Tailoring Instruction-Set Extensions for an Ultra-Low Power Tightly-Coupled Cluster of OpenRISC Cores

Michael Gautschi*, Andreas Traber*,
Antonio Pullini*, Luca Benini*
*Integrated Systems Laboratory, ETH Zurich,
Gloriastrasse 35, 8092 Zurich, Switzerland.
{gautschi, atraber, pullinia, lbenini}@iis.ee.ethz.ch

Michele Scandale§, Alessandro Di Federico§,
Michele Beretta§, Giovanni Agosta§
§Dipartimento di Elettronica, Politecnico di Milano,
Piazza Leonardo da Vinci 32, 20133 Milano, Italy.
{michele.scandale,alessandro.difederico}@polimi.it,
{michele.beretta,giovanni.agosta}@polimi.it

***Abstract*—Baseline RISC instruction sets for ultra-low power processors are constantly being tuned to reduce cycle count when executing computation-intensive applications. Performance improvements often come at a non-negligible price in terms of area and critical path length and imply deeper pipelines and complex memory interfaces. This penalizes control-intensive code execution and significantly increases cost and complexity of building multi-core clusters. In addition, some extensions are not easily exploited by compilers and may increase code development effort, especially when considering parallel applications. In this paper we describe our efforts in enhancing a baseline open ISA (OpenRISC) and its LLVM compiler back-end to significantly reduce execution cycles while minimizing the impact on core micro-architecture complexity, number of pipeline stages, area and power. In addition, we improved the core micro-architecture to streamline its integration in a tightly-coupled cluster, sharing instruction cache and data memory, thereby further enhancing parallel execution efficiency. The combined effect of ISA, compiler and micro-architecture evolution gives an average energy efficiency boost of 59% on vector intensive code and 41% otherwise, at an area and power increase of 2.3% and 18% on a four-core processor cluster.**

I. Introduction

The increasing performance requirements in computing intensive and mobile devices raise the need for powerful, but also ultra-low power (ULP) processors and computing architectures. Since power scales quadratically with the supply voltage, it is more energy efficient to operate a digital circuit near the threshold voltage of transistors [1]. The reduced speed can be compensated with multiple processors working in parallel at lower voltages [2]. To maximize energy efficiency, single processors can be turned off depending on the current workload, while resources like memories and caches can be shared among the cores. To form an energy-efficient multi-core cluster capable of processing computation intensive applications, the processing element (PE) needs to fulfill certain requirements like having a small area footprint, being energy efficient and having a shallow pipeline allowing fast inter-core communication and data exchange to allow fine-grained data and task-level parallelism. Moreover, great attention must be paid to integration of the cores to avoid the creation of a bottleneck in the system due to the instruction and data interface. We claim, that it is not enough to just use the most energy efficient 32 bit processor on the market, the ARM Cortex M0+, to construct an energy efficient multi-core cluster, but point out the requirement of careful optimization for an operation in a tightly-coupled cluster. First, the ARM RTL code is not publicly available which makes an efficient cluster integration unfeasible. And second, while its energy efficiency of only $9.39\,\mu$W/MHz [3] would suite very well for ULP-operation, it is not meant to be used for computation intensive or parallel applications. Other architectures, like the ARM Cortex M4 are more powerful in this domain but also show $3-4\times$ higher requirements in area and power consumption [3]. This increase in area and power consumption comes from moving to a deeper pipeline and from enriching the instruction set with DSP like features and more advanced ALU operations. In order to keep the area and power figures under control a careful evaluation of additional features and instructions must be done, in particular in ultra-low power contexts.

In our work we have chosen the OpenRISC architecture which is a) open source and suitable to be optimized to work in a multi-core environment, and b) shows a low area footprint with only 35.5 kGE which allows the core to achieve an energy efficiency of $25.8\,\mu$W/MHz in 65 nm technology, which is competitive with an ARM Cortex M4 with an energy efficiency of $32.8\,\mu$W/MHz at similar area costs in a 90 nm technology [3]. While the computational efficiency of the OpenRISC architecture has been improved in terms of instructions per cycle (IPC) [4], it is missing important DSP-like features which allow a Cortex M4 to execute a program in less cycles due to its enriched instruction set. Enhancing an instruction set with very specific instructions is the key to increase performance in application-specific computing [5].

