Journal of Signal Processing Systems
https://doi.org/10.1007/s11265-020-01578-3

Energy Efficient Low Latency Multi-issue Cores for Intelligent Always-On IoT Applications

Joonas Multanen[1] · **Heikki Kultala**[1] · **Kati Tervo**[1] · **Pekka Jääskeläinen**[1]

Received: 2 May 2019 / Revised: 29 June 2020 / Accepted: 6 July 2020

Abstract

Advanced Internet-of-Things applications require control-oriented codes to be executed with low latency for fast responsivity while their advanced signal processing and decision making tasks require computational capabilities. For this context, we propose three multi-issue core designs featuring an exposed datapath architecture with high performance, while retaining energy-efficiency. These features are achieved with exploitation of instruction-level parallelism, fast branching and the use of an instruction register file. With benchmarks in control-flow and signal processing application domains we measured in the best case 64% reduced energy consumption compared to a state-of-the-art RISC core, while consuming less silicon area. A high-performance design point reaches nearly 2.6 GHz operating frequency in the best case, over 2× improvement, while simultaneously achieving a 14% improvement in system energy-delay product.

Keywords Low power · Instruction stream · Energy-efficiency · Instruction register file · IoT · Always-on · RISC-V · TTA · Exposed datapath · Transport Triggered Architecture

1 Introduction

It is estimated, that the *information and communication technology* (ICT) sector will consume up to 20% of global energy production by 2025 [1]. From an environmental point of view, there are estimates that around 14% of total greenhouse gas emissions emerge from the ICT sector by 2040 [2]. The era of *Internet-of-Things* (IoT) and its increasing demands on computational complexity are expected to result in the introduction of billions of compute devices. Many of these small form factor devices are battery-powered or use energy harvesting for their power supply, requiring energy-efficient and low power operation.

While maintaining low energy consumption, devices such as always-on surveillance cameras, small drones, and sensor nodes, are required to react to events and perform demanding signal processing and artificial intelligence tasks, and also to handle external events with low control code execution latency. Besides their low power and energy consumption requirements, this calls for the devices to be highly performance scalable.

For maximal energy-efficiency, fixed function accelerators are typically used. Compared to programmable devices, their hardware is optimized at design-time to match predefined requirements. This allows removing instruction delivery overheads and tailoring the datapath, resulting in high computational capability and energy efficiency in small chip area. The clear drawback is that the accelerators perform poorly or not at all with tasks not defined at design time. Moreover, their design, optimization and verification is a costly, time consuming process requiring manual effort [3, 4]. In contrast, software programmable devices offer flexibility [5] in terms of non-predefined tasks and reduce the design cost and time with reusable compute and logic elements, but incur overheads due to the flexible software based control.

✉ Joonas Multanen
joonas.multanen@tuni.fi

Heikki Kultala
heikki.kultala@tuni.fi

Kati Tervo
kati.tervo@tuni.fi

Pekka Jääskeläinen
pekka.jaaskelainen@tuni.fi

[1] Faculty of Information Technology and Communication Sciences, Tampere University, Tampere, Finland

Published online: 26 July 2020

In this article, we propose energy-efficient, programmable processor cores for always-on applications. The cores feature fast branching and efficient exploitation of exposed datapath *instruction-level parallelism* (ILP) to achieve high performance. In order to reduce the energy overhead of instruction delivery, a compiler-controlled *instruction register file* (IRF) is used. The cores are compared to a publicly available *LatticeMico32* (LM32) [6] core offered by Lattice Semiconductor and *zero-riscy* [7], a RISC-V [8] ISA implementation using benchmarks representing both control oriented and signal processing tasks. All the cores are evaluated on a 28 nm ASIC standard cell technology.

For this article we extended our previous conference paper [9] with the following additions:

- The previously proposed LoTTA design was adapted to two additional design targets: high energy-efficiency and high clock frequency.
- More accurate results obtained by place & route as opposed to original synthesis results.
- A comparison to a state-of-the-art low power RISC-V based core, *zero-riscy* was added.
- Extended textual presentation with detail added specifically to parts that were pointed out by the conference paper reviews.

The article is organized as follows. Section 2 overviews related work. Section 3 describes the three core variants along with an explanation of their underlying architecture and programming model. Section 4 details the concepts of instruction register files and the two implementation variations used in this work. Section 5 presents the evaluation results, and Section 6 concludes the article.

2 Related Work

The emerging era of IoT has resulted in plenty of processor proposals suitable for always-on energy constrained scenarios. The approaches vary from generic to highly domain-specific. *SleepWalker* [10] and the IoT SoC [11] proposed by Klinefelter et al. are targeted for scenarios, where energy delivery is difficult and solutions such as energy-harvesting are required to power the device, such as wireless sensor networks. The latter is targeted for biomedical applications and includes domain-specific accelerators to maximize energy-efficiency with the trade-off of computational flexibility. *Recryptor* [12] is an ARM Cortex-M0 based IoT-targeted cryptoprocessor, where energy-efficiency stems from near-threshold voltage operation and in-memory computing. Senni et al. [13] leverage the non-volatility and low leakage power consumption of *magnetic random access memory* (MRAM) to reduce processor energy consumption when idling. Wang et al. [14] propose a dual-core processor system, where one core has high performance and the second core has relatively low performance, but is 3× more energy-efficient. The authors utilize their two-core system with a proposed energy-aware task scheduler. Roy et al. [15] utilize sub-threshold voltage operation in their implementation of MSP430 targeted for IoT domain and especially for biomedical applications.

The recently proposed *zero-riscy* [7] implements the popular RISC-V *instruction set architecture* (ISA). It includes lightweight instruction prefetching and compressed instructions, and thus seems to be closest to our work in terms of processing capabilities and target context. Since its implementation is available as open source, it was picked for closer comparison to this article.

Another openly available alternative is the *LatticeMico32* [6]. It is a 32-bit microprocessor core with a Harvard architecture and a RISC ISA, with optional configurable multiplier and shifter units.

Sleepwalker [10] is also close to our work. It uses adaptive voltage scaling along with sub-threshold voltage operation to achieve energy-efficiency. Both of these techniques could be applied on top of our proposed cores to improve the energy efficiency even further. In fact, there is already previous work of a chip implementation of an application-specific sub-threshold voltage *transport-triggered architecture* (TTA) core developed using the same tools and the processor template as the cores proposed in this paper [16].

The most apparent feature differentiating the proposed processor is its *transport-triggered* programming model, where data forwarding is controlled by software, eliminating the forwarding logic hardware overheads and supporting instruction-level parallelism with simpler register files. Maxim Integrated commercialized a TTA-based microcontroller in 2004. The microprosessor called MAXQ [17] uses the transport programming model to simplify the processor structure and is optimized for control oriented codes characterized by heavy branching. From their white papers we observe that the key difference of MAXQ to the cores presented in this article are is its 16b scalar data path, while the proposed cores include a 32b data path and integrate additional features to reduce the instruction stream energy footprint and to support higher maximum clock frequencies combined with instruction-level parallelism.

3 Proposed Cores

In previous work [9], we designed the *Low-power Transport Triggered Architecture* (LoTTA) by using the *TTA-based Co-Design Environment* (TCE) [18] processor design tools. Figure 1 shows LoTTA in the TCE processor designer with

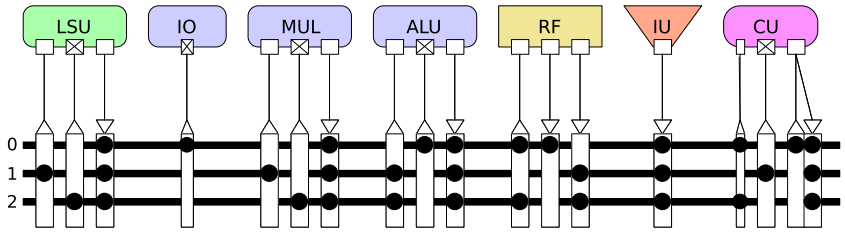

Figure 1 TCE view of the function unit and interconnection architecture common to the proposed cores. LSU—load-store unit, IO—standard input-output, MUL—multiplier, ALU—arithmetic logic unit, RF—register file, IU—immediate unit, CU—control unit.

an overview of the databus interconnection network and individual function units. The same interconnection network and function unit organization was used as a basis for all three of the proposed cores. As increasing the number of connections in the interconnection network increases the logic required, often ends up in the critical path, and widens the instruction width of the bus transport programmed architecture, it is important to prune the interconnection network carefully to a minimal level, that can still maintain good performance.

The core intends to combine qualities needed for fast execution of control code along with signal processing, and is designed with the mindset to support various workloads. Therefore, special purpose function units were not utilized in addition to generic load-store units, register files, ALUs and multipliers. This mindset was continued in the other two proposed cores. Operations implemented in the cores are listed in Table 1. The cores utilize software based data forwarding enabled by a transport-triggered programming model which result in a very simple and energy efficient control unit and reduced register file microarchitecture complexity.

