Energy-Delay Trade-Offs in Instruction Register File Design

Joonas Multanen, Heikki Kultala, Pekka Jääskeläinen
Tampere University of Technology, Finland
Email: {firstname.lastname}@tut.fi

***Abstract*—In order to decrease latency and energy consumption, processors use hierarchical memory systems to store temporally and spatially related instructions close to the core. *Instruction register file* (IRF) is an energy-efficient solution for the lowest level in the instruction memory hierarchy. Being compiler-controlled, it removes the area and energy overheads involved in cache tag checking and adds flexibility in the separation of the instruction fetch and execution. In this paper, we systematically evaluate for the first time the effect of three IRF design variations on energy and delay against an unoptimized baseline IRF. Having instruction fetch and decode with IRF in the same pipeline stage allows minimal delay branching, but results in low operating clock frequency and impaired energy delay product compared to splitting them into two stages. Assuring instruction presence in IRF before execution with software reduces the area and increases maximum clock frequency compared to assurance with hardware, but requires compiler analysis. With a proposed compiler-analyzed instruction placement and co-designed hardware implementation, energy consumption with the best IRF variant is reduced by 9% on average with EEMBC Coremark and CHStone benchmaks. The energy delay product is improved by 23% when compared to the baseline IRF approach.**

I. Introduction

The era of *Internet-of-Things* (IoT) increases demands for the energy-efficiency and performance of devices. Battery-dependent devices, such as surveillance cameras, drones and sensor nodes operate on a limited energy budget. Their applications often require "burst style" execution: After idling for long periods, they might be waken up to react to external events and execute demanding data and control oriented tasks with low response latency.

For maximal energy-efficiency, fixed function accelerators are typically used. However, they offer poor flexibility as tasks not known at design time cannot likely be executed on the same fabricated hardware. Modern applications prefer programmable solutions due to the complex, growing computational needs [1].

Application-specific instruction-set processors (ASIPs) can deliver efficiency and flexibility, but the instruction supply can be considered as an additional overhead; software instructions just control the processor execution, but do not perform actual data processing. The instructions are typically read from on-chip caches or memories implemented with SRAMs, which can consume up to half of the power consumption [2], [3] of the processing. Ideally, a processor would be programmable, but offer the performance and energy-efficiency of a fixed function accelerator.

Processors include hierarchical instruction memory organizations to keep temporally related code close to the core, in order to allow accessing instructions from faster and more energy-efficient storages. For this purpose, various different storage components have been developed. *Dynamic caches* fetch instructions from the next hierarchy level upon a cache miss. Their benefit is that they require no knowledge of the cache details from software developers, but their unpredictable timing can become an issue in timing-critical applications. *Locked caches* address this by allowing fetching to be disabled for the duration of a task with special locking instructions. *Loop buffers* are dedicated to cache loops of the program, exploiting the observation of them usually being the hot spots in the program.

Instruction *scratchpad memories* (SPMs) are a more generic structure; they can store also blocks with more complex control flow while streamlining the caching hardware by removing the cache tag checking, but require the programmer or the compiler to add instructions to load their content and usually need separate instructions to branch around inside the SPM. *Instruction register files* (IRFs) [4] are like SPMs, but typically smaller in size to enable implementation with small register files instead of memories leading to even better energy efficiency and latency. Efficient utilization of IRFs and SPMs require careful compiler code analysis in order to determine optimal instruction placement.

Previous work considers design alternatives only for the distribution of IRFs among processor function units and concentrates on evaluating the IRF as a means to reduce energy consumption. However, as compute demands on low-power devices increase with IoT and edge computing, energy-efficiency has become at least as critical design aspect as performance. This paper presents the first in-depth evaluation about the implementation of three different fetch strategies compared to a baseline IRF from our previous work [5] and evaluates the energy and delay trade-offs of each strategy.

II. Instruction Register File Concepts

IRFs are programmer visible, thus presenting different design choices at the architecture and microarchitecture level. At the architecture level, IRFs require the compiler to decide and control when to execute from the IRF and when to load instructions to it from the lower hierarchy level. For this, a user-visible control interface such as control bits are added to instructions, or special IRF control instructions are typically used.