In order to push the OpenRISC architecture to higher performance and energy efficiency, we have looked into possible ISA-extensions which a) allow the core to significantly reduce the number of executed instructions to run computation-intensive applications, b) do not significantly increase the core's area and power consumption, c) can be efficiently integrated in a multi-core cluster, and d) are sufficiently general such that the compiler can automatically map their intermediate representation to the proposed instructions. Throughout this paper we discuss the benefits and costs of ISA-extensions, like hardware loops, extended addressing modes, vector ALU support and others in a multi-core cluster. Further, we present solutions how to efficiently implement these ISA-extensions in an ULP-cluster by taking into account the performance of each PE and its joint functionality with the cluster components, such as the shared instruction cache and the shared data memory. On our benchmarks, the execution of computation intensive benchmarks shows $1.1-5\times$ faster execution than with the initial ISA, and $1.1-4.5\times$ faster execution than on

978-1-4673-9140-5/15/$31.00 ©2015 IEEE

a Cortex M4. The faster execution and slightly higher power consumption combine to an overall energy efficiency boost of 47.8% compared to the initial ISA. The area of the extended core increases by 25% and the power efficiency per core of 33.8 μW/MHz on average is comparable with an ARM Cortex M4 (32.8 μW/MHz in 90 nm [3]). Integrated in the multi-core cluster the area overhead diminishes to only 2.3%.

The rest of this paper is organized as follows. In Section II we present the PULP cluster and OpenRISC architecture. In Section III we introduce the proposed ISA-extensions and its costs. Section IV describes several aspects to efficiently include the extensions in a multi-core cluster. In Section V we report area, power, and timing results of the ISA-extended OpenRISC cores on a four core cluster environment and in Section VI we draw our conclusions.

II. Overview of the PULP-Platform

PULP (Parallel processing Ultra-Low Power platform) represents an effort to design a many-core platform responding to the demands of heavily-constrained embedded applications. Such applications would greatly benefit from a low-power, flexible computing fabric that is able to provide significant performance when needed and remain in a very low-consumption state otherwise. To achieve these goals, PULP features *clusters* of simple PEs that can be used to exploit both coarse- and fine-grain data- or task-level parallelism. At the same time, voltage and frequency scaling can be controlled at a fine granularity to achieve high energy efficiency when the performance constraints are more relaxed or when the power budget is tighter. Fig. 1 shows the PULP cluster architecture. In its current configuration it consists of four PEs, a shared 4-way associative I$ with four cache banks of 1 KB each and a L0 buffer of 128 bits per core holding the most recent cacheline to reduce access contentions at the cache banks [6]. The refill port of the shared I$ connects to the system bus together with an L2 memory, and several peripherals (not shown). The PEs have a multi-banked tightly coupled data memory (TCDM) acting as a shared scratchpad memory, instead of private data caches to avoid memory coherency overhead. TCDM is further divided into 16x4 KB SRAM banks, and 16x0.5 KB standard cell based memory (SCM) banks allowing the cluster to operate at ultra low voltages to achieve maximum energy efficiency [7]. Intra-cluster communication is based on a high bandwidth, low-latency interconnect, implementing a word-level interleaving scheme to reduce the access contention to TCDM banks. A lightweight, low-programming-latency, multi-channel DMA enables fast and flexible communication with other clusters, the L2 memory and external peripherals. In the following we focus on a single core and its integration in the cluster.

A. The OpenRISC Processing Element

The OpenCores community [8] developed the OpenRISC architecture, an open-source processor using a GCC-based toolchain. A RISC architecture is well-suited to be integrated in a tightly-coupled multi-core cluster because of its low area footprint and the low pipeline depth allowing to interact with other processors in a single cycle. In a previous work we developed OR10N, a complete redesign of the micro-architecture in order to balance pipeline stages, and increase IPC [4]. The redesigned core is divided into four pipeline stages, *instruction fetch (IF)*, *instruction decode (ID)*, *execute (EX)*, and *write back (WB)* and achieves near-optimal IPC values of 1. All operations can be completed in a single cycle except for

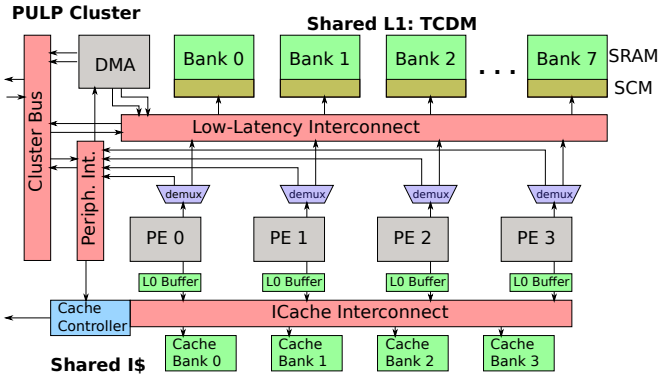

Fig. 1. PULP cluster featuring four OpenRISC processor cores, a shared instruction cache, eight TCDM banks utilized as L1 memory and a DMA for fast, concurrent data movements.