This paper extends our previous work by proposing two design points modified from the original processor core. For the purposes of this paper we set the instruction bit width to the exact number derived from the core features, even if it is not a power of two or even byte-aligned. Instead of extending the instruction width to a byte alignment or a power of two, we assume that in an application-specific processor, the memory architecture can be customized for maximal energy-efficiency. If needed, the instruction width can be aligned depending on the use case.

Next we describe the underlying architecture and programming model in the proposed cores, followed by details of the core variants.

Table 1 Operations implemented in the proposed cores.

Function unit	Operations
LSU	{8, 16 & 32} bit load/store
IO	write to stdout
MUL	multiply
ALU	abs, add, and, eq, gt, gtu, ior, max, maxu, min, minu, ne, neg, shl, shr, shru, sub, sxhw, sxqw, xor
RF	read/write register
IM	load/store 32 bit immediate
CU	jump, call, {bz, bnz}[a]

[a] *LoTTA* core uses predicated execution instead of conditional branch operations

3.1 Transport Triggered Architecture

Transport triggered architecture (TTA) belongs to the family of "exposed datapath architectures", where the datapath interconnection network is controlled by the programmer. TTAs feature a long instruction word to describe instruction-level parallelism in applications. Compared to similar *very long instruction word* (VLIW) processors, having an exposed datapath allows TTAs to transport data values between function units via *software bypassing*. This has the advantage of not requiring hardware to detect data hazards when forwarding. Optionally registered input and output ports of function units reduce unnecessary register file traffic, allowing TTAs to reach equal performance with a simpler register file, with fewer physical ports when compared to a VLIW [19].

An understudied advantage of TTA is its simplified execution pipeline. Similar to traditional RISC processors with multiple pipeline stages, the first stages in TTA pipeline are *instruction fetch and decode*. However, whereas different types of operations are fixed to a distinct stage in a RISC pipeline, TTAs can perform any type of operation after the decode stage. These include memory operations, register file transfers, or any kind of ALU or custom operations. This is illustrated in Fig. 2, where the three different TTA execution pipelines of the proposed cores are compared against a classic RISC pipeline and two of its variants implemented in the reference cores.

While TTAs have been targeted and studied extensively as overcoming some of the shortcomings of VLIW processors since the 1990s, they were first originally introduced

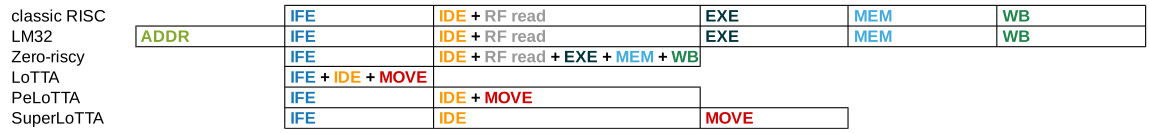

Figure 2 Execution pipelines stages of the evaluated cores compared with a classic RISC pipeline organization.

as control processors in the 1970s [20]. Here, only a single data *move* instruction was used to transport data between memory mapped control registers.

The pipeline flexibility, simplified register files and elimination of data forwarding hardware allow TTAs to merge the instruction fetch and decode stages, while retaining a high clock frequency. This results in fast branching and, therefore, efficient execution of control code. This is done while still providing ILP required by more complex algorithms.

The next sections describe the details of the proposed cores. Execution pipelines of the cores are compared in Fig. 2. Comparison of the differences in the architecture of the proposed cores is listed in Table 2.

3.2 LoTTA

In the *LoTTA* core, the primary design goal was to minimize the use of hardware resources and to minimize branching delays for fast execution of control-oriented code [9], while maintaining good signal processing performance. For fast branching, we merged the instruction fetch, decode and *move* stages into the same pipeline stage. This is feasible thanks to the straightforward, yet efficient fetch and decode logic of the TTA programming model. *LoTTA* utilizes predicated execution for streamlining conditional codes. However, together with the merged fetch and decode, this results in a long critical path delay [9] as the instruction SRAM implemented in the processor system ends up in the critical path.

3.3 PeLoTTA

PeLoTTA is a clock frequency optimized high-*per*formance version of *LoTTA*. Here, the target was to optimize the maximum clock frequency, while still supporting the fast branches and energy-efficient operation of the original *LoTTA*. To achieve this, the instruction pipeline was split from one to two stages; the first, instruction fetch stage, retrieves an instruction from the memory, with decode and the move/execute in the second.

In order to further optimize the clock frequency, the predicated execution of *LoTTA*, which ended up in the critical path [21], was replaced with *conditional branch* operations *branch equal to zero* (bz) and *branch not equal to zero* (bnz). This also reduced the instruction width, as the boolean RF used to hold predicate values was removed, saving bits in the instruction word that were used to control it. Likewise, the instruction decode unit was simplified, as the predicate evaluation was removed.

3.4 SuperLoTTA

As a design point targeting maximum operating clock frequency even with the trade-off in additional operation execution latencies that need to be dealt with in the software side, we designed the *SuperLoTTA* core. In order to achieve this design target, we added extra pipeline registers to the ALU, load-store unit and multiplier unit, isolating them from the interconnection network. The same was done to the control unit, which allowed to isolate instruction memory from the critical path.

As with *PeLoTTA*, to further optimize the clock frequency we added conditional branch operations. This allowed us to simplify the decoder logic compared to full predicated execution capabilities.

4 Instruction Register File

While "exposed datapath processors" have benefits in terms of simplified hardware, they incur a trade-off in increased compiler complexity due to the additional programmer responsibilities. Moreover, increased programmer control

Table 2 Feature comparison of the proposed cores.

	Instr. width	Branch delay slots	Predicated execution	Cond. branch operations	ALU delay	Optional IRF style	Design target
midrule LoTTA	49	0	✓		1	*basic*	Branch delay
PeLoTTA	45	1		✓	1	*improved*	Energy-efficiency
SuperLoTTA	43	3		✓	2		Clock frequency

translates to a wider instruction word and increased code size in general, which conflicts with the need to optimize the instruction stream's energy consumption. [22]

Instruction memory hierarchies typically employ small storages close to the core to store temporally and spatially related code, while larger but slower storages are kept farther away. Traditionally, caches controlled by hardware have been used as lower level instruction storages. Their operation requires keeping track of values present in the cache with separate tag bits, which need to be checked upon a cache read. If the value requested is not present in the cache, execution stalls while the cache miss is resolved from higher levels in memory hierarchy.

Caches can be integrated into a system with relatively small effort, due to being hardware-controlled, but with the cost of control logic overhead. This overhead can be removed, if control is moved to software, such as in an instruction register file. An IRF is explicitly instructed to fetch new instructions, with the additional benefit of possibility to separate the fetching of instructions from their execution. While this allows a type of speculative fetching, the compiler is required to efficiently decide the placement of special IRF load instructions for good performance. With their fine-grained control qualities, IRFs have been shown to be suitable for low power architectures. [23]

IRFs require certain design choices for their implementation. *Bypassing* allows instructions to be read directly from the next level in memory hierarchy without first storing them into the IRF. However, the access time of the larger instruction memory unit is likely to add to the design's critical path. With a smaller unit such as IRF, the effect on the critical path is smaller. This requires analyzing the application for instructions, that are beneficial to be executed from the IRF such as loops, and instructions whose writing to the IRF would incur an energy overhead in relation to their execution amount. In the latter case, bypassing the instruction would be preferred.

Common to both of the two IRF design variations we used in our cores introduced in the next section, IRF execution starts with a compiler-inserted *header instruction*, which contains the number of entries to fill into the IRF. At compile time, the instruction scheduler groups sequential instructions into IRF *instruction windows*. These are groups, that can simultaneously exist in the IRF, and are indicated by the header instructions.

4.1 Basic IRF

The *basic* IRF from our original work [9] is described in Fig. 3. Two program counter registers are utilized here: one for addressing the global instruction memory address space and another for addressing IRF entries. In cases where execution does not fall through from the IRF and a conditional branch exits IRF execution, instructions in the branch not taken would not be executed. To eliminate unnecessary fetching, the IRF is accessed using *presence bits*, which use a single bit per IRF entry to indicate, whether an instruction has been fetched or not. If the presence bit is not set, the instruction is fetched and then executed. The presence bits are reset at the start of a new *IRF window* execution. This differs from the state-of-the-art work [23], where no presence bits are used.

The presence bits also prevent cases, where an instruction branches forward inside the IRF and the target instruction is not fetched. Without presence bits the compiler would be forced to discard every instruction window containing forward jumps from being placed to the IRF.

As a design choice, we implement IRF bypassing to avoid writing instruction with low execution counts to the IRF while simultaneously maintaining zero delay slot branching. In other words, we allow execution of code directly from the next level in the instruction memory hierarchy. To maintain good performance, this would require low access times for the next level in memory hierarchy. Otherwise, the maximum clock frequency would be limited by the memory access times. Previous work [23] avoids this issue by writing all instructions to the IRF and then executing them with no option for bypassing.

