LoTTA: Energy-Efficient Processor for Always-On Applications

Joonas Multanen, Heikki Kultala, Pekka Jääskeläinen, Timo Viitanen, Aleksi Tervo, Jarmo Takala
Tampere University of Technology, Finland
Email: {firstname.lastname}@tut.fi

***Abstract*—Various use cases in the era of Internet-of-Things (IoT) demand processor devices to have low energy consumption in order to maximize the battery life. In addition to energy constraints, there is often a need to both swiftly execute control-oriented code to provide low reaction times and to occasionally perform real time signal processing tasks efficiently. As a response to these requirements, we propose LoTTA, an extremely energy-efficient exposed datapath core. Its transport-triggered programming model helps in lowering the execution latency via low cost data forwarding. Control efficiency is achieved by an optimized control unit with zero delay slot branches and predicated execution. An instruction register file is included for frequently executed program hot spots to reduce the instruction stream energy consumption. These features allow the processor to execute CHStone and EEMBC CoreMark benchmarks on average with 19% fewer cycles compared to a 6-stage LM32, a traditional RISC core with similar datapath resources. The core consumes 53% less energy on average compared to the RISC core. When including the instruction stream overheads, in the best case, LoTTA saves 79% energy, and on average 40%.**

I. Introduction

Many *Internet-of-Things* (IoT) applications require energy-efficient, high-performance compute devices in order to respond to the limitations and demands of battery-powered appliances. Always-on surveillance cameras, small drones, and sensor nodes should use minimal amount of energy, while being able to instantly perform demanding signal processing tasks when required, and also reacting to external events with a low control code execution latency. In addition to low power and energy operation, this calls for devices to be highly scalable.

Traditionally fixed-function accelerators are used for the most energy-efficient operation. Compared to a typical programmable accelerator, the power, energy, and possible performance gains result from the lack of instruction delivery overhead to the compute elements and datapath tailoring. However, fixed-function hardware development is time consuming, requiring extensive verification and manual effort. In contrast, programmable processors can reuse hardware better to speed up design and verification cycles. They can also be reprogrammed to handle tasks that are not known or yet implemented at hardware design time.

In this paper, we propose an extremely energy-efficient, easy-to-program processor for always-on use cases. The energy-efficiency is achieved with the combination of an exposed datapath and fast execution of code control structures by utilizing zero-delay slot branches and predicated execution. In order to reduce the instruction stream energy overheads, the processor incorporates a compiler-controlled *Instruction Register File* (IRF). The processor is compared to a *LatticeMico32* (LM32) [1] core with benchmarks consisting of both control and signal processing code. Standby energy consumption is an important aspect of IoT device design, but is left outside the scope of this paper, as techniques such as power gating and retention registers can be applied quite modularly to designs.

II. Related Work

Several academic and commercial processors targeting IoT applications, both generic and highly application-specific, have been proposed [2]–[6]. Perhaps the closest recent one in terms of capabilities is *zero-riscy* [2]. It is a 2-stage RISC-V [7] core utilizing a prefetch buffer and compressed instructions.

Sleepwalker [3] is another related processor. It achieves low power consumption with sub-threshold voltage operation and adaptive voltage scaling. The proposed processor differs from the previously mentioned ones in its use of transport triggered programming model, which can provide the benefits of data forwarding without its hardware logic overheads.

While the transport-triggered programming model used in the proposed processor has been studied extensively for improved VLIW processors since the 1990s [8], in fact, the basic concept of transport programming was first published for control processors already in the mid-1970s [9]. The described processor included only one instruction: a data move between memory mapped control registers. Even the ALU was attached to the core like an I/O device. A very similar architecture was commercialized in 2004 by Maxim Integrated. The microcontroller called MAXQ [10] is optimized for branch-heavy code, and the transport programming model simplifies the processor structure.

The proposed processor expands upon the early work on transport programmed control processors. While MAXQ is a 16-bit architecture, the proposed processor can perform arithmetics and memory accesses with 32-bit words. It also supports predicated execution for accelerating programs with very small branches, and adds an IRF for alleviating the instruction stream energy consumption.