When considering the internal implementation details (microarchitecture), some sort of *presence assurance* is required to ensure that the instructions to be executed from the IRF

978-1-5386-7656-1/18/$31.00 ©2018 IEEE

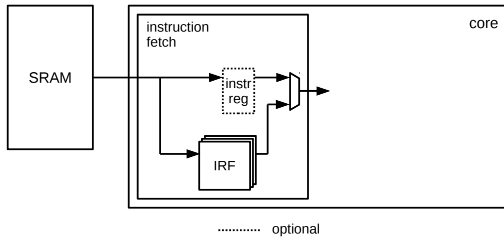

Fig. 1. The core used as evaluation platform for IRF variants.

are present. An alternative is to use a software-based implementation by prefilling all or some of the instructions in the *instruction window* (IW) (a set of instructions that can be placed simultaneously in the IRF) before their execution. Another option is to include hardware logic, such as a *presence bit* for each IRF entry and fetch the missing instructions on demand.

In IRF *bypassing*, instructions are fed directly from the next instruction hierarchy level to the core. As it is not beneficial to execute instructions that are executed only once from the IRF, and as the IRF allows isolation of instruction fetching from their execution, some consideration is required for two aspects of IRF instruction placement:

1) *Which* instructions to place in the IRF?
2) *When* should the instructions be fetched to the IRF?

A special design consideration results from how to handle *fallthroughs* which occur when the program execution does not branch at the last instruction of a *basic block* (BB), but continues from the next sequential instruction address. Alternative solutions include hardware detection of IW boundaries, or by inserting branches out of the IRF in software.

III. Evaluated IRF Design Variants

We chose interesting IRF implementation variants for closer inspection and integrated them into the instruction fetch unit of a processor core as shown in Fig. 1. In all of the variants, we used special *header* instructions as the control mechanism. As there are unused bit combinations available in the core immediate control field, we used them to indicate header instructions. The instruction fetch unit *pre-decodes* the immediate control field and reads the header upon the correct pattern. The processor is stalled for one cycle during this time, while the length of the instruction window is read from the header into the *IW size register* seen in Fig. 2. Then, execution starts from the first index of the IRF.

In the baseline IRF [5], instructions are fed directly to the core when bypassing and filling the IRF. This has the advantage of minimal delay in branching, as the branch target is available on the next cycle. However, this is detrimental to maximum clock frequency as the critical path starts from the next level instruction storage data read port, going through both the *instruction fetch* and *decode* units and looping back into the instruction address port.

For presence assurance in the baseline, a *presence bit register* is read during each IRF execution. If the presence bit for an entry to be executed next is not set, that instruction is fetched and the presence bit is set.

In all of the evaluated alternatives, fallthroughs are detected by hardware and cause the next instruction to be fetched from the next level in hierarchy. At this point a new instruction window can be loaded and started via a header instruction, or the execution can start bypassing the IRF. At each IRF execution cycle, the IW size register is compared to the *IRF next PC*. Upon detection of a *fallthrough* to the next code BB, execution is stalled and the next instruction address is calculated as *addr reg + IW size reg*.

The differences in the evaluated variants are described in the following.

A. Pipelined

To address the long critical path incurred from bypassing instructions in the baseline IRF, we insert an *instruction register*, where bypassed instructions are pipelined into, as seen in Fig. 2. Increasing the maximum clock frequency in this way presents a trade-off of increased clock cycles due to the additional fetch stall cycles. Compared to the baseline, a stall cycle is incurred when execution branches out of the IRF or when an instruction is not present.

B. iw-fill

A simplified logic diagram of the *iw-fill* variant is illustrated in Fig. 3. This variant of the IRF fetches an entire instruction window into the IRF, when starting execution from it. The whole instruction window is fetched in order to ensure, that due to forward branching, IRF execution cannot reach an instruction which has not been fetched. This variant removes

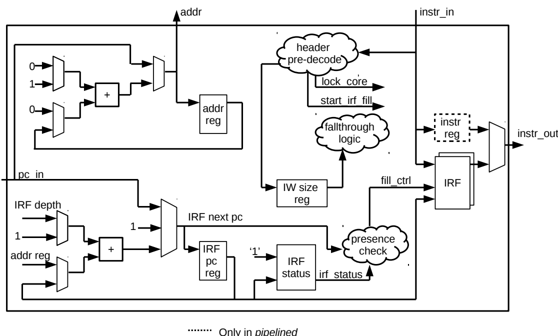

Fig. 2. Simplified logic of the Baseline/*pipelined* instruction fetch unit.

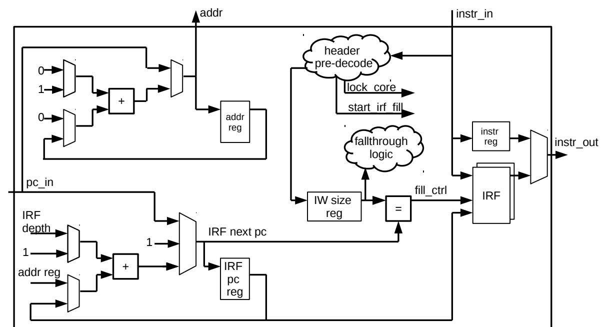

Fig. 3. Simplified logic of the instruction fetch unit with *iw-fill* variant.