In order to not increase the instruction width due to the addition of new IRF control instructions, we implemented the special header instruction into the immediate control field of the TTA instruction, where enough free bit combinations were available. A comparator *pre-decodes* this field during the instruction fetch stage and stalls execution for one clock cycle, as the header instruction is only used by the instruction fetch unit and not executed by the rest of the core. At this time, length of the IRF instruction window and prefill amount are read from the instruction and stored into a register in the instruction fetch unit. Next instruction following the header is the first instruction in the instruction window.

To differentiate between global and IRF address spaces, we implement a new branch instruction targeting the IRF address space and use existing branch instructions to target the global address space. The unconditional *irfjump* instruction is only allowed when executing from the IRF and branches to a target inside the current instruction window. Regular branches are allowed anywhere in the code. Encountering them during IRF execution transfers execution into bypassing the IRF.

When IRF execution reaches the last instruction in the current instruction window or the last physical IRF entry and execution does not branch back into the IRF, execution is again transferred into bypassing the IRF with a fallthrough to the next code block. We implement these with hardware comparators in the instruction fetch unit.

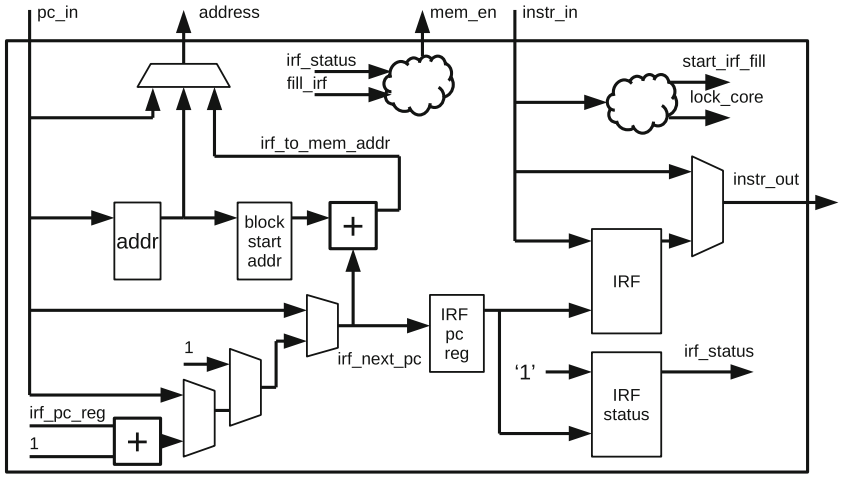

Figure 3 Instruction fetch unit of the proposed cores when using the *basic* IRF.

Adding the *irfjump* instruction to *LoTTA* core, where guarded execution is used for branching, increased the instruction size from 49 bits to 50 bits.

4.2 Improved IRF

According to our investigation [21] of IRF design choices, we optimize the IRF design in order to improve its energy-efficiency and maximum operating clock frequency. The *improved* IRF is depicted in Fig. 4. We remove the hardware presence assurance, which guarantees that instructions are written into the IRF before trying to execute them and instead, we move the presence assurance to software. This removes the status bit register from the IRF.

During compilation, groups of instructions, *instruction windows*, to be placed into the IRF are analyzed for forward branches. Different from the *basic* IRF, if there are forward branches in the window, the compiler writes the number of instructions to prefill when starting IRF execution, into the *header* instruction, in addition to the current window size. The prefill guarantees the presence of instructions, removing the need to check each IRF entry individually.

As an optimization to improve clock frequency, we implement conditional branches into the IRF in addition to the *irfjump* operation: *irfjumpnz* and *irfjumpz* correspond to branch not equal to zero and branch equal to zero instructions. To minimize the hardware overhead, these are implemented as having 1-bit condition operand.

4.3 Compiler Support

Efficient utilization of the IRF requires consideration of two questions: *which* instructions should be executed from the IRF and *when* should the IRF be filled. Program control analysis in the context of IRFs has been previously studied in detail [24]. We implement compiler support primarily targeting loops and nested loops as these are typically the most heavily executed code structures in applications. Forming the instruction windows is done as a post-pass following the compiler instruction scheduling.

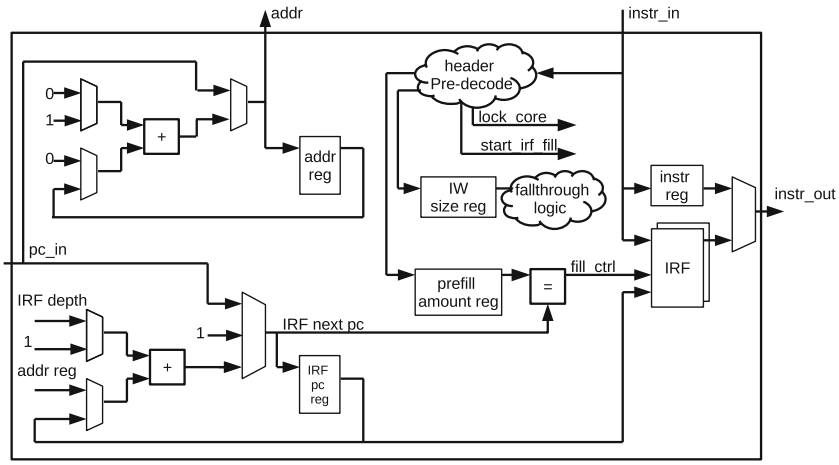

Figure 4 Instruction fetch unit of the proposed cores when using the *improved* IRF.

Our IRF instruction window allocation algorithm is presented in Fig. 5. Overall, on lines 3 to 9 the algorithm first splits program *basic blocks* (BBs) into blocks that fit into the IRF and merges them (lines 10 to 18) with two constraints:

1. Incoming jumps must target the first instruction of the instruction window.
2. Function calls are only allowed as the last instruction of an instruction window.

Respecting these constraints, upon encountering a function call, instruction windows are split at function calls, which are left as last instructions of a window. Returning from the function, execution continues from the next instruction address, which can contain a header instruction to start a new IRF instruction window, or start direct execution, bypassing the IRF.

Instruction windows with no backward branches are assigned to bypass the IRF on lines 19 to 26. For instruction windows with backward branches, the window's first local branch target and the last local branch instruction are marked on lines 23 and 24, as these are effectively the loop boundaries. The algorithm first assigns inner loops into IRF instruction windows and continues to outer loops if the IRF capacity allows, as inner loops are expected to be program hot spots.

On lines 27 to 38, code outside loop boundaries is removed from each instruction window, as this is code that is only expected to execute once each time an instruction window is programmed into the IRF.

When the instruction windows are completed, a header instruction is inserted into the beginning of the window on lines 39 to 44. As we stall execution during IRF programming, the minimum amount of stall cycles required (prefill amount) is calculated according to forward branches in the instruction window. Instructions in IRF windows branching back into the window are converted into IRF branch instructions on lines 45 to 53 and their targets are converted into IRF indices. As a last step, all branch targets are fixed with respect to the additional header instructions.

As an improvement over our previous work [9], code that is executed only once is removed from the end of instruction windows. This further eliminates unnecessary writes to the IRF registers, saving energy. Moreover, support for conditional IRF branches is added. For the software presence assurance of instructions, on line 41 the minimum amount of stall cycles is calculated, if the IRF supports it. In this work, this is done for *PeLoTTA* core, which uses the *improved* IRF.

As the *improved* IRF is used in *PeLoTTA* core, IRF operations for the conditional branching were required. *Irfjumpz* and *irfjumpnz*, corresponding to the *bz* and *bnz* for regular branching were added to *PeLoTTA*, resulting in an increase of the instruction word from 43 to 45 bits.

```
1: windowFitsToIRF: irfWindow.size + nextWindow.size < irfSize
2: canMergeIRFWindows: NOT irfWindow.endsInCall and
   nextWindow has no incoming branches from outside these two
   windows and irfWindow.fitsToIRF
3: for all basicBlocks in CFG do
4:     if basicBlock.size > irfSize then
5:         split BB to irfSize
6:     end if
7:     create a new irfWindow for bb
8:     place irfWindow into queue
9: end for
10: for all irfWindow in queue do
11:     nextWindow ← irfWindow.successor
12:     if canMergeIRFWindows then
13:         merge(irfWindow, nextWindow)
14:         requeue(irfWindow.predecessor)
15:     else
16:         remove irfWindow from queue
17:     end if
18: end for
19: for all irfWindow in irfWindows do
20:     if irfWindow.hasNoBackwardBranches then
21:         irfWindow.setBypassed
22:     else
23:         mark first local jump target
24:         mark last local jump position
25:     end if
26: end for
27: for all irfWindow in irfWindows do
28:     if NOT irfWindow.isBypassed then
29:         if NOT irfWindow.startsWithLocalBranchTarget then
30:             split irfWindow from beginning
31:             mark beginning as bypassed.
32:         end if
33:         if NOT irfWindow.hasLocalBranchAtEnd then
34:             split irfWindow from end
35:             mark the end as bypassed.
36:         end if
37:     end if
38: end for
39: for all irfWindow in irfWindows do
40:     if NOT irfWindow.isBypassed then
41:         calculate stall cycles needed due to forward jumps
42:         createHeaderInstruction(irfWindow)
43:     end if
44: end for
45: for all irfWindow in irfWindows do
46:     if NOT irfWindow.isBypassed then
47:         for all branch in irfWindow do
48:             if branch.destination inside irfWindow then
49:                 convert branch to local branch
50:             end if
51:         end for
52:     end if
53: end for
```

Figure 5 The IRF window allocation routine.