III. Transport Triggered Architecture

Transport triggered architecture (TTA) is an "exposed datapath architecture", where the datapath interconnect is programmable. TTAs can exploit instruction-level parallelism in

programs statically by utilizing a long instruction word, similar to *Very Long Instruction Word* (VLIW) processors. A major advantage of TTA is its ability to *software bypass* values between function units and additional programming freedoms. They allow given performance to be reached with a simpler register file with fewer physical ports, and latency-reducing data forwarding can be supported without needing data hazard detection in hardware.

In comparison to traditional multi-stage RISC-processors, TTA simplifies the execution pipeline; as with RISC, TTAs typically include *instruction fetch and decode* stages in their pipeline. However, unlike RISC, TTAs are not limited to performing only register reads in the decode stage. In TTAs, after the decode stage, an instruction can perform any operation typically fixed in a stage of a RISC pipeline: memory read/write, register file read/write or an ALU or custom operation. Thanks to the explicit data transport programming, the function units can be independently pipelined in a simple fashion.

A key aspect of TTAs that we exploited in the processor proposed is that due to their pipeline flexibility, the capability to simplify register files and the omission of data forwarding logic, TTAs can combine the instruction fetch and decode stages into a single stage while still reaching high clock frequencies. This allows the proposed processor to perform both single-cycle jumps and predicated execution in high frequency for efficient execution of control-oriented code, while retaining good signal processing performance with the long instruction word.

IV. Instruction Register File

A clear drawback of exposed datapath architectures that add programming freedoms is the increased code size which results in instruction stream energy overheads. Instruction memory hierarchies commonly consist of large storages farther away from the core, while smaller storage components store temporally and spatially related code near the core. Hardware-controlled caches have been traditionally used to exploit the temporal and spatial locality. An instruction cache checks whether the instruction requested is located in the cache and if it is not, a miss occurs, halting the execution.

Unlike an instruction cache, an instruction register file does not independently decide when to fetch instructions. Instead, the control is given to the program compiler. This allows separating the fetching of instructions from their execution. For fetching, separate IRF load instructions are used. Since these instructions can be placed arbitrarily, blocks of code can be prefetched ahead of their execution, leading to a form of speculative fetching. Compared to a dynamic cache, an IRF is by design simpler due to the absence of tag bits. IRFs are especially well suited for statically scheduled low power architectures. [11]

If execution happens directly from a larger instruction memory, *bypassing* the IRF, its access time is likely to contribute to the critical path of the design. However, if code is executed only from a smaller instruction storage such as IRF, the access time is much smaller and the larger memory storage's access time is removed from the critical path. However, executing all instructions from the IRF is not optimal in every situation. If an instruction is executed only once from the IRF, it increases the overall energy consumption, as the instruction is first read from the next level in instruction memory hierarchy, written into the IRF, and then read from it when executing. In this case, executing directly from the next level in the hierarchy would be more energy-efficient.

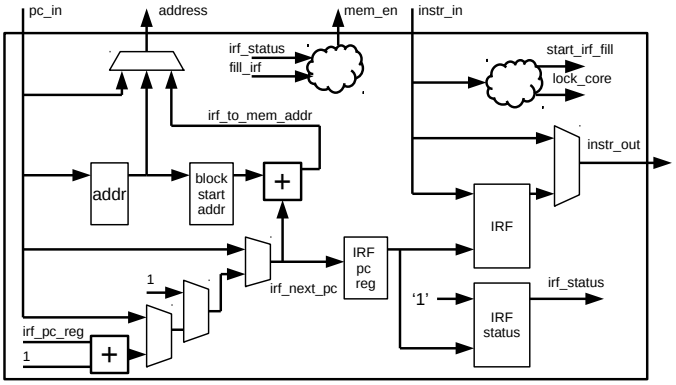


Fig. 1. Instruction fetch implementation with IRF.

A. Implementation

Our proposed IRF is illustrated in Fig. 1. At compile time, the instruction scheduler decides sequences of instructions to be placed in the IRF. However, depending on the control flow, all instructions placed in the IRF might not be executed. In order to eliminate unnecessary fetching, we added a separate *presence bit register*. The presence bits indicate if an IRF entry has already been fetched from memory after the start of the current IRF instruction block, that is, a set of instructions that can be placed in the IRF concurrently. The presence bits are reset at the start of each new IRF block. The IRF is accessed with its own program counter, which is also used to access the presence bits. This differs from the state-of-the-art work [11], where no presence bits are used.