multiplications which are pipelined once, and can lead to stalls if the result is used in the subsequent cycle. While the core was being attached to an instruction and data memory [4], it has now been integrated in a PULP-cluster by connecting it to an I$ and a low-latency interconnect. Implemented in the cluster, the OR10N core utilizes 35.5 kGE. In this work we focus on increasing the efficiency of the generated machine code by introducing zero-overhead hardware loops, auto-incrementing memory operations, a more efficient multiplier architecture, and vectorial ALU operations. As we will show, getting rid of control code in small loops can lead to speedups of up to 2×. Auto-incrementing memory operations allow to get rid of instructions to maintain counters and addresses, by storing the updated memory address back to the latch-based register file. Ultimately, a vector unit that allows to concurrently process four 8-bit or two 16-bit values has been implemented in order to increase the throughput of the core. However, such instruction extensions are only useful if the compiler is able to produce such instructions. Therefore, we have modified the backend of the LLVM compiler to automatically generate code for the proposed ISA extensions and used it to evaluate the costs and benefits of the introduced ISA-extensions.

III. ISA-Extensions for OpenRISC

A. Hardware Loops

Zero-overhead hardware loops are a common feature in many processors, especially DSPs. Basically, a hardware loop is an implementation of a countable loop that avoids the need to explicitly test the loop counter and perform the branch. This is achieved by providing the hardware with information about the trip count and the beginning and end address of a loop, which are then used by specialized hardware in the computation of the next program counter (PC). The impact of hardware loops can be amplified by the presence of a loop buffer, i.e. a specialized cache holding the loop instructions, which removes any fetch delay [9]. In OR10N, we evaluated up to four nested hardware loops through the instructions shown in Tab. I. Each hardware loop has associated 3 special purpose register: `HWLP_START` and `HWLP_END` for the start and end address of the loop, and `HWLP_COUNT` for the loop counter.

Hardware loop setup can be performed either explicitly, by initializing each SPRs using the *lp.start*, *lp.end* and *lp.count* (or *lp.counti*) instructions, or initializing them in a single instruction using *lp.setup* (or *lp.setupi*).

Hardware loops are implemented in the micro-architectural

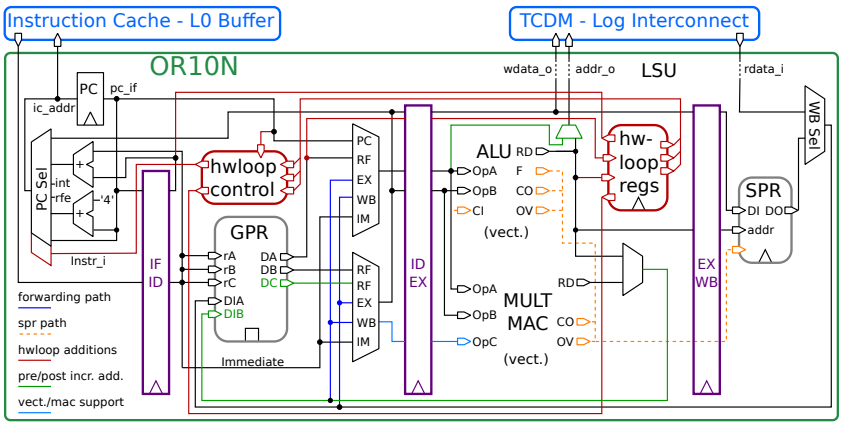

Fig. 2. Simplified block diagram of the OpenRISC micro-architecture, highlighting the additional hardware for hardware loop support.

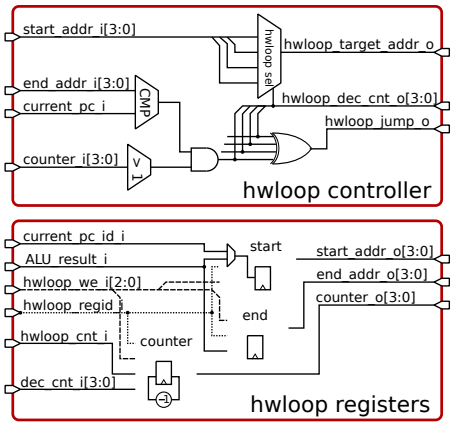

Fig. 3. Detailed implementation of the hardware loop control and register block.

TABLE I. ZERO-OVERHEAD HARDWARE LOOP OPERATIONS

Each hardware loop is identified with an ID: `L0`-`L3`. The notation `HWLP_START[J]` is used to refer to the `HWLP_START` register of the J^{th} hardware loop.