5 Evaluation

The proposed cores and the reference cores were 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* [25] 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* [26]. 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 simulating them at *register transfer level* (RTL) with Mentor Graphics ModelSim 10.5.

For the RISC cores, benchmarks were compiled with '-O3' optimization level. *Zero-riscy* uses a custom compiler of the PULPino project, based on GCC 5.4.0 and *LatticeMico32* uses a custom compiler from Lattice Semiconductor, also based on GCC.

To compare the *LoTTA* core with a traditional multistage RISC architecture in our previous work, the *LatticeMico32* core was used as a reference point. In addition to comparing *LoTTA* with *LatticeMico32*, in this extended work we compare *PeLoTTA* with a closer match, the recent low-power *zero-riscy* [7] core. *SuperLoTTA* is also compared to *zero-riscy* and *PeLoTTA*, as its architecture is closer to the latter than that of *LoTTA*.

For comparison to *LatticeMico32*, hardware operations, their latencies, and register file sizes were matched in *LoTTA*. Arithmetic and logic operations in both cores were made very similar with separate multiplier and barrel shifter units. Register files in all cores are 32 × 32 bit, with two read ports and one write port. Physical area of the *LatticeMico32* and *LoTTA* was validated to be roughly the same via ASIC place and route.

To evaluate our proposed core architecture with a recent low-power core intended for similar application domains, we used *zero-riscy* [7], a RISC-V core from the PULP [27] platform, and more specifically PULPino platform. For evaluation, we matched the number of pipeline stages, memory access delays and hardware operations closely in the *PeLoTTA-ZR* core, and adopted the latch-based register file used in *zero-riscy*. Of the two multiplier choices shipped with *zero-riscy*, a fast version was chosen over a slow variant. The PULPino framework was used to simulate the benchmarks on *zero-riscy* and generate switching activity files for power analysis after place and route. To provide a reference point of the *PeLoTTA* core also to *LatticeMico32*, we produced another subvariant with datapath components matching the those of *LatticeMico32*, named *PeLoTTA-LM32* in the results.

Comparison of the architecture features of the proposed and reference cores is listed in Table 3.

In addition to *LoTTA* and *PeLoTTA* being evaluated individually, to evaluate the two IRF design alternatives, *LoTTA* was also evaluated together with the *basic* IRF and *PeLoTTA* with the *improved* IRF.

All the cores evaluated were synthesized with Synopsys Design Compiler Q-2019.12. The process technology was 28 nm *fully depleted silicon on insulator* (FD-SOI) with 0.95 V voltage, with typical process corner and 25 °C temperature. After synthesis, place and route was performed with Synopsys IC Compiler II Q-2019.12. *Switching activity interchange format* (saif) files were produced with ModelSim 10.5 and used to estimate power for the synthesized designs.

5.1 Code Size

Compiled code sizes are listed in Table 4. As the *zero-riscy* core programs include operating system and debug related code, those were removed from the reported code sizes. *Jpeg* is not reported as the compiler could not fit it into the 32 kB instruction memory set in the PULPino framework.

For the TTA cores with non-byte-aligned instruction widths, results are reported as total instruction bits as bytes. On average, *LoTTA* program size is 1.84× larger than the LatticeMico32 code. *Coremark* has the largest difference, where *LoTTA* instruction bit amount is 3.5× 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*, *LoTTA* code size is smaller than that of *LatticeMico32*. This is due to loop unrolling and function inlining. These allow the wide TTA instructions to be efficiently utilized. Moreover, *LatticeMico32* requires 32 bits for each instruction, whereas in *LoTTA*, a maximum of 3 instructions can fit into the 49 instruction bit wide instruction word per cycle, in the best case resulting in 16.3 bits on average per instruction slot. However, it is to be noted that the instructions in the two cores cannot be directly compared, as in the TTA programming model, individual data *moves* are controlled, whereas in the RISC model, the internal moves are controlled by hardware based on the input operations. In this sense, the TTA instructions more or less correspond to the internal data moves of a RISC processor.

Table 3 Architecture comparison of evaluated cores.

	Instruction width (bits)	Architecture	Issue width	HW mul	HW div	mul latency	div latency	Pipeline stages	Data mem. access delay (cc)
LoTTA	49	TTA	3	✓		3 cc		1	1
PeLoTTA	45	TTA	3	✓		3 cc		2	1
SuperLoTTA	43	TTA	3	✓		5 cc		3	3
LatticeMico32	32	RISC	1	✓	✓	3 cc	34 cc	6	
zero-riscy	32 & 16	RV32IMC	1	✓	✓	3 cc	36/0 cc	2	1

Table 4 Cycle counts and program sizes.

	coremark	adpcm	aes	blowfish	gsm	jpeg	mips	motion	sha
	Cycle count								
LatticeMico32	582818	88355	75184	798794	26798	3018543	27348	7757	677708
zero-riscy	407036	149865	45268	942344	19872		26099	3846	781108
LoTTA	403416	81683	25690	673151	12745	8097755	23649	7797	543428
PeLoTTA	521658	82814	34855	661369	13071	2361449	27735	9525	559836
SuperLoTTA	802529	93691	41505	765160	16973	3275662	40023	7412	630984
	Code size (B)								
LatticeMico32	12400	9396	17052	6820	6000	17520	3328	8620	4148
zero-riscy	18292	9254	11660	5328	7290		5724	4472	4712
LoTTA	42979	16445	16372	6100	11851	50500	2597	6161	4269
PeLoTTA	13787	12298	13406	5607	10283	42979	2887	9052	3704
SuperLoTTA	18662	13949	16432	6440	14346	64775	3903	9858	4220

Compared to *LoTTA*, benchmark code size for the *PeLoTTA* core is smaller in all but *mips* and *motion* benchmarks. This is mostly due to the smaller instruction word in *PeLoTTA*, 43 bits as opposed to 49 bits in *LoTTA*. Although only visible in *mips* and *motion*, the conditional branches used in *PeLoTTA* typically affect code efficiency as they are not always scheduled in an optimal fashion. The predicated execution in *LoTTA* typically results in more efficient code.

The effect of IRF on the compiled code size was similar with the *basic* IRF and the *improved* IRF. Table 5 lists the code sizes for each benchmark on different sizes of the *improved* IRF. Code size with IRF is slightly larger than without IRF, due to the additional header instructions required for starting IRF execution. In *coremark, gsm* and *jpeg* the code size is smaller with IRF. This likely results from a different scheduling result due to the chaotic nature of *tcecc* in some cases.

5.2 Clock Frequency

Timing results are presented in Table 6. To evaluate the potential for maximizing serial performance, the *LoTTA* and *LatticeMico32* 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 LatticeMico32 reached 1667 MHz. The ALU output port in *LoTTA* can be used directly for predicated execution, and thus reduce need for branching, but the trade-off becomes visible in the critical timing path which ended up between the ALU output and the instruction fetch unit. *LatticeMico32* utilizes a six-stage pipeline without predication support, allowing a shorter critical path and higher clock frequency in this case.

Between *LoTTA* and *PeLoTTA-LM32*, replacing guarded execution with conditional branching and splitting the

Table 5 Performance and code size results for PeLoTTA core with the *improved* IRF. Code size did not vary significantly between IRF entry amounts.

	coremark	adpcm	aes	blowfish	gsm	jpeg	mips	motion	sha
	Cycle count								
8	511517	82728	34783	667430	12938	2397328	27153	7762	557779
16	502962	82728	34783	667560	12938	2389895	27153	7762	557779
32	500691	82728	34756	667560	12938	2477417	27153	7763	557779
64	500864	82728	34891	667239	12864	2508551	27153	7767	557523
128	583237	82728	34906	667239	12852	2583892	27153	7784	555210
256	583304	82728	34600	667986	12987	2578699	27153	8169	553668
512	583304	82728	34600	667986	12914	2581820	27153	8688	553154
	Code size (kB)								
	13.2	12.8	13.9	5.7	10.3	42.7	3.0	9.1	3.8

Table 6 Timing results for the evaluated cores after place and route on a 28 nm process technology.

	max C_f (MHz)	Result after
LatticeMico32	1667	Synthesis
LoTTA	1333	Synthesis
PeLoTTA-LM32	1670	Place and route
PeLoTTA-ZR	1790	Place and route
SuperLoTTA	2630	Place and route
zero-riscy	1230	Place and route

instruction fetch and decode stages allowed an increase in maximum clock frequency of 25%.