Another motivation for the presence bits are forward jumps. We considered fetching each IRF block completely when IRF execution starts without using the presence bits. However, this would lead to the compiler discarding each candidate IRF block containing forward jumps. In our preliminary evaluation, this reduced the IRF utilization. This could also be handled with prefetching at least a part of the block into the IRF.

In order to avoid writing instructions to the IRF that are only executed once, our implementation supports IRF bypassing. That is, executing code directly from the next level of instruction memory hierarchy. Due to the implemented bypassing, in order to avoid degrading the performance, checking the presence of instructions in our IRF requires the access time of the next memory hierarchy level to be low. Otherwise the access time of the memory can limit the maximum clock frequency of the processor. In previous work [11], all instructions are written to the IRF and executed from it.

Switching to IRF execution is performed with special *header* instructions. Our implementation utilizes the immediate control field of the TTA instruction, as there were unused bit combinations available in that field. In the instruction fetch stage, a single comparator *pre-decodes* the immediate control field and, if a header pattern is found, stalls the core execution for one cycle while the header is read. The header instruction conveys the length of the IRF code block to the instruction fetch. The instruction immediately following the header is the first instruction of the IRF block and is written to the IRF, while simultaneously being passed on to instruction decoding.

Branching inside the IRF is implemented as a separate *irf-jump* instruction, that works similarly to a regular jump, but continues execution inside the IRF. Encountering a regular jump during IRF execution starts execution from the next level in memory hierarchy. When program execution reaches the last index of the IRF and does not branch, execution falls through to the instruction following the IRF code block in the next level instruction storage.

B. Compiler Support

A crucial factor in the IRF efficiency is deciding *which* instructions to store in the IRF. Another factor is deciding *when* to fill the IRF, as the software control allows fetching arbitrary instructions long before they are accessed. Program control analysis has been previously studied extensively [12]. As loops and nested loops are typically program hot spots, we first focused the compiler support for them. The algorithm is executed after instruction scheduling in the compiler.

The IRF block allocation algorithm is illustrated in Fig. 2. First, all program basic blocks are split into lengths equal to the IRF size. Then, each of the split blocks are assigned to an individual IRF block and merged with following constraints:

1) All incoming jumps to the IRF block must target the first instruction, and
2) function calls cannot exist inside IRF blocks.

In other words, a function call always splits an IRF block. The allocation algorithm first assigns inner loops as IRF blocks and if the capacity allows, outer loops are also included. An instruction block bypasses the IRF if there are no backward jumps in it. These are executed directly from the next level in memory hierarchy. As a final step, branch targets are fixed taking into account the inserted header instructions.

```
1: for all basicblocks in CFG do
2:     if basicblock.size > irfsize then
3:         split BB to irfsize
4:     end if
5:     create a new irf block for bb
6:     queue created irfblock
7: end for
8: for all irfblock in queue do
9:     nextblock ← irfblock.successor
10:    if irfblock does not end in a call and nextblock has no incoming jumps from outside these
       two blocks and irfblock.size + nextblock.size < irfsize then
11:        merge current irf block with next
12:        requeue(irfblock.predecessor)
13:    else
14:        remove irfblock from queue
15:    end if
16: end for
17: for all irfblock in irfblocks do
18:    if irfblock has no backwards jump inside the block then
19:        irfblock.setbypassblock
20:    else
21:        create irf block header instruction for fetch and execute with IRF
22:    end if
23: end for
24: for all irfblock in irfblocks do
25:    if not irfblock.isbypassblock then
26:        for all jump in irfblock do
27:            if jump destination is inside same irf block then
28:                convert jump to local irfjump
29:            end if
30:        end for
31:    end if
32: end for
```

Fig. 2. The IRF Block allocation routine.