Instruction format and Opcode	Semantics
`lp.start J, S (eg. lp.start L0, 10)` `000010 000 JJ SSSSSSSSSSSSSSSSSSSS`	`HWLP_START[J]=sext(S*4)+PC`
`lp.end J, S (eg. lp.end L0, -8)` `000010 001 JJ EEEEEEEEEEEEEEEEEEEE`	`HWLP_END[J] =sext(E*4)+PC`
`lp.counti J, C (eg. lp.counti L0, 8)` `000010 010 JJ CCCCCCCCCCCCCCCCCCCC`	`HWLP_COUNT[J]=zext(C)`
`lp.count J, rA (eg. lp.count L0, r5)` `000010 011 JJ AAAAA----------------`	`HWLP_COUNT[J]=[rA]`

`lp.setupi J, E, C (eg. lp.setupi L0, 4, 8)`
`000010 100 JJ CCCCCCCCCCCCEEEEEEEE`
	`HWLP_START[J]=PC+4`
	`HWLP_END[J] =zext(E*4)+PC`
	`HWLP_COUNT[J]=zext(C)`

`lp.setup J, E, rA (eg. lp.setup L0, 8, r5)`
`000010 101 JJ AAAAAEEEEEEEEEEEEEEE`
	`HWLP_START[J]=PC+4`
	`HWLP_END[J] =zext(E*4)+PC`
	`HWLP_COUNT[J]=[rA]`

level with only two additional blocks. A controller, shown in the upper part of Fig. 3 and a set of registers to store the hardware loop variables. The controller is a purely combinational block which checks if the current PC matches one of the end addresses, and if the counter is greater than 1. In case both checks are true, the hwlp-controller informs the main controller to set the next PC to the corresponding start address of the loop. If two or more end points are equal, the controller gives priority to the lowest hwlp-ID (i.e., `L0` has the highest priority). Since the performance gain is maximized when the loop body is small, it is not beneficial to support a lot of register sets. Therefore, we introduced two register sets in order to allow two nested loops. The support of additional hardware loops would bring marginal performance improvements at a non-negligible cost in terms of area ($\approx$ 1.5 kGE per register set).

B. Extended Addressing Modes

Along with *Zero-overhead hardware loops*, we evaluated the performance and area impact of extended addressing modes. In the basic OR10N implementation only one type of load and store (*ld/st*) was available, in which the effective address was computed by adding a base address stored in a register and an offset encoded as an immediate value. The OR10N core was extended by implementing *ld/st* with both base and offset in register and *ld/st* with pre- and post-increment with both immediate and register offset.

Supporting all the variants of those instructions requires the addition of 65 new opcodes. To avoid saturation of available opcodes in the OpenRISC ISA, the new instructions are encoded using only 4 main opcodes and a sub-opcode field, which imposes a limitation on the immediate size from 16 to 11 bits. Out of those 65 new instructions, 3 instructions (1 for word, 1 for half word, 1 for byte transfer) are dedicated to each type of store, 6 instructions, coding different data sizes and sign extension, for each type of load with overall 5 new types of *ld/st*. These instructions are encoded reusing as much as possible the encoding scheme of the OpenRISC, keeping the source and destination registers at the same positions as shown in Tab. II.

TABLE II. LOAD/STORE ADDRESSING MODES.

Register-register addressing mode is expressed with the notation `rX(rY)`. Auto-incrementing addressing modes are identified with a `!` before (preincrement) or after (postincrement) the base address. MMMMM in the sub-opcode is used to indicate bits that are dedicated to differentiate each type of load/store.

Old Instruction format	Opcode
`l.s[bhw] I(rA), rB`	`MMMMMM IIIII AAAAA BBBBB IIIIIIIIII`
`l.l[bhw][zs] rD, I(rA)`	`MMMMMM DDDDD AAAAA IIIII IIIIIIIIII`
New Instruction format	**Opcode**
`l.s[bhw] rD(rA), rB`	`010101 DDDDD AAAAA BBBBB -----0 1MMMM`
`l.l[bhw][zs] rD, rB(rA)`	`010111 DDDDD AAAAA BBBBB -----0 1MMMM`
`l.s[bhw] I(rA!), rB`	`010100 IIIII AAAAA BBBBB IIIIII 0MMMM`
`l.s[bhw] rD(rA!), rB`	`010100 DDDDD AAAAA BBBBB -----1 1MMMM`
`l.l[bhw][zs] rD, I(rA!)`	`010110 DDDDD AAAAA IIIII IIIIII 0MMMM`
`l.l[bhw][zs] rD, rB(rA!)`	`010110 DDDDD AAAAA BBBBB -----1 1MMMM`
`l.s[bhw] I(!rA), rB`	`010101 IIIII AAAAA BBBBB IIIIII 0MMMM`
`l.s[bhw] rD(!rA), rB`	`010101 DDDDD AAAAA BBBBB -----1 1MMMM`
`l.l[bhw][zs] rD, I(!rA)`	`010111 DDDDD AAAAA IIIII IIIIII 0MMMM`
`l.l[bhw][zs] rD, rB(!rA)`	`010111 DDDDD AAAAA BBBBB -----1 1MMMM`

In OR10N the effective address is calculated in the ALU and then used to access the memory during regular *ld/st* and pre-increment operations. In case of a post-increment the address generation is bypassed and the memory is addressed with the base address. Loads with pre- or post-increment need to write two registers at the same time (the data read from memory and the incremented address pointer) and this required the addition of an extra write port to the register file. Due to the non criticality of this path, storing the incremented address in the write back stage can be avoided and the register file can be written directly in the current cycle. To support stores with the offset or increment in a register an extra read port was required, in fact, 3 different values have to be fetched from the register file at the same time (base address, offset and data to write).