Adding the 3-stage execution pipeline, conditional branching and the additional ALU delay allowed an increase of 97% in the *SuperLoTTA* core maximum clock frequency in comparison to *LoTTA*. The maximum clock frequency of 2630 MHz is enabled by the efficient instruction fetch and decode logic of the TTA programming model, the lightweight interconnection network connecting components inside the core, and the simple register file that still supports the multi-issue capabilities. The critical path ended up inside the latch-based register file.

The effect of adding an IRF to *PeLoTTA-ZR* on maximum operating clock frequency is presented in Table 7. As reading instructions from the IRF ends up on the timing critical path, increasing the IRF size decreases the maximum clock frequency. With the smallest size of 8, the maximum clock frequency is reduced by 15%. With size 512, the reduction is 37%.

In our previous work, *LoTTA* 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. With the 10 ns constraint, the *LoTTA* core reached a maximum clock frequency of 311 MHz after synthesis, with the timing critical path starting from the ALU predicate output, through RF read into the instruction fetch unit and ending in the program counter register.

With the same configuration, *LatticeMico32* reached a maximum clock frequency of 351 MHz, with the critical path in the multiplier unit. *LoTTA* including the *basic* IRF with 256 entries reached a maximum clock frequency of 221 MHz.

Table 7 PeLoTTA-ZR place & route results with IRF. To obtain the maximum clock frequency, instruction memory timing characteristics modeled with 0.40 ns hold time for instruction read and 0.15 ns for address setup time.

IRF entries	max C_f (MHz)	Cell area (μm^2)
8	1612	10769
16	1538	12880
32	1515	16121
64	1351	22036
128	1266	33018
256	1266	55098
512	1136	99269

5.3 Execution Performance

Benchmark execution cycle counts for the evaluated cores are presented in Table 4. The data memory accesses are scheduled statically in TCE, so the reported cycle counts are instruction cycles. Due to the zero delay slot branching in *LoTTA*, its cycle counts are the lowest in all benchmarks except *jpeg* and *motion*. A large fraction of execution time in *jpeg* consists of manually written *memcpy* operations. Due to a chaotic effect in the TCE compiler between compiling for *LoTTA* and *PeLoTTA*, the latter resulted in a significant improvement in execution cycles, as the compiler unrolled the manual *memcpy* loop, unlike with *LoTTA*.

Compared to *LoTTA*, disregarding *jpeg* not to emphasize the effect of compiler *memcpy* behaviour, *LatticeMico32* requires a geometric average of 1.29× more cycles to execute the benchmark set. This is explained by the 6-stage pipeline as opposed to the 1-stage of *LoTTA*, as well as the ILP provided by LoTTA. In addition, predicated execution from RF and directly from the ALU output port help in saving clock cycles. In the control-oriented *Coremark*, the predicated ALU execution of *LoTTA* improved the cycle count by 10%. The best speedup (2.9×) was obtained in *aes*. Here the TTA compiler could exploit the instruction-level parallelism in the benchmark very efficiently.

Compared to *LoTTA*, *PeLoTTA* uses on geometric average 1.08× more cycles to execute the benchmarks. This is due to the additional instruction pipelining and conditional branching added to increase the operating clock frequency. *Zero-riscy* uses 1.34× more clock cycles compared to *LoTTA*. Although otherwise matched to *zero-riscy*, *PeLoTTA* is able to achieve lower cycle counts due to its multi-issue architecture and aggressive loop unrolling. *Zero-riscy* performs better in *Coremark* and *mips*, which are control-oriented benchmarks, and in the data oriented *motion*. In *Coremark*, with IRF sizes larger than 64 entries, the cycle count increases by 16%. This is caused by an inner loop, that does not fit into an IRF of 64 entries and its execution always exits on the second iteration. Due to its structure, this loop requires 99 instructions to be prefilled into the IRF before execution can resume. As the IRF block allocation routine does not in its current form take into account the loop iteration counts even if they are known at

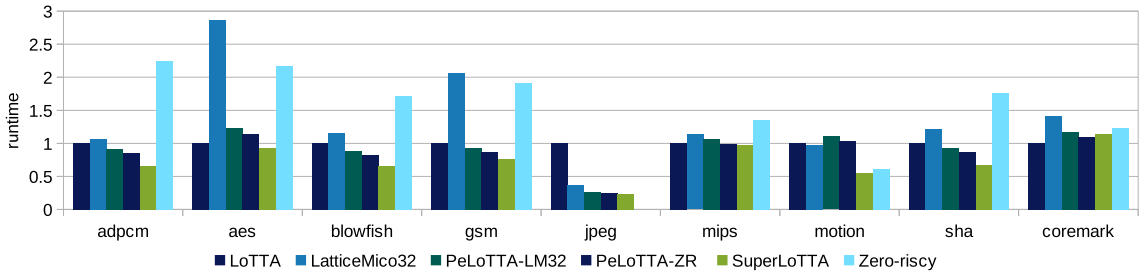

Figure 6 Runtime comparison. Results normalized to *LoTTA* per benchmark.

compile time, this inner loop always adds 99 stall cycles to execution when it is entered.

SuperLoTTA uses 1.35× more cycles compared to *LoTTA*. The largest increase in cycle counts is found in the control-flow-oriented *coremark* and *mips*. The increase is explained by the 3 delay slots in branching, conditional branches instead of predicated execution, and the added ALU latency.

On average, *SuperLoTTA* and *zero-riscy* consume close to same amount of clock cycles, although *SuperLoTTA* has a longer branching delay. This is explained by the multi-issue architecture and aggressive loop unrolling in *SuperLoTTA*. *SuperLoTTA* used less clock cycles in *adpcm*, *aes*, *blowfish* and *sha*.

Cycle counts for benchmarks when using *PeLoTTA* with IRF are listed in Table 5. As the number of IRF entries grows, the cycle count grows. This is explained by the IRF prefilling required for instruction presence assurance, and the stall cycle required when entering and exiting IRF execution. As the number of entries grows, the compiler allocates larger instruction windows into the IRF. This typically leads to increased prefill amount, which increases the cycle count. Also, the current IRF architecture used in TCE always assumes that there is no valid data in the IRF, when starting execution. This increases the cycle count, if prefill is required in an instruction window, that is executed multiple times from the IRF, but the execution switches to instruction memory in between.

Figure 6 presents absolute runtimes for the benchmarks. In nearly all cases, *SuperLoTTA* with the highest clock frequency is the fastest when measured in absolute time units. Only in *Coremark*, *PeLoTTA* with *LatticeMico32*-matched hardware is slightly faster. As seen from Table 4, even though it requires more cycles than any of the other evaluated core, it performs the best when measured in absolute runtime. The increase in cycles and achieved clock frequency stem from the same reason: increased pipelining in the instruction stream and compute units.

Organized in the order of geometric average of the runtimes over the benchmark set (excluding *jpeg*)), *SuperLoTTA* is the fastest with 36.7 µs, followed by *PeLoTTA-ZR*: 45.0 µs, *LoTTA*: 47.7 µs, *PeLoTTA-LM32*: 48.7 µs, *LatticeMico32*: 66.4 µs and *zero-riscy*: 72.6 µs.

5.4 Area

Area comparison for the evaluated cores after place and route is presented in Fig. 7. LoTTA without IRF is roughly the same size as the LatticeMico32. When incorporating an IRF, 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.

PeLoTTA matched with *zero-riscy* hardware is the smallest, followed by *PeLoTTA* matched with *LatticeMico32*. The difference between the two is mostly due to the latch-based RF being smaller than the flip-flop based, as can be seen from Fig. 7.

SuperLoTTA occupies more area than *PeLoTTA* due to the added registers for pipelining in the instruction stream,

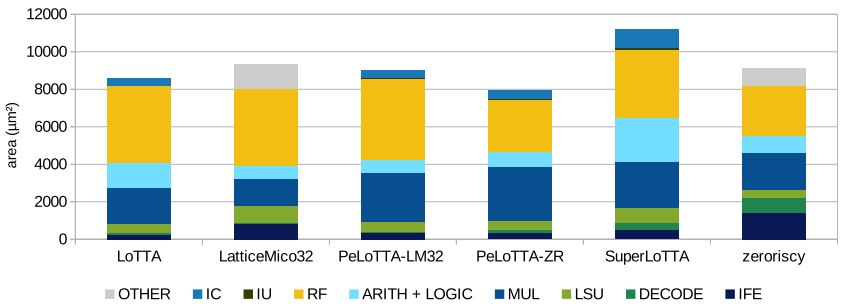

Figure 7 Cell area (µm^2) distribution after place and route.