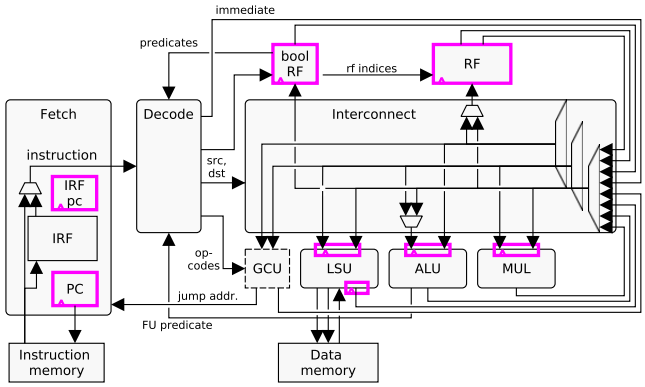


Fig. 3. Overview of LoTTA.

V. Evaluation

The proposed ultra-low power processor core, *Low-power Transport Triggered Architecture* (LoTTA), was designed using the *TTA-based Co-Design Environment* (TCE) [13] tools. A block-level representation of the designed core is presented in Fig. 3. To compare the TTA programming model with a traditional multistage RISC architecture, a LatticeMico32 core was used as a reference point. The proposed core intends to combine the qualities from both of these areas. For fair comparison, operations available on hardware, their latencies, and the amount and size of register files were matched. Both cores had similar arithmetic and logic operations, a hardware multiplier, and a barrel shifter. The register file contained 32 x 32-bit registers and had two read ports and one write port. Although implementation of the cores was different, a rough similarity of the two was ensured via ASIC synthesis, where the area occupied by each component was roughly the same between the two cores. Adding the IRF support in this specific design increased the instruction size from 49 bits to 50 bits.

LoTTA was evaluated with benchmarks from two different use cases typical to always-on microcontrollers. All benchmarks were compiled with *tcecc*, the program compiler of TCE. *Coremark* [14] was used to evaluate the performance in control-oriented code. Competence in the other area of interest, *Digital Signal Processing* (DSP), was evaluated with eight fixed-point benchmarks from *CHStone* [15].

To verify the correct functionality of the C language benchmark programs, they were compiled for the processor and simulated using *ttasim*, TCE's instruction cycle-accurate simulator. Hardware level correctness was ensured by generating memory images from the compiled programs and then

TABLE I
AREA AND TIMING RESULTS FOR BASELINE TTA (NO IRF) AND LM32.

architecture	TTA	TTA	LM32	LM32
target	100 MHz	max.	100 MHz	max.
max. clock frequency (MHz)	311	1333	351	1667
area (μm^2)	7426	7542	8904	11456

simulating them at *Register Transfer Level* (RTL) with Mentor Graphics ModelSim 10.5.

All the cores evaluated were synthesized with Synopsys Design Compiler I-2013.12. The process technology was 28 nm FD-SOI with 0.95 V voltage, with typical process corner and 25°C temperature. *Switching Activity Interchange Format* files were produced with ModelSim 10.5 and used to estimate power for the synthesized designs.

The baseline TTA core was synthesized with two timing constraints: a relaxed constraint to obtain a low-power design point and a tight constraint for a high-performance design point. Since the 28 nm technology library includes body-biased variations of the standard cells, a 10 ns timing constraint was chosen in order to mostly utilize the less energy consuming but slower standard cells. Area and timing results after synthesis are presented in Table I. With the 10 ns constraint, the TTA core reached a maximum clock frequency of 311 MHz after synthesis. With the same configuration, LM32 reached a maximum clock frequency of 351 MHz, with the critical path in the multiplier unit. LoTTA including an IRF with 256 entries reached a maximum clock frequency of 221 MHz. To compare energy consumptions and minimize the amount of leakage power, an operating clock frequency of 200 MHz was chosen, as all the IRF variants of the LoTTA could reach this.

To evaluate the potential for maximizing serial performance, the LoTTA and LM32 were synthesized with target clock frequencies at intervals of 0.05 ns in order to find the maximum clock frequency allowed by the ASIC technology. LoTTA reached a maximum clock frequency of 1333 MHz and LM32 reached 1667 MHz. The ALU output port in LoTTA can be used for predicated execution and because of this, the critical timing path ended up between the ALU and the instruction fetch unit. LM32 utilizes a six-stage pipeline without predication support, allowing a short critical path and higher clock frequency in this case.