The additional write and read port costs less than 1 kGE due to its non critical timing and its latch based implementation.

C. ALU Vector Support

Aiming for a more efficient processing of 8, and 16 bit data leads to the introduction of a vectorized ALU where the datapath is segmented into two, or four parts and allows to compute up to four bytes in parallel leading to a speedup of up to a factor of four. Such operations are also known as subword parallelism [10], packed-SIMD or micro-SIMD [11] instructions. However, the 32-bit operations remain the norm, meaning that extending a processor with such vector capabilities is only promising if the area and power overhead can be kept at a minimum. In order to extend our 32-bit OpenRISC ISA with subword parallelism, we have added two new opcodes for vector operations, one for ALU operations and one for vectorial comparisons. Each operation is available in three different vector modes for halfword (h) and byte (b) operations:

- `lv.inst.{h,b}     rD, rA, rB`,
- `lv.inst.{h,b}.sc  rD, rA, rB`,
- `lv.inst.{h,b}.sci rD, rA, I`,

where the first is a register to register operation, and the other two are vector operations with a register `rA` and a scalar replication of the register `rB`, or an immediate. The operations based on an adder and shifter have been realized by splitting the data path in four segments. The full 32 bit result is computed by chaining the four adder results with the carry. Multiplications and MACs on the other hand, are complicated because of the additional muxing to support four concurrent multiplications.

The OpenRISC ISA supports full 64 bit result multiplications and MAC operations, which cannot be implemented within a single cycle without impacting the maximum frequency. In a previous implementation [4], the multiplication was realized with a two cycle multiplier and MAC operations were based on a special purpose accumulator which is accessible through special instructions. Given the fact that the full 64 bit result is often not required, and that the compiler is not always able to group instructions in a way such that no stalls occur, we have decided to simplify the multiplication by only generating the 32-bit results. This allows to support vectorial multiplications with subword selection [10] in a single cycle. Notice, that the full 64-bit product can still be generated by the addition of four partial products which can be generated in sequence. While in the original implementation three cycles were required to generate a 64-bit result, with subword selection it is possible to create it with 10 instructions. Reducing the multiplications to 32-bit makes the 64-bit accumulation register useless. Instead of the large accumulator, a normal register can be used as accumulation register which leads to two major advantages: a) it is possible to concurrently maintain multiple accumulation registers and b) the additional delay of moving data back and forth from the accumulation special register to a general purpose register vanishes since the accumulator is placed in the GPR in the first place.

Implemented in the micro-architecture of OR10N as depicted in Fig. 2, the vectorial ALU and fused MULT/MAC unit increase the core area by 6 kGE which accounts for 13% of the core but less than 1% of the complete cluster. The additional read port of the register file is shared with the advanced addressing modes and costs less than 1 kGE. In Section V we show the benefit of the new multiplier with and without vectorial support.

D. Other Extensions

Along with the presented instruction extensions, several small ALU extensions such as *min, max, avg, abs* have been implemented. The area penalty of these extensions is less than 1%. These instructions are seldom used, however they can speed up significantly operations, such as normalization.

IV. Implementation in the ULP-Cluster

To keep the hardware complexity at a minimum, the introduced ISA-extensions have been implemented to a large extent with existing resources. The combined costs of all extensions, including all vectorial operations, are 9 kGE which is 25% of the core area. As pointed out, a good core architecture is not enough for an efficient integration in a multi-core cluster which is why we optimized the data and instruction interface without increasing the critical path. In the following we highlight three cluster-specific integration details which allow the processor to work more efficiently in a multi-core cluster.

A. Reducing Cache Accesses with a L0 Buffer

The amount of energy consumed by an instruction cache is not to be underestimated. In fact, a core with a very high IPC is accessing the instruction cache every cycle. Keeping this in mind and the fact that hardware loops are most effective if the loop body is small and contains no branches, adding a small L0 buffer between the instruction cache and the fetch interface (shown in Fig. 1) is a promising approach to lower power consumption by reducing the cache accesses. We have chosen a 128 bit wide L0 buffer, which is capable of holding the most recent cacheline. Hence, this architecture benefits if the compiler is capable of placing hardware loops aligned with 128 bits. In particular, when paired with a shared instruction cache, this approach is very effective since it significantly reduces access contention at the cache banks [6] and allows a loop to be fetched only once if the size does not exceed four instructions. Even though the buffer can only hold 4 instructions, this is not a restriction on the loop size. Larger loops would still benefit from the L0 buffer because only every fourth request has to be forwarded to the cache controller.