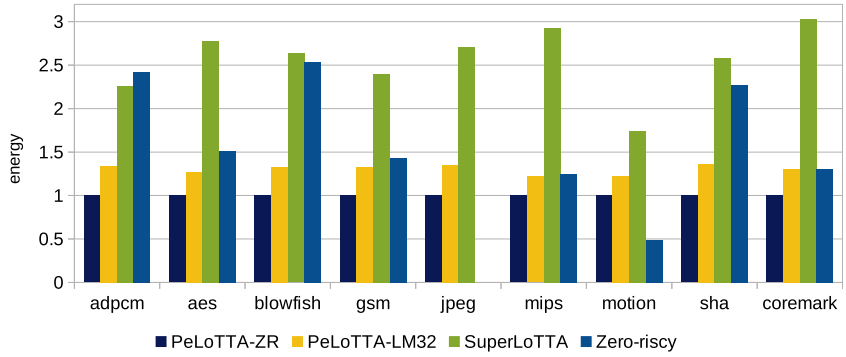

Figure 8 Core energy consumption compared to PeLoTTA-ZR.

load-store unit and ALU. Also, to achieve the higher clock frequency, the synthesis tool has replaced some cells with more energy-consuming and larger but faster versions.

5.5 Energy Consumption

Figure 8 presents the core energy consumption for the benchmark set relative to *PeLoTTA-ZR*. Here, *PeLoTTA-LM32* with *LatticeMico32*-matched hardware consumes slightly more energy than *PeLoTTA-ZR* due to the register-based RF. *Zero-riscy* consumes on average 58% more energy on the benchmarks than *PeLoTTA* and in the worst case, 175% more. *SuperLoTTA* consumes on average 153% more energy than *PeLoTTA* and in the worst case 203% more. In *motion*, *zero-riscy* consumes less energy than *PeLoTTA-ZR*, as it executes the benchmark in 2.5× less clock cycles.

In order to highlight the execution latency in conjunction with the energy consumption, Fig. 9 presents the *energy-delay-product* (EDP) of the cores relative to *PeLoTTA*. Here, *PeLoTTA* achieves 6.8× better EDP in the best case compared to *zero-riscy* and 2.5× on average. This is due to better overall execution cycle counts in the benchmarks, smaller power consumption and higher clock frequency. The difference to *LatticeMico32*-matched *PeLoTTA* is negligible, as it has a slightly higher clock frequency, compensating for its higher energy consumption in the EDP results. *PeLoTTA* has 2.1× better EDP in the best case over *SuperLoTTA*, but on average the difference is not significant.

To evaluate the effect of using different IRF sizes on the instruction stream overall in a processor, we evaluated a system, where the instruction stream hierarchy consists of an on-chip SRAM instruction memory and and IRF. Energy consumption for *LoTTA* core with different *basic* IRF sizes is presented in Fig. 10. Here, the combined core and instruction memory energy consumption is presented with an instruction memory size of 16k entries. As *LoTTA* has a 49 bit instruction word and 50 bit instruction word with IRF, we equalized the memories between *LoTTA* and *LatticeMico32* not according to the amount of bytes, but number of entries. This takes into account the wider instruction word in *LoTTA* for a fair comparison. Here, the energy benefit of the IRF increases, as the memory size increases. This is due to the low-power flavour *ITRS-LSTP* SRAMs energy consumption consisting mostly of the number of accesses to them, due to having a low standby power. The instruction SRAM access energy numbers were generated with Cacti-P [28].

Comparing the energy consumption of different IRF sizes with the proportions of instruction executed from either memory or IRF in Fig. 11, it can be seen that with small IRF sizes, when the IRF utilization is low, energy is not saved significantly or not at all. As the amount of IRF entries grows, so does the IRF utilization and the energy

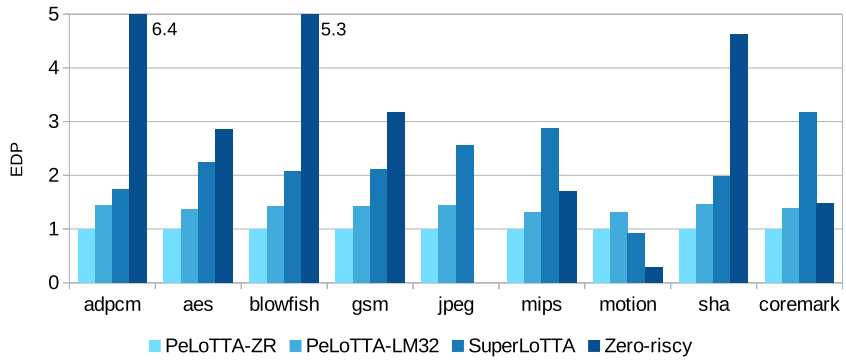

Figure 9 Energy-delay product of cores compared against PeLoTTA-ZR, lower is better.

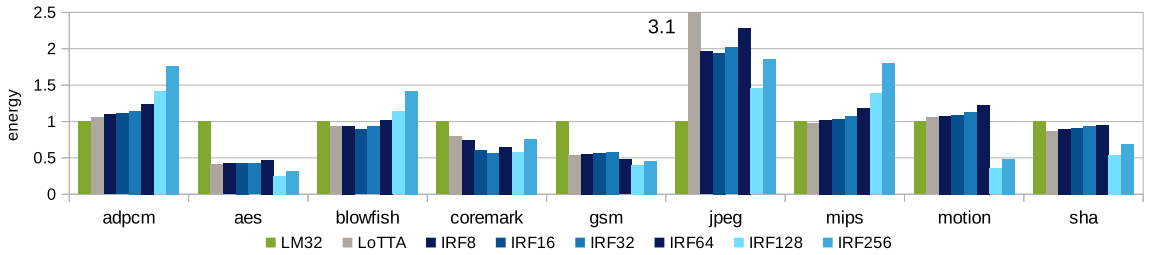

Figure 10 Core + instruction memory energy of LoTTA without and with *basic* IRF at 200MHz. Each benchmark normalized to LatticeMico32. Instruction memory size is 16k entries (LoTTA: 100 kB, LoTTA + IRF: 102 kB, LatticeMico32: 64 kB).

savings. Increasing the number of entries from 128 does not increase the IRF utilization significantly, worsening the energy saving as the larger, but less utilized IRF now consumes more energy.

Figure 12 shows comparison of the combined core and instruction memory energy consumption for *zero-riscy*, *SuperLoTTA* and *PeLoTTA*, which is compared with and without the *improved* IRF. Compared to *SuperLoTTA*, in *aes, gsm, mips, motion, gsm* and *coremark*, *zero-riscy* consumes significantly less energy. This is due to the increased cycle count in the benchmarks on *SuperLoTTA* due to the added pipelining in conjunction with the wider instruction word. In other words, *SuperLoTTA* reads significantly more bits from the instruction memory, consuming more energy as a trade-off for high clock frequency. At small IRF sizes, as seen from Fig. 13, the benchmarks mostly execute directly from the instruction memory instead of the IRF. Increasing the amount of entries in IRF also increases its utilization. From size 128 onwards, IRF utilization does not increase significantly, except in *aes*. This behaviour in utilization is translated to energy consumption in Fig. 12. At large IRF sizes, the large amount of entries starts to result in excessive energy consumption, countering the benefits gained from IRF. On geometric average, *SuperLoTTA* consumes 1.4× more energy than *zero-riscy* on the benchmark set.

PeLoTTA without IRF consumes slightly less energy in *adpcm* and *blowfish* compared to *zero-riscy*. In *motion*, *PeLoTTA* requires 2.5× more cycles to execute the benchmarks, resulting in the significantly higher energy consumption. Comparing the geometric average of energy consumption over the benchmark set, *PeLoTTA* consumes 42% more energy compared to *zero-riscy*.

Although the proposed cores consume less energy compared to *zero-riscy* in most benchmarks, the *zero-riscy* compiler is able to utilize a high number of compressed instructions in all benchmarks, leading to good energy-efficiency of the instruction stream. It is left as a future work for us to add instruction compression to further reduce energy consumption in addition to the locality based optimization provided by IRF.

Finally, evaluation of energy-delay product in a processor system with an on-chip instruction memory with either *zero-riscy*, *PeLoTTA* or *SuperLoTTA* is presented in Fig. 14. *SuperLoTTA* achieved the best individual benchmark EDP improvement compared to *zero-riscy* in *adpcm*, where it's EDP was 3.0× lower. On a geometric average over the full benchmark set, *SuperLoTTA* achieved and EDP within 1% difference to *zero-riscy*. In *Coremark, mips* and *motion*, *SuperLoTTA* had higher EDP compared to *zero-riscy*, due to using more clock cycles and, thus, increasing the instruction memory energy consumption as more instructions were fetched. In these cases, the higher clock frequency of *SuperLoTTA* could not compensate for the increased energy consumption, leading to higher EDP than *zero-riscy*.