Compiled code sizes are listed in Table II. On average, LoTTA program size is 1.84x larger than the LM32 code. *Coremark* has the largest difference, where LoTTA instruction bit amount is 3.5x larger. When long instruction word processors cannot fill all of their instruction slots for each cycle, *No-Operations* (NOPs) are inserted into the code, bloating the code size. However, in *aes*, *blowfish*, *mips* and *motion* , TTA code size is smaller than that of LM32. This is due to loop unrolling and function inlining. These allow the wide TTA instructions to be efficiently utilized. LM32 requires 32 bits for each instruction, whereas in TTA custom instruction set depending on the operation, 1-3 instructions can fit into the 49 instruction bits.

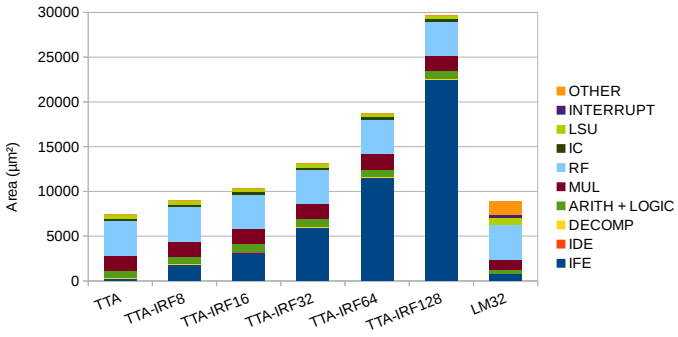


Fig. 4. Area comparison of the evaluated cores.

Area comparison for the evaluated cores after synthesis is presented in Fig. 4. LoTTA without IRF is roughly the same size as the LM32. The increment in area is quite linear in relation to the number of IRF entries. The entries are implemented as flip-flops in the RTL description. In the baseline LoTTA, the RF occupies more than half of the silicon area. This is undesirable, as RF is only a storage for data and not an actual compute element. Ideally, the amount of logic contributing to computation should be maximized. The TTA programming model typically allows the core to have smaller register files and fewer register file ports compared to VLIWs, as the datapath is controlled explicitly and (intermediate) results might not need to be written to the RF [8], but can be temporarily stored in output port registers in function units. To evaluate this aspect, the nine benchmarks used here were also executed on an alternative design where the RF size was reduced to 16 entries, showing only a minimal performance hit. However, the RF size was set to 32 entries to match the LM32 resources.

Benchmark execution cycle counts for LoTTA core are presented in Table II. Except for *jpeg* and *motion* the TTA core executes the benchmarks in fewer clock cycles. A large fraction of execution time in *jpeg* consists of manually written *memcpy* operations, which do not optimize well currently in the TCE compiler, unlike normal memcpy operations. The best speedup (2.9x) was obtained in *aes*. Here, the TTA compiler exploited very efficiently the instruction-level parallelism in the benchmark. All the benchmarks benefit from the fast

TABLE II
PERFORMANCE AND CODE SIZE RESULTS. VALUES FOR TTA WITHOUT IRF.

benchmark	TTA cycle count	LM32 cycle count	TTA code size (B)	LM32 code size (B)
adpcm	81683	88355	16445	9396
aes	25690	75184	16372	17052
blowfish	673151	798794	6100	6820
gsm	12745	26798	11851	6000
jpeg	8097755	3018543	50500	17520
mips	23649	27348	2597	3328
motion	7797	7757	6161	8620
sha	543428	677708	4269	4148
coremark	403416	582818	42979	12400