B. Unaligned Memory Access

Data structures are not always word aligned, and even if the compiler is aware of a vector unit, it is not always possible to align data. For example, in a simple 2D-filter on 8 bit data types, as depicted in Fig. 4, it is not possible to read aligned data in every cycle, thus leading to unnecessary masking and shifting or resorting to byte-wise loads. To support unaligned memory accesses in the tightly-coupled cluster, we have two options: 1) extend the data interface to 64 bit — 2) implement the unaligned memory access in 2 cycles.

Since the critical path in the cluster is already on the return path of the data memory, and that reading 64 bit of data means doubling the size of the interconnect and also the number of banks, the former option is not the most promising. In fact, by doubling the size of the interconnect, the depth increases due to its logarithmic tree. Moreover, since the delay of the interconnect depends on the depth, it would impact the length of the cluster's critical path, leading to a lower energy efficiency. For this reason we did not further evaluate this solution. The second approach enables unaligned memory

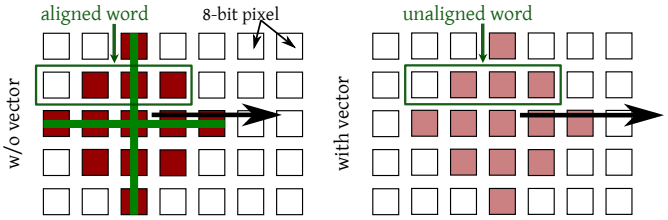

Fig. 4. Example showing the benefit of unaligned access when computing a stencil over an image with 8 bit pixels.

access with two subsequent memory requests, which are then merged into a single word by the load store unit. This hardware implementation does not add to the critical path and brings no additional hardware complexity to the core. Since the TCDM serves requests within a single cycle, it is possible to read unaligned data in only two cycles. On the other hand, to perform the same load operations, an architecture without support for unaligned data requests would require at least five instructions. Two cycles to read the two data items, two cycles to shift data and one cycle to combine them in a single register.

Our implementation of unaligned memory requests does allow to use the vector unit more often and is much more efficient in code size, and latency than a software only support. Compared to a single cycle load operation, our implementation does not require significant additional hardware resources and is therefore very well suited for the multi-core cluster.

C. Branch Prediction to Balance the Paths to I$

Branching was initially performed by using the forwarded flag from the ALU, thus allowing to branch without stalling the pipeline or having to predict and probably revert the pipeline. This was particularly important when processing loops. With hardware loop support this argument falls apart and stall free branching is no longer required for efficient loop handling. Moreover, since the L0 buffer delays the instruction request path, branch prediction becomes a requirement to prevent slowing down the system. Specially at lower voltages, where the cluster becomes extremely energy efficient, this path becomes critical. Synthesizing the cluster without branch prediction and an L0 buffer of 128 bits in a 28 nm process leads to a frequency degradation of 13% at 0.6 V. Branch prediction would split this path, but also slightly increase the runtime due to mispredictions. Given the fact that we want to operate at ultra-low power, and that the misprediction penalty is only one cycle in our flat pipeline, we have chosen the most simple branch predictor, which is to always take backward branches, never take forward branches.

The branch prediction removes the critical path to the I$, while increasing the number of cycles in our benchmarks by only 1.32 % on average, with a maximum of 3.4 %. In combination with hardware loops, this number decreases to only 0.7 % additional cycles. Hence, we conclude that even a very simple branch predictor, if paired with hardware loops, does only increase the number of cycles by 0.7 % while allowing the core to be clocked 13% higher at low voltages.

V. Performance, Area, and Power Results

In the following we are discussing the performance, area, and power impact of the introduced ISA-extensions. The execution time in number of cycles has been measured in RTL-simulations, while the area increase was determined with Synopsys Design Compiler in topographical mode, and the power estimations were performed on a back-annotated placed&routed netlist at nominal voltage of 1.2 V and a frequency of 50 MHz. The complete design, including a final tapeout of the PULP chip, was done in UMC 65 nm technology. Unless otherwise specified, all power and area numbers are related to this technology.