PeLoTTA achieved a geometric average of 16% better EDP than *zero-riscy*. Like *SuperLoTTA*, it had worse EDP in *Coremark, mips* and *motion*. As the IRF size was increased,

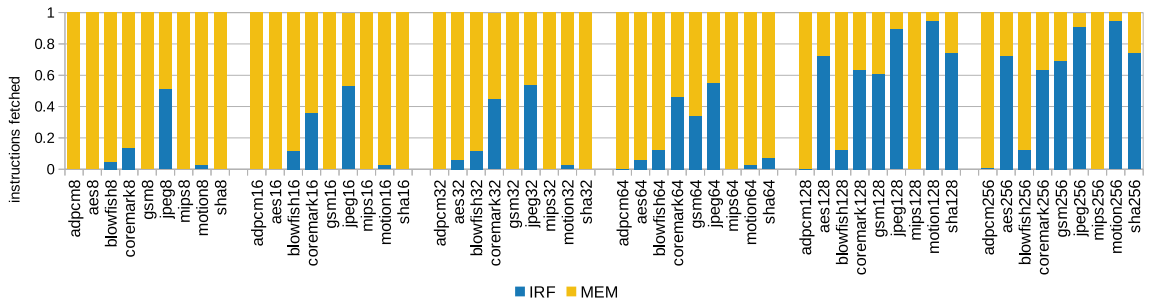

Figure 11 Fraction of instructions fetched from memory and from the *basic* IRF in LoTTA.

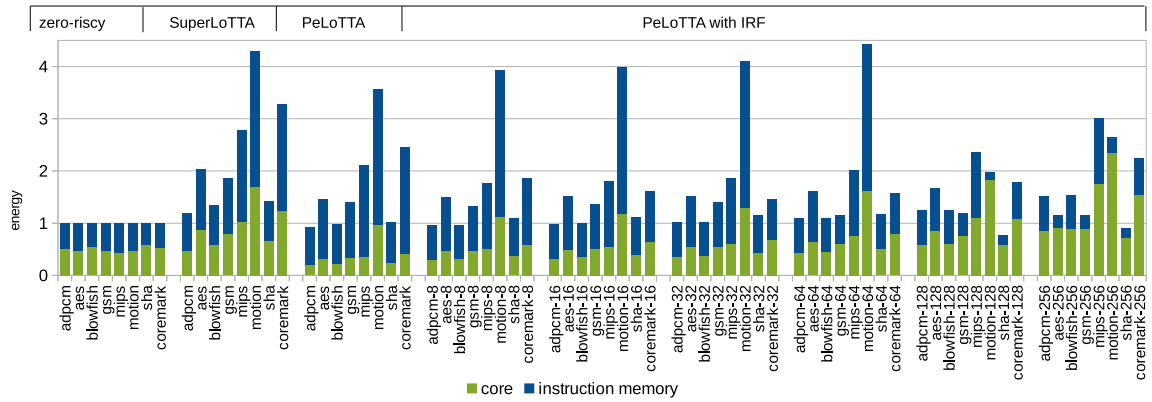

Figure 12 Comparison of processor energy consumption with *adpcm, aes, blowfish, coremark, gsm* and *sha*. Numbers normalized to zero-riscy. Instruction memory of 8192 entries, 32 kB on zero-riscy, 44 kB on SuperLoTTA and 46 kB on PeLoTTA.

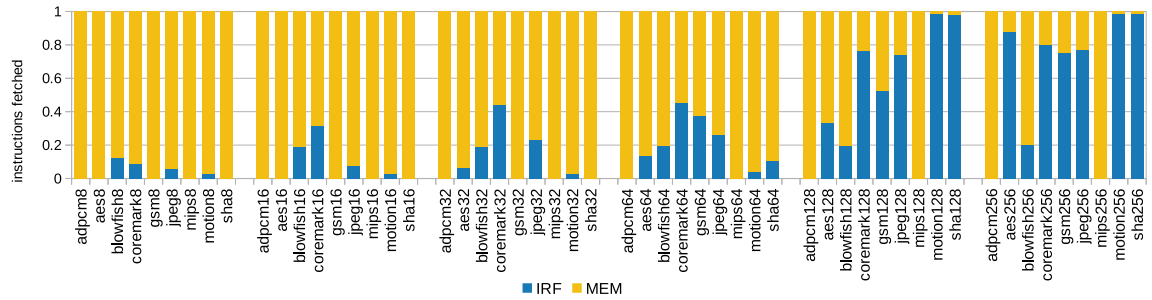

Figure 13 Instructions fetched from memory and from the *improved* IRF in PeLoTTA. Instruction memory has 16 k entries.

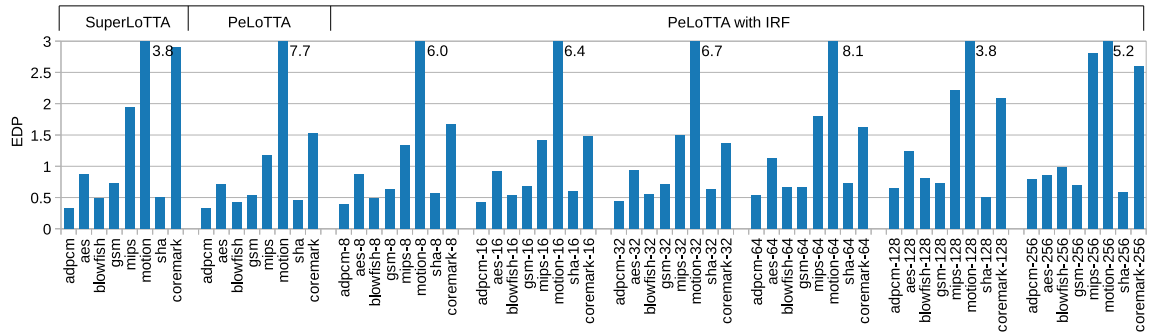

Figure 14 Relative energy-delay product, normalized to zero-riscy with an instruction memory of 8192 entries. Lower is better.

Table 8 Comparison of the proposed cores with related work.

	Technology	Max. C_f (MHz)	V_{dd} (V)	µW/MHz	pJ/cycle
SleepWalker [10]	65 nm	25	0.32–0.48	7.0	2.6
zero-riscy [7][a]	65 nm	560	0.8–1.2	2.3–11.0	
PeLoTTA-ZR	28 nm	1790	0.95	14.1	4.4
SuperLoTTA	28 nm	2630	0.95	63.6	9.2

[a]Results for *zero-riscy* from publication [7]

the EDP increased, as the maximum clock frequency decreased as illustrated in Table 7. Simultenously, energy consumption of the IRF increased and while the IRF utilization improved with larger IRF sizes, the combined effect was not enough to improve the EDP.

Finally, a comparison of *PeLoTTA-ZR* and *SuperLoTTA* is presented in Table 8. Although *SleepWalker* and *zero-riscy* have better µW/MHz values, our proposed cores provide similar order of magnitude for pJ/cycle values, with significantly higher maximum operating clock frequencies.

5.6 Analysis and Effect of IRF

As the IRF header instructions incur a stall of one clock cycle in program execution while the IRF window size is read in the proposed cores, there is overhead in execution time. Similarly, when entering IRF execution, there is a stall of one clock cycle. However, the increase from these stalls in all benchmarks was less than 0.8% and had no significant impact on performance.

With *PeLoTTA* and the *improved* IRF, cycle counts when using the *improved* IRF were in many cases slightly lower than *PeLoTTA* without IRF. This is due to the additional instruction pipeline register in *PeLoTTA*, that does not exist in *LoTTA*. If a section of code executes directly from instruction memory, there is always a delay of one clock cycle when branching. Depending on the program, the compiler delay slot filler may not be able to fill in useful instruction to these delay slots. If the same section of code is executed from IRF, branching inside the IRF has no delay. With 8 IRF entries, only *blowfish* and *jpeg* used more clock cycles with IRF in the design. From IRF size 128 onwards, also *coremark* starts using more cycles with than without the IRF.

Due to the code structures in the benchmark programs, utilization of the IRF is low, when IRF size is less than 128, as seen in Figs. 11 and 13. In *mips*, the IRF is never used, due to a large while loop containing all the code in the benchmark. This loop cannot be split by the TCE compiler to fit into the IRF. As similar structure is found in *adpcm*, where an encoding and a decoding function are executed within a for loop and do not fit into the IRFs evaluated. An optimization to improve the utilization and reduce energy consumption, would be allowing execution of code sequences such as loops from both the IRF and the next level of memory hierarchy consequently. In the current IRF design, if there is a loop that does not fully fit into the IRF, the compiler does not utilize the IRF at all. Partial execution of loops in this manner from IRF would reduce the energy consumption.

With the smallest *basic* IRF configuration, eight entries, the IRF is only utilized efficiently in *jpeg*. This is due to a heavily executed loop in the code containing exactly eight instructions. Compared to the *basic* IRF, due to replacing predicated branching with conditional branch instruction and different instruction scheduling, the *improved* IRF is not used as efficiently in *jpeg* with small IRF sizes. Both IRFs are 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.

5.7 Discussion

As previous work [23] has extensively studied the energy benefits of the instruction register files compared to small filter caches, and concluded that IRF can save energy over an already effective filter cache, we did not make comparisons to caches in this work.

