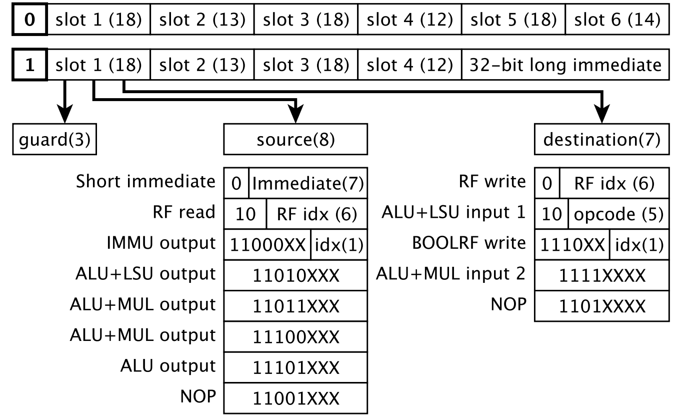

Fig. 2. Instruction encoding for the example TTA processor, using two instruction templates.

II. Transport Triggered Architecture

TTA is a highly modular exposed-datapath relative of VLIW [9]. 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 [10]. Since the many-port RF is a major power sink in a VLIW, this allows significant power savings. Fig. 1 shows an example TTA processor, comprised of several *function units* (FU), two register files and a 6-bus *interconnection network*.

The original work proposed various instruction encodings for TTAs including connection encoding, socket encoding and bus encoding. 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 tool set [11] 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 two moves with a special long immediate move.

III. 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, and the approach is easier to take into use in static multiple-issue architectures than in superscalar architectures. 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.

A. Instruction Template Selection and 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. 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

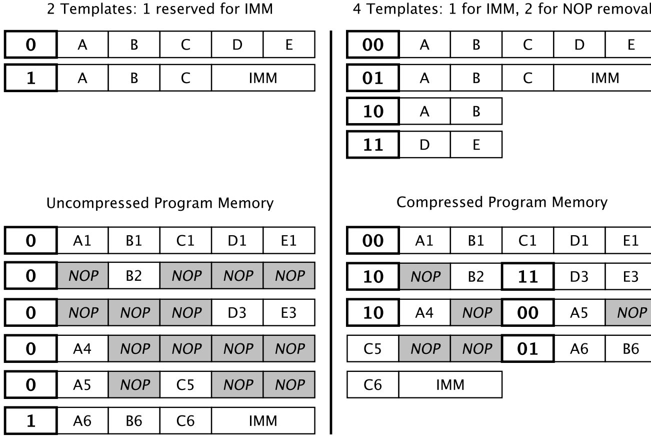

Fig. 3. A short program before (left) and after (right) assigning two new instruction formats, which define two move slots to be used out of the five in the processor. Most of the NOP operations are removed by using the shorter instruction formats in the 2nd, 3rd, and 4th instruction.

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.

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.

B. 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 are no longer 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 architecture is used, consuming more dynamic power. 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 designed 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. It was then decided to remove call instructions from the TTA instruction format altogether to simplify return address calculation, replacing each call with a jump and a register write operation. This moves the return address calculation task to the compiler.

Our solution for random access support on TTA is addressed partially on the compiler and partially inside the fetch unit. Jumps are supported by having all control flow operation targets in the program code to begin aligned 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 due to a jump target. An issue with sudden alignment is that the instruction prior to an aligned instruction may contain redundant information, *padding bits*, which are not to be executed. They are detected by appending 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.

In order to reduce the fetch and decoder complexity, we constraint the smallest variable length instruction size allowed in the processor to a certain *quantum* (q) size. The q and maximum instruction size I_{max} define the size of the multiplexer and shifter networks generated. This q can be increased from instruction template bit field width $IT_w + 1$ to I_{max} for the least logic in the processor, but worst compression efficiency of the instruction template compression. If q equals I_{max}, the instructions become fixed length. The q and I_{max} should be power-of-two values for minimal logic increase.