A. Execution Time

Each proposed instruction set extension has been analyzed individually by enabling it in the compiler through dedicated flags. Specifically, we compared the basic ISA with 4 different configurations, which, respectively, enable 2 sets of hardware loops (H), pre- and post-increment addressing modes (I), the new single cycle multiplication and three operand MAC unit (M) and vector operations with unaligned access (V). Finally, we also performed a comparison with the ARM Cortex M4 (Cortex M4). Fig. 5 incrementally shows the benefit of each ISA-extension on our benchmark applications which range from basic matrix multiplications, based on 8, 16 and 32 bits, through convolutions, filters and cryptographic algorithms. In particular matrix multiplications are presented in two forms: the classical implementation with row-by-column products, and an optimized version with row-by-row products preceded by the transposition of the second matrix, being a good candidate for auto-vectorization. All benchmarks are executed on a single core. Hardware loops and automatic addressing updates bring speedups up to $1.75\times$. The MAC unit has a good impact on computational intensive benchmarks like convolution and matrix multiplications and allows to process those benchmarks up to $2.25\times$ faster. The use of the vector unit can bring an additional boost when it is applicable. E.g., on the matrix multiplication on 8 bit data, our vector optimization achieves a $5\times$ speedup with respect to the original ISA. Besides the impact of the ISA extensions, this figure results from the ability to vectorize row-by-row products. Similarly, on the 16 bit optimized matrix multiplication we achieve an overall $3.5\times$ speedup. On the classical row-by-column implementation on 8 bit data, data access on the column cannot be efficiently vectorized as there is no hardware support for either strided or gather/scatter load/store operations. On 16 bit data matrix multiplication the compiler does not apply vectorization as it is detected as not effective by the heuristics.

B. Area and Power Efficiency

Fig. 7 shows the increase in area and power with each additional feature. We can conclude that even though the area per core increases by 25% to 44.5 kGE, the overhead at cluster level remains small with only 2.3% of overhead due to the presence of interconnects, peripherals, and large amount of memory in the cluster. The total power consumption of one core running at 50 MHz with all features enabled is 1.688 mW, which translates to an average energy efficiency of $33.8\,\mu$W/MHz. The core shows a higher power consumption during the execution of vector heavy code where an energy efficiency of $47.8\,\mu$W/MHz is achieved.

At the maximum frequency of 362 MHz, which is not affected by the ISA-extensions, the four-core cluster is capable of processing 1.4 GOPS at a power budget of 93 mW of which 55% is consumed by the cores. With respect to a single core implementation, the four-core cluster can process four times more operations, at a power increase of only 67%. Thus, moving from a single core to a four-core cluster increases the power-efficiency by a factor of 2.4.

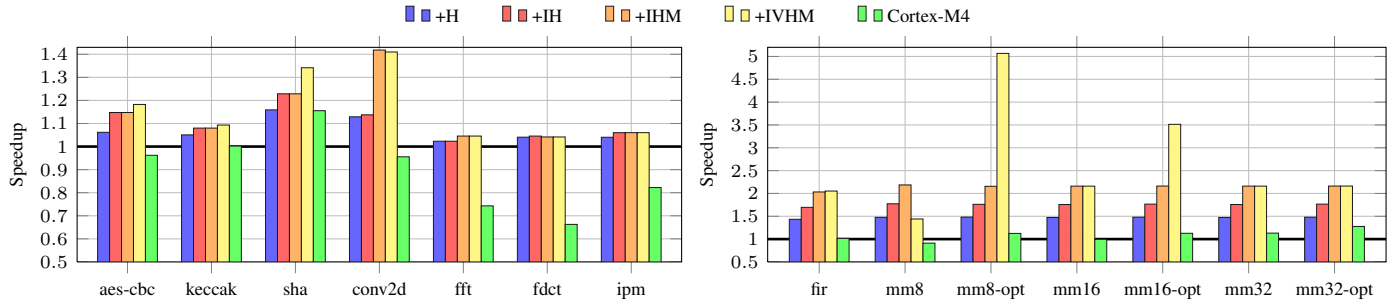

Fig. 5. Comparison of the speedup with the introduced ISA-extensions versus the initial ISA. Number of cycles are measured in RTL simulations for the OR10N core and in case of the Cortex M4 on the real hardware. All the results are normalized w.r.t. the number of cycles on the plain OpenRISC ISA.

C. Core and Cluster Energy Efficiency

In Fig. 6 the total amount of energy required to run each benchmark is shown for the original ISA, the extended ISA with and without vector unit, and a Cortex M4. To calculate the numbers for the Cortex M4, an energy efficiency of 16.7 μW/MHz is assumed, which is scaled down to 65 nm from a 90 nm technology [3]. In normal execution, where no vector code is used, the energy savings are 41.1% on average. Due to the additional speedup of the vector unit, the savings increase to 59.6% in the vector intensive matrix multiplications. When comparing the efficiency to an ARM Cortex M4, we observe that the extended ISA is up to 48% more energy efficient when processing vector code due to the vector instructions. If no vector code can be used, the performance of the proposed ISA shows no clear trend towards Cortex M4 or OpenRISC.