Although an important design consideration for IoT devices, we left the optimization of standby energy consumption outside the scope of this article, as techniques such as retention registers and power gating are considered generic enough to be adapted on top of our approach as well.

As the IRFs are mapped as registers in the ASIC technology, and in the synthesis flow we used a simple clock gating scheme, where all the registers are gated using the same condition, there is likely room for implementing hierarchical clock gating to the IRF and doing so, decreasing the energy consumption.

As some of the benchmarks have no or poor utilization of IRF, for a processor system it would make sense to implement power gating to the IRF for these cases. These benchmarks should be compiled not to use the IRF, and executed directly from the instruction memory, powering off the costly registers in the IRF.

6 Conclusions

In this paper, we proposed three cores targeting mixed control flow and data processing applications in internet-of-things always-on devices. We evaluated design points targeting maximum clock frequency for high throughput tasks and energy-delay product for energy and delay critical tasks. We compared against two RISC cores, *LatticeMico32* and *zero-riscy*.

The TTA programming model alone saved on geometric average 60% of core energy compared to *zero-riscy*. As the most obvious drawback of the transport-triggered programming model used in the proposed cores is the larger program size, we designed, optimized and integrated an instruction register file to act as the first-level instruction store. This allowed us to mitigate the effect of the wide instruction word, allowing us to combine the excellent data path energy-efficiency of transport-triggered architectures with an energy efficient instruction supply.

Even though our proposed cores consumed less energy compared to *zero-riscy*, due to their long instruction word, the energy consumption was higher in a system including an instruction memory. However, in the case of an energy-optimized design point, we achieved on average 14% better EDP compared to *zero-riscy* and a best-case of 68% better EDP in a processor system with an on-chip instruction memory. In a high-performance design point we achieved 2.1× higher maximum clock frequency with a similar level of energy-delay product.

Evaluation of only the processor cores without the instruction memory hierarchy motivates further research on optimizing the instruction stream, as our energy-optimized core achieves nearly 2.5× improvement in energy consumption and an 8× improvement in energy-delay product in the best case. In addition to the IRF, we plan to investigate instruction compression to further mitigate the overhead of the instruction stream. Other plans for future work include moving even more of the control of the IRF from hardware to software, and developing more efficient IRF utilization strategies as these are the low hanging fruits for improvement.

Acknowledgements This work is part of the FitOptiVis project [29] funded by the ECSEL Joint Undertaking under grant number H2020-ECSEL-2017-2-783162.

References

1. Huawei Technologies, Anders, S., Andrae, G. (2017). Total consumer power consumption forecast. Presentation available: https://www.researchgate.net/publication/320225452_Total_Consumer_Power_Consumption_Forecast.
2. Belkhir, L., & Elmeligi, A. (2018). Assessing ICT global emissions footprint: trends to 2040 & recommendations. *Journal of Cleaner Production*, *177*, 03.
3. Magaki, I., Khazraee, M., Gutierrez, L.V., Taylor, M.B. (2016). Asic clouds: specializing the datacenter. In *Proceedings of the international symposium on computer architecture* (pp. 178–190).
4. Khazraee, M., Zhang, L., Vega, L., Taylor, M.B. (2017). Moonwalk: NRE optimization in asic clouds. *SIGARCH Computer Architecture News*, *45*(1), 511–526.
5. Silven, O., & Jyrkkä, K. (2007). Observations on power-efficiency trends in mobile communication devices. *EURASIP Journal on Embedded Systems*, *2007*(1), 056976.
6. Lattice Semiconductor. Latticemico32, May 2018. http://www.latticesemi.com/en/Products/Design/SoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx.
7. Schiavone, P.D., Conti, F., Rossi, D., Gautschi, M., Pullini, A., Flamand, E., Benini, L. (2017). 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* (pp. 1–8).
8. Waterman, A., Lee, Y., Patterson, D.A., Asanovic, K. (2011). The risc-v instruction set manual, volume i: base user-level isa. EECS Department, UC Berkeley, Tech. Rep UCB/EECS-2011-62.
9. Multanen, J., Kultala, H., Jääskeläinen, P. (2018). Energy-delay trade-offs in instruction register file design. In *Proceedings of the international workshop on signal processing systems, Tallinn*.
10. Bol, D., De Vos, J., Hocquet, C., Botman, F., Durvaux, F., Boyd, S., Flandre, D., Legat, J. (2013). 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*, *48*(1), 20–32.
11. Klinefelter, A., Roberts, N.E., Shakhsheer, Y., Gonzalez, P., Shrivastava, A., Roy, A., Craig, K., Faisal, M., Boley, J., Oh, S., Zhang, Y., Akella, D., Wentzloff, D.D., Calhoun, B.H. (2015). 21.3 A 6.45μW self-powered IoT SoC with integrated energy-harvesting power management and ULP asymmetric radios. In *Proceedings of the international solid-state circuits conference* (pp. 1–3).
12. Zhang, Y., Xu, L., Yang, K., Dong, Q., Jeloka, S., Blaauw, D., Sylvester, D. (2017). Recryptor: a reconfigurable in-memory cryptographic cortex-M0 processor for IoT. In *Proceedings of symposium on VLSI circuits* (pp. C264–C265).
13. Senni, S., Torres, L., Sassatelli, G., Gamatie, A. (2016). Non-volatile processor based on MRAM for ultra-low-power iot devices. *Journal on Emerging Technologies in Computing Systems*, *13*(2), 17:1–17:23.
14. Wang, Z., Liu, Y., Sun, Y., Li, Y., Zhang, D., Yang, H. (2015). An energy-efficient heterogeneous dual-core processor for internet of things. In *Proceedings of the international symposium on circuits and systems* (pp. 2301–2304).
15. Roy, A., Grossmann, P.J., Vitale, S.A., Calhoun, B.H. (2016). A 1.3μw, 5pJ/cycle sub-threshold msp430 processor in 90 nm xLP

FDSOI for energy-efficient IoT applications. In *Proceedings of the international symposium on quality electronic design* (pp. 158–162).
16. Teittinen, J., Hiienkari, M., Žliobaitė, I., Hollmen, J., Berg, H., Heiskala, J., Viitanen, T., Simonsson, J., Koskinen, L. (2017). A 5.3 pJ/op approximate TTA VLIW tailored for machine learning. *Microelectronics Journal*, *61*, 106–113.
17. Maxim Corporation (2004). Introduction to the MAXQ architecture.
18. Jääskeläinen, P., Viitanen, T., Takala, J., Berg, H. (2017). HW/SW co-design toolset for customization of exposed datapath processors. In *Computing platforms for software-defined radio* (pp. 147–164).
19. Hoogerbrugge, J., & Corporaal, H. (1994). Register file port requirements of transport triggered architectures. In *Proceedings of the international symposium on microarchitecture, San Jose* (pp. 191–195).
20. Lipovski, G. (1976). The architecture of a simple, effective control processor. In *Proceedings of Euromicro symposium on microprocessing and microprogramming* (pp. 7–19).
21. Multanen, J., Kultala, H., Jääskeläinen, P., Viitanen, T., Tervo, A., Takala, J. (2018). LoTTA: energy-efficient processor for always-on applications. In *Proceedings of the international workshop on signal processing systems, Cape Town* (pp. 193–198).
22. Jääskeläinen, P., Kultala, H., Viitanen, T., Takala, J. (2015). Code density and energy efficiency of exposed datapath architectures. *J. Signal Process. Syst.*, *80*(1), 49–64.
23. Black-Schaffer, D., Balfour, J., Dally, W., Parikh, V., Park, J. (2008). Hierarchical instruction register organization. *Computer Architecture Letters*, *7*(2), 41–44.
24. Park, J., Balfour, J., Dally, W.J. (2010). Fine-grain dynamic instruction placement for l0 scratch-pad memory. In *Proceedings of the international conference on compilers, architectures and synthesis for embedded systems* (pp. 137–146).
25. EEMBC—The Embedded Microprocessor Benchmark Consortium (2018). Coremark benchmark May. http://www.eembc.org/coremark.
26. Hara, Y., Tomiyama, H., Honda, S., Takada, H. (2009). Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, *17*, 242–254.
27. Conti, F., Rossi, D., Pullini, A., Loi, I., Benini, L. (2016). Pulp: a ultra-low power parallel accelerator for energy-efficient and flexible embedded vision. *Journal of Signal Processing Systems*, *84*(3), 339–354.
28. Li, S., Chen, K., Ahn, J.H., Brockman, J.B., Jouppi, N.P. (2011). Cacti-p: architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *Proceedings of the international conference on computer-aided design* (pp. 694–701).
29. Al-Ars, Z., Basten, T., de Beer, A., Geilen, M., Goswami, D., Jääskeläinen, P., Kadlec, J., de Alejandro, M.M., Palumbo, F., Peeren, G., et al. (2019). The FitOptiVis ECSEL project: highly efficient distributed embedded image/video processing in cyber-physical systems. In *Proceedings of the 16th ACM international conference on computing frontiers. CF '19* (pp. 333–338).