IV. Hardware Unit Implementation

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

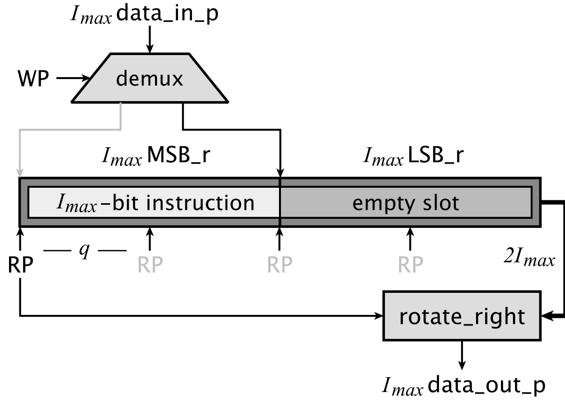

Fig. 4. The structure of the ring buffer fetch unit for $I_{max} = 2q$. An instruction has been written to the MSB register on the previous cycle and is currently being read out of the buffer, indicated by the RP. WP is assigning the next memory word to the LSB register. The RB's contents are rotated 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.

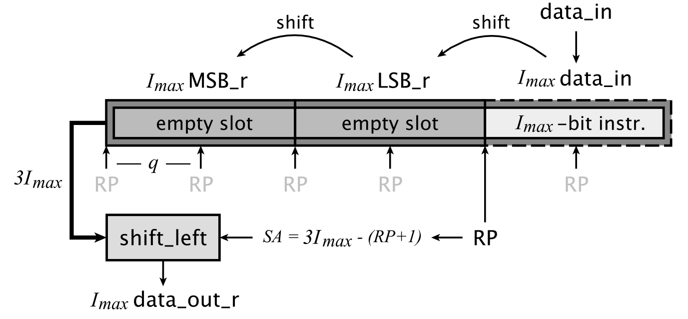

Fig. 5. The shift register fetch structure for $I_{max} = 2q$. An instruction is seen in the data_in port and treated as a part of the buffer. The incoming instruction is I_{max} length and is shifted left to the MSB for output, indicated by the SA value, which is calculated from a virtual RP value. Other possible SA amounts are defined by the q. The data propagates through a data_out register before output.

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 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, it is a requirement 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 WP selection. Due to RB's minimal size, the cycle-accurate internal control logic became 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} 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.

V. EVALUATION

A TTA processor with an I_{max} of 256 bits was created for a subset of the CHStone test suite to measure the compression efficiency of two different instruction template compression configurations: 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 measured 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.

A. Compression Efficiency

The efficiency is reported as space saved by compression:

$$Space\ saving\ (\%) = 1 - \frac{Compressed\ size}{Uncompressed\ size} \times 100 \quad (1)$$

We used the CHStone C-based high-level synthesis test bench for measuring the compression efficiency of the instruction template-based compression. A TTA machine with an I_{max} of 256 and a q of 32 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. The benchmark programs' uncompressed sizes were in the range of 14–50 KB, with the exception of the *jpeg* test which was approximately 376 KB.

In this work, we used greedy workload-based template selection: we first schedule a program on a TTA processor without any NOP templates, then iterate through all possible templates with a given number of NOP slots and select the one which can represent the most instructions in the program. Subsequent templates are selected based on how much they improve the number of covered NOP moves. The narrowest templates are optimized first, since they give a better compression ratio. In our tests with a small 6-bus TTA, this greedy algorithm produced the same templates with a much shorter run-time as an exhaustive search of all template permutations.

For the machine with 2 NOP templates, we selected a 64-bit and a 128-bit template optimized for the *adpcm* benchmark using the greedy workload-based selection process. For the machine with 6 NOP templates, we added 64-bit and 128-bit templates optimized for the *gsm* program, which had the weakest compression ratio, as well as two 32-bit templates optimized for *adpcm*. The 32-bit templates were placed on buses controlling *load-store unit* and *control unit* trigger ports, which are likely to be used in any serial code.

Resulting memory space savings are shown in Fig. 6. The 2-template TTA reached an average program size reduction of 37% and a maximum of 46%, and the 6-template TTA improved to an average of 44% and a maximum of 51%.