Taking into account all cluster resources, a cluster consisting of the improved cores consumes 18% more power with respect to a cluster with the initial micro-architecture. Taking the complete cluster power into account, the energy savings of the cluster based on the extended ISA with all features enabled ranges from 39 % to 66 %. On average the cluster is 47.8 % more energy efficient than the initial architecture.

VI. Conclusions

We have evaluated several instruction extensions for the OpenRISC-based OR10N core targeting an efficient integration in a multi-core cluster with a shared memory and instruction cache. The proposed ISA-extensions include hardware loops, pre- and post-increment addressing modes, new MAC instructions, and a set of vector instructions. The introduced ISA-extensions allow the core to compete with an ARM Cortex M4 in terms of runtime in cycles and in energy efficiency. Compared to the initial ISA, the support of the extended ISA increases the area of the OR10N core by 25% but at the same time allows to execute computation intensive applications up to 5× faster. The combined effort of ISA-extensions and smart cluster integration results in a total energy improvement of 47.8% on average while increasing the area and power of the four-core cluster by only 2.3% and 18%.

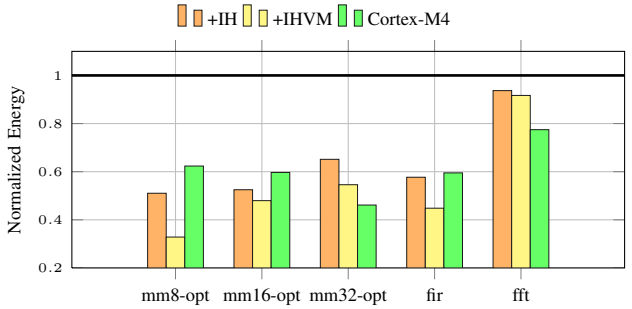

Fig. 6. Energy efficiency of the core with the optimized ISA integrated in the multi-core cluster in UMC 65 nm. All the results are normalized w.r.t. the absorbed energy on the plain OpenRISC ISA.

Acknowledgment

This work was partially supported by the IcySoC RTD project evaluated by the Swiss NSF and funded by NanoTera.ch with Swiss Confederation financing.

References

[1] R.G. Dreslinski et al., "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, Feb 2010.

[2] A.Y. Dogan et al., "Low-power processor architecture exploration for online biomedical signal analysis," *Circuits, Devices Systems, IET*, vol. 6, no. 5, pp. 279–286, Sept 2012.

[3] ARM, "ARM Cortex M0+/M4," http://www.arm.com/products/processors/cortex-m/, [Online; accessed 7-May-2015].

[4] M. Gautschi, et al., "SIR10US: A tightly coupled elliptic-curve cryptography co-processor for the OpenRISC," in *IEEE Int. Conf. Appl.-specific Syst. Archit. Processors (ASAP)*, June 2014, pp. 25–29.

[5] M. Arnold and H. Corporaal, "Designing domain-specific processors," in *Int Symp. Hardw./Softw. Codesign*, 2001, pp. 61–66.

[6] I. Loi, D. Rossi, G. Haugou, M. Gautschi, and L. Benini, "Exploring Multi-banked Shared-L1 Program Cache on Ultra-Low Power, Tightly-Coupled Processor Clusters," *CF '15*.

[7] P. Meinerzhagen, C. Roth, and A. Burg, "Towards generic low-power area-efficient standard cell based memory architectures," in *MWSCAS*, Aug 2010, pp. 129–132.

[8] OpenCores Community, "OpenRISC Community Portal," http://opencores.org/or1k/Main_Page, 2014, [accessed 18-September-2014].

[9] G.-R. Uh et al., "Techniques for effectively exploiting a zero overhead loop buffer," in *Compiler Construction*. Springer, 2000, pp. 157–172.

[10] R. Lee, "Subword parallelism with max-2," *Micro, IEEE*, vol. 16, no. 4, pp. 51–59, Aug 1996.

[11] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "A comparison between processor architectures for multimedia applications," in *ProRISC*, 2004, pp. 138–152.

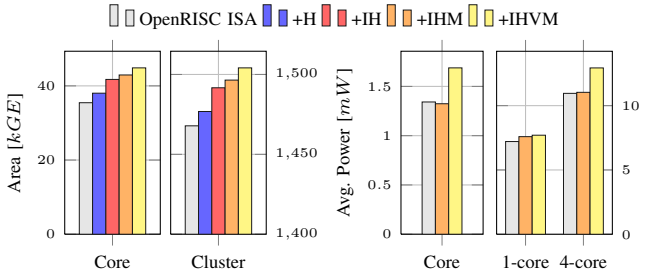

Fig. 7. Area and power comparison with different ISA-extensions on core and cluster level. Power is shown on a cluster with 1 and 4 cores.