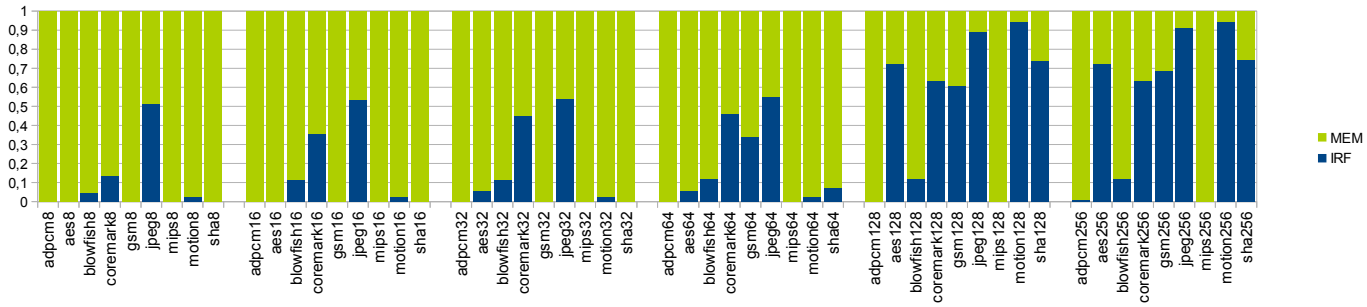


Fig. 5. Distribution of executed instructions between memory and IRF.

branching, which is helpful when executing control-oriented code. This is not only due to the simple instruction fetch and decode logic, but also the guarded execution directly from ALU output port. In the control-oriented *Coremark*, the guarded ALU execution improved the cycle count by 10%.

The IRF control header instructions affect the execution times of programs as each header execution incurs a stall of one clock cycle. However, the cycle count increase in all benchmarks was less than 0.8% and thus had no significant impact on overall performance.

The proportion of instructions executed from IRF and memory per benchmark are presented in Fig. 5. Due to the nature of the benchmark programs, utilization of the IRF is quite low, when IRF size is less than 128. In *mips*, the IRF is not used at any size, due to a large while loop containing all the code in the benchmark. The compiler cannot split this loop to fit into the IRF. In *adpcm*, there is an encoding and a decoding function, that are executed within a for loop. Similar to *mips*, this code structure does not fit into the IRFs evaluated. An optimization that would benefit the utilization and reduce energy consumption, would be to allow execution of code sequences such as loops from both the IRF and the next level of memory hierarchy consequently. Currently, if there is a loop that does not fit fully into the IRF, the compiler does not utilize the IRF at all. Executing a loop even partly from IRF would reduce the energy consumption.

With the smallest IRF configuration, eight entries, only *jpeg* utilizes the IRF efficiently. This is due to the heavily executed loop in the code containing exactly eight instructions. IRF is mostly used in the benchmark set, when it has 128 entries. At this point, in *jpeg* and *motion* nearly all code structures fit into the IRF. Doubling the IRF size to 512 entries does not notably increase the IRF utilization.

To evaluate the effect of different IRF sizes on the instruction stream overall, we evaluated a processor, where the instruction stream consists of an on-chip SRAM instruction memory and and IRF in the core. For the instruction memory, we estimated access energy numbers with Cacti-P [16]. Four SRAM sizes were chosen to represent different design points, using ITRS-LSTP SRAMs.