B. Program Memory Power Consumption

The power consumption of the program memory was estimated with CACTI before and after instruction template compression. Only ITRS-LSTP was chosen for the SRAM transistor type, because we were interested in the power consumption at 600 MHz clock frequency, which is the upper limit where the LSTP SRAM cells still operate according to CACTI. Technology node used was 40 nm and temperature was a pessimistic 300K for all measurements. The number of bits read out was matched with the memory width, i.e., the full instruction length of 256 bits. One read/write port was used. SRAM size was set exactly to the size of the program, which is unrealistic as SRAM is 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

$$P_{dyn} = \frac{E_{dyn}}{t} = E_{dyn} f_{clk} \quad (2)$$

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. 7. The difference between 2 and 6 instruction templates used for the NOP removal is also visible in the power results: 6 templates covered more of the NOP moves, thus allowing better power saving. 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 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 not manufactured in arbitrary sizes but in power of two sizes, the power saved when switching to a half smaller memory size was estimated with CACTI

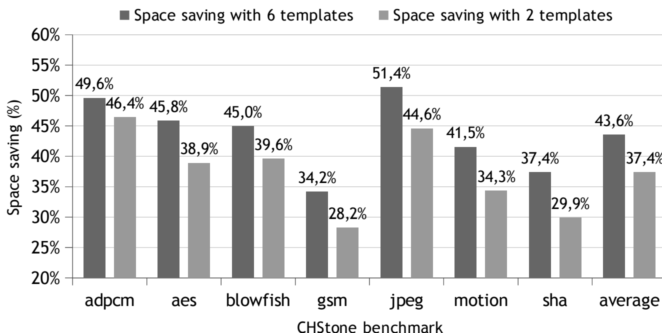

Fig. 6. The amount of memory space saved in the CHStone benchmarks.

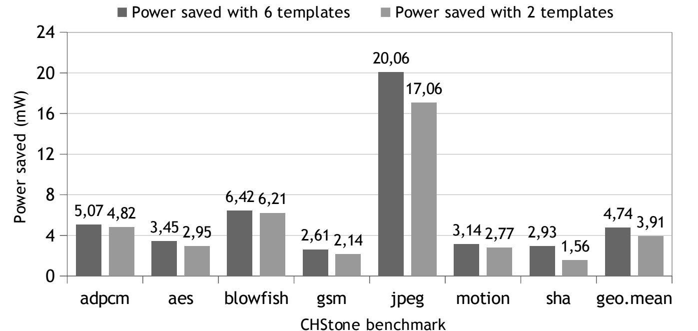

Fig. 7. Power saved with instruction template compression, using 2 and 6 instruction templates.

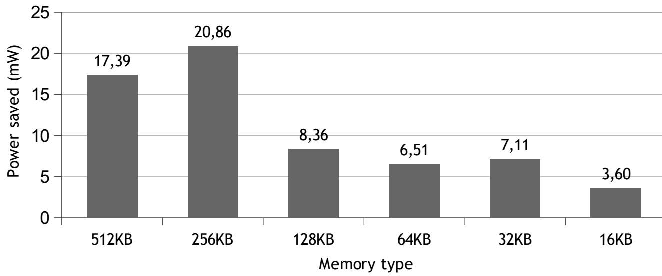

Fig. 8. Memory power saved when LSTP SRAM memory size is reduced by half.

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

C. Fetch Unit Power Consumption

The original and the two alternative fetch designs were synthesized with Synopsys Design Compiler 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 test 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, including the original fixed length fetch unit, 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 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 efficiency 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.

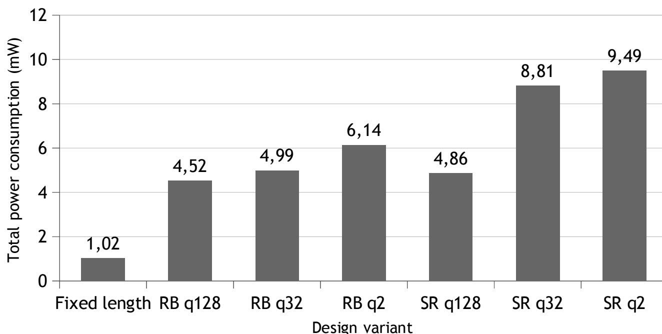

Fig. 9. Fetch units' total power consumption with quanta (q) of 2, 32 and 128 bits, showing worst case test results.

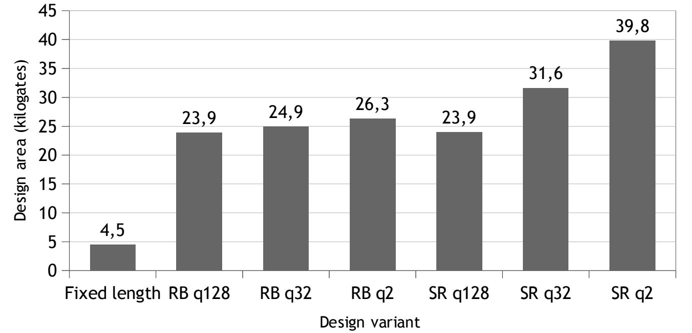

Fig. 10. Fetch units' area in kilogates with quanta (q) of 2, 32, and 128 bits, using target clock speed of 600 MHz.