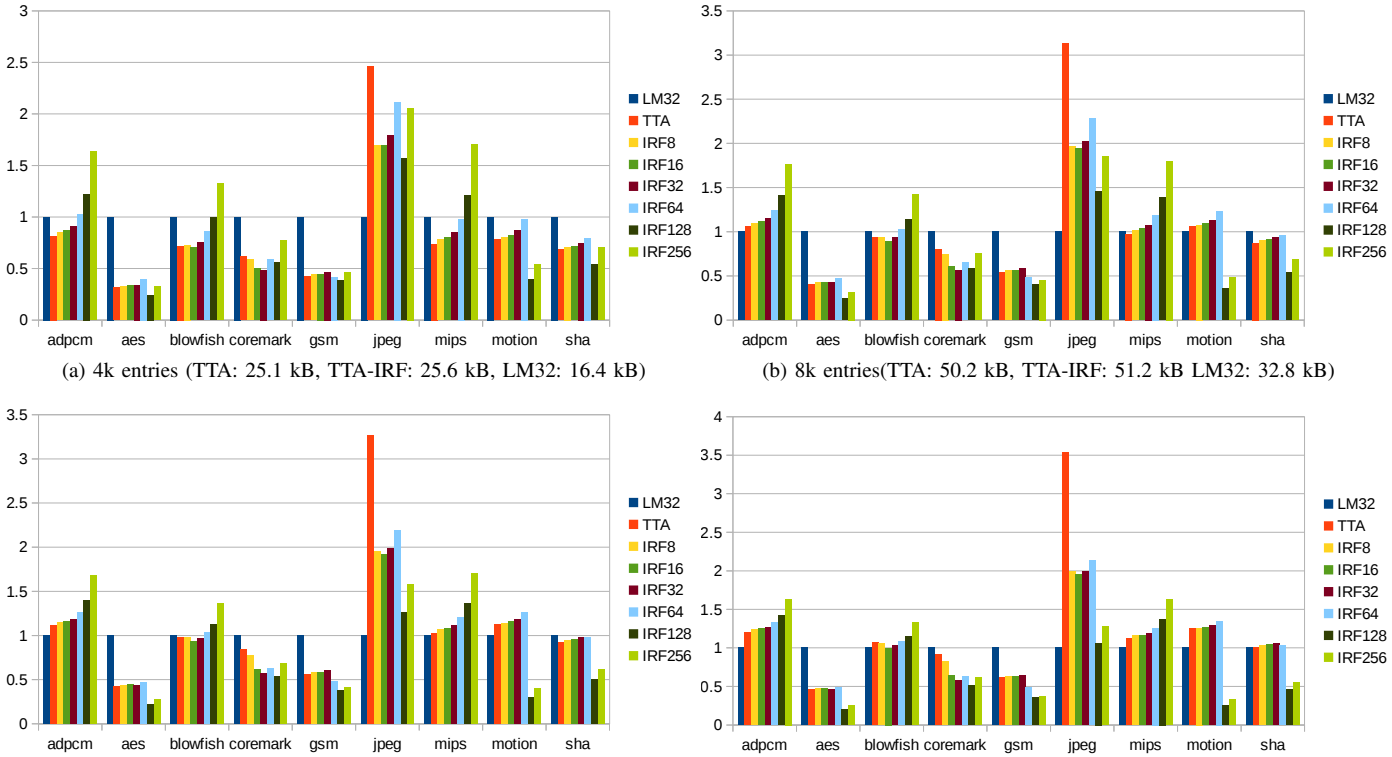
As previous work [11] has extensively studied the energy benefits of the instruction register files compared to small filter caches, we do not make comparisons to caches in this work. In Fig. 6, the four design points are compared. Based on the IRF utilization, we calculate the energy consumption normalized to that of LM32. For a fair comparison, as LoTTA has a 49-bit instruction word (50 bits with IRF) and LM32 has only 32-bit instructions, the cores are evaluated with SRAMs equalized not according to the byte amount, but to the number of entries. This way we take into account the wider instruction width of LoTTA. As the ITRS-LSTP SRAMs have a low standby power, their energy consumption consists mainly of the number of accesses to them. Thus, naturally the energy benefits of the IRF increase, when the SRAM size increases. However, very large SRAMs are not realistic since they would be accessed through a smaller low-level cache to reduce access latency and energy.

Taking into closer inspection the design point with a 32 k-entry SRAM, the 128-entry IRF saves the most energy with an average of 40% across all benchmarks. The most energy is saved in *aes*, 79% and the average saving is 53%. This configuration uses the most energy in *adpcm*, where the consumption is 1.45x more than that of LM32. Comparing this IRF + SRAM configuration to the TTA core without an IRF, the IRF saves a geometric average of 43% of energy across all benchmarks.

VI. Conclusions

In this paper, we compared a transport triggered architecture based processor to an open-source LatticeMico32 core when targeting mixed control and data processing. With matching hardware resources, the TTA core used 19% fewer clock cycles on average (geometric). The benchmarks consisted of control-oriented code and code modeling signal processing tasks, in order to evaluate suitability for IoT applications.

A drawback of TTAs is their larger program size. With the evaluated benchmarks, the programs were 1.84x larger compared to LM32 on average. To mitigate the increased code size and improve the otherwise good energy-efficiency of the TTA approach, an instruction register file with bypass support was designed and integrated to the core. The TTA programming model alone saved on average 53% of energy (geometric), when comparing the two cores without memories. With the help of IRF, a 128-entry IRF in a processor with a 32 k-entry on-chip memory saved 40% of the combined core and instruction memory system energy. For future work, we



(a) 4k entries (TTA: 25.1 kB, TTA-IRF: 25.6 kB, LM32: 16.4 kB)

(b) 8k entries(TTA: 50.2 kB, TTA-IRF: 51.2 kB LM32: 32.8 kB)

(c) 16k entries(TTA: 100.4 kB, TTA-IRF: 102.4 kB, LM32: 65.5 kB)

(d) 32k entries(TTA: 200.7 kB, TTA-IRF: 204.8 kB, LM32: 131.1 kB)

Fig. 6. Energy comparison of the cores with four instruction memory sizes.

plan to transfer control of the IRF more to the compiler, rather than hardware and create a design optimized for high clock frequency burst mode execution.

ACKNOWLEDGMENT

The authors thank the following financial support: Tampere University of Technology Graduate School, Business Finland (FiDiPro Program funding decision 40142/14), HSA Foundation, and the Academy of Finland (funding decision 297548).

REFERENCES

[1] Lattice Semiconductor. (2018, May) Latticemico32. http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/ IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx.

[2] P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," in *Proceedings of International Symposium on Power and Timing Modeling, Optimization and Simulation*, Sept 25-27 2017, pp. 1–8.

[3] D. Bol, J. De Vos, C. Hocquet, F. Botman, F. Durvaux, S. Boyd, D. Flandre, and J. Legat, "SleepWalker: A 25-MHz 0.4-V Sub-mm^2 7-μm^2 μW/MHz microcontroller in 65-nm LP/GP CMOS for low-carbon wireless sensor nodes," *Journal of Solid-State Circuits*, vol. 48, no. 1, pp. 20–32, Jan. 2013.

[4] Y. Zhang, L. Xu, K. Yang, Q. Dong, S. Jeloka, D. Blaauw, and D. Sylvester, "Recryptor: A reconfigurable in-memory cryptographic cortex-m0 processor for iot," in *Proceedings of Symposium on VLSI Circuits*, June 18-22 2017, pp. C264–C265.

[5] S. Senni, L. Torres, G. Sassatelli, and A. Gamatie, "Non-volatile processor based on MRAM for ultra-low-power iot devices," *Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 2, pp. 17:1–17:23, Dec. 2016.

[6] Z. Wang, Y. Liu, Y. Sun, Y. Li, D. Zhang, and H. Yang, "An energy-efficient heterogeneous dual-core processor for internet of things," in *Proceedings of the International Symposium on Circuits and Systems*, May 24-27 2015, pp. 2301–2304.

[7] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[8] J. Hoogerbrugge and H. Corporaal, "Register file port requirements of transport triggered architectures," in *Proceedings of the International Symposium on Microarchitecture*, Nov. 30 - Dec. 2 1994, pp. 191–195.

[9] G. Lipovski, "The architecture of a simple, effective control processor," in *Proceedings of Euromicro Symposium on Microprocessing and Microprogramming*, October 1976, pp. 7–19.

[10] Maxim Corporation, "Introduction to the MAXQ architecture," 2004. [Online]. Available: https://www.maximintegrated.com/en/app-notes/index.mvp/id/3222

[11] D. Black-Schaffer, J. Balfour, W. Dally, V. Parikh, and J. Park, "Hierarchical instruction register organization," *Computer Architecture Letters*, vol. 7, no. 2, pp. 41–44, July 2008.

[12] J. Park, J. Balfour, and W. J. Dally, "Fine-grain dynamic instruction placement for l0 scratch-pad memory," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Oct. 24-29 2010, pp. 137–146.

[13] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, *HW/SW Co-design Toolset for Customization of Exposed Datapath Processors*. Springer International Publishing, 2017, pp. 147–164.

[14] EEMBC – The Embedded Microprocessor Benchmark Consortium. (2018, May) Coremark benchmark. Http://www.eembc.org/coremark.

[15] 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," *Journal of Information Processing*, vol. 17, pp. 242–254, Oct. 2009.

[16] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Proceedings of the International Conference on Computer-Aided Design*, Nov. 6-10 2011, pp. 694–701.