As long as a SRAM memory power saving of approximately 3.50 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 consumes additional dynamic power to re-assemble the decompressed instructions. This can be projected to be a fairly efficient operation, as it is a multiplexer network which simplifies by choosing a reasonably large q and using few instruction templates.

D. 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. 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.

VI. 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 [12], EPIC [4] 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 [13].

Heikkinen evaluated instruction template-based compression for TTA in [14]. 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 space saving of 53,5% was reached with maximum templates for the processors. The results in this paper are more favorable for several reasons: The quantum in [14] 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 maximum instruction length of 256 bits. The two processor configurations in [14] 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 [4], 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.

VII. Conclusion

This paper proposes a solution for compressing away excess NOP instructions on TTA architectures using a variable length instruction encoding approach and instruction template-based compression. The compression reduces the SRAM memory size required for the program code, lowering the total power consumption of the processor. Two instruction fetch designs are 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.

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. Despite 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.

Future work will include developing a more rigorous template selection method for the case of multiple templates and test workloads. Moreover, we will investigate compiler techniques for taking better advantage of the NOP templates. The instruction scheduler could avoid putting operations to execution slots which can be encoded by the NOP templates. The scheduler could also post-optimize already scheduled code by testing if operations fit easily to nearby templates.

Acknowledgment

The authors would like to thank Finnish Funding Agency for Technology and Innovation (project "Parallel Acceleration", funding decision 40115/13) and Academy of Finland (funding decision 253087).

References

[1] L. Sheng, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of Annual International Symposium on Microarchitecture*, December 2009, pp. 469–480.

[2] B. S. Deepaksubramanyan and A. Nuñez, "Analysis of subthreshold leakage reduction in CMOS digital circuits," in *Proc. Midwest Symp. Circ. Syst.*, Montreal, QC, August 2007, pp. 1400–1404.

[3] H. Pilo, C. A. Adams, I. Arsovski, R. M. Houle, S. M. Lamphier, M. M. Lee, F. M. Pavlik, S. N. Sambatur, A. Seferagic, R. Wu, and M. I. Younus, "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*, February 2013, pp. 322–323.

[4] M. S. Schlansker and B. R. Rau, "EPIC: An architecture for instruction-level parallel processors," Hewlett-Packard, Tech. Rep., February 2000.

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

[6] 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.

[7] CACTI: An integrated cache and memory access time, cycle time, area, leakage and dynamic power model. 2014. [Online]. Available: http://www.hpl.hp.com/research/cacti/

[8] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "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.*, 2008, pp. 51–62.

[9] H. Corporaal, "Transport triggered architectures, design and evaluation," Ph.D. dissertation, Delft Univ. Tech., Netherlands, 1995.

[10] Y. He, D. She, B. Mesman, and H. Corporaal, "MOVE-Pro: A low power and high code density TTA architecture," in *Proc. Int. Conf. Embedded Comput. Syst.: Architectures, Modeling and Simulation*, Samos, Greece, 2011, pp. 294–301.

[11] O. Esko, P. Jääskeläinen, P. Huerta, C. de La Lama, J.Takala, and J. Martinez, "Customized exposed datapath soft-core design flow with compiler support," in *Proc. Int. Conf. Field Programmable Logic and Applications*, Milan, Italy, August 2010, pp. 217–222.

[12] S. Segars, K. Clarke, and L. Goudge, "Embedded control problems, thumb, and the ARM7TDMI," *IEEE Micro*, vol. 15, no. 5, pp. 22–30, Oct 1995.

[13] H. Pan and K. Asanović, "Heads and tails: A variable-length instruction format supporting parallel fetch and decode," in *Proc. Int. Conf. Compilers Arch. Synthesis for Embedded Syst.*, Atlanta, Georgia, 2001, pp. 168–175.

[14] J. Heikkinen, "Program compression in long instruction word application-specific instruction-set processors," Ph.D. dissertation, Tampere Univ. Tech., Finland, 2007.