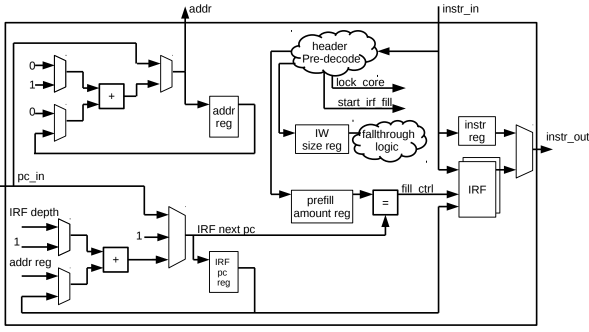

Fig. 4. Simplified logic of the instruction fetch unit with *fw-jump-fill* variant.

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

Fig. 5. The instruction window allocation routine.

the need for a presence bit register, but incurs additional clock cycles, as execution is stalled for the length of the instruction window.

C. *fw-jump-fill*

Similar to *iw-fill*, this variant removes the presence bit checking. Since the reason for prefilling instructions are forward jumps, this variant stalls the execution for a compiler-analyzed amount of cycles that guarantees that it is not possible to execute an instruction that has not yet been fetched, if the window contains forward jumps. The prefill amount is encoded into the header instruction along with the instruction window length. A logic diagram is presented in Fig. 4. Compared to the *iw-fill*, prefill logic and the *prefill amount register* are added. This is required, since the variant continues filling the IRF while IRF execution starts.

IV. Compiler Support

As a first step to the placement of more complicated control structures, our implementation focuses on loops and nested loops, as they are typically program hot spots. We implemented an algorithm for allocating the instruction windows as a post-pass after instruction scheduling as presented in Fig. 5 and detailed in the following.

First, all program basic blocks are split into blocks whose maximum size is the IRF size and assigned as individual instruction windows. These windows are then merged with two conditions:

1) Incoming jumps must target the first instruction in the IW.
2) Function calls are not allowed inside an IW.

The IW, however, can end in an function call. Conforming to the second condition, if a function call is encountered, the instruction window is split to exclude it.

The algorithm starts allocation from inner loops and continues to outer loops if the IRF capacity allows them. As it is only beneficial to write instructions to the IRF if they are executed multiple times, blocks with no backward jumps into itself are discarded. Instead, these blocks are bypassed into the instruction register.

As a last step, branch target addresses are fixed, taking into account the inserted header instructions. For jumps inside the IW, we implemented a *near jump* instruction, whose target is an IRF index. For other jumps, a *global jump* is used, targeting the actual memory address space.

For presence assurance in the *fw-jump-fill* variant, analysis of the prefill amount is required. First, for each BB in an instruction window, the compiler calculates the earliest possible execution cycle. This cycle is relative to the execution cycle of the first instruction in the IW. The earliest possible execution cycle is calculated by traversing the *control flow graph* (CFG) in top-down direction, with the first BB in the IW having a minimum execution cycle of 0.

When encountered, the blocks are queued for processing. When processed, a basic block inherits the minimum execution cycle of its predecessor, to which its individual minimum execution cycle is added, unless the successor has a smaller minimum execution cycle count. When the minimum execution cycle of a BB is updated, the block is queued again in order to propagate the update to its successors. As backward control flow edges from backwards jumps of loops always imply a higher execution cycle than the first iteration of the loop, these do not propagate and are discarded.

The compiler then compares the earliest possible execution cycle of each basic block to the position of the BB inside the IW. The largest individual count is selected as the prefill/stall count for the IRF.

TABLE I. Benchmark instruction type distribution (%).

	control	mem	ALU	NOP	data move & RF
coremark	4.7	9.9	17.1	38.7	29.5
adpcm	0.6	12.8	17.5	45.0	24.2
aes	1.7	11.1	20.8	37.8	28.7
blowfish	1.4	11.9	22.0	31.4	33.3
gsm	2.8	6.3	20.3	38.4	32.3
jpeg	2.6	9.6	15.3	30.3	42.2
mips	6.1	7.9	23.0	33.2	29.8
motion	1.9	13.0	18.4	37.8	28.9
sha	0.8	8.9	24.7	26.5	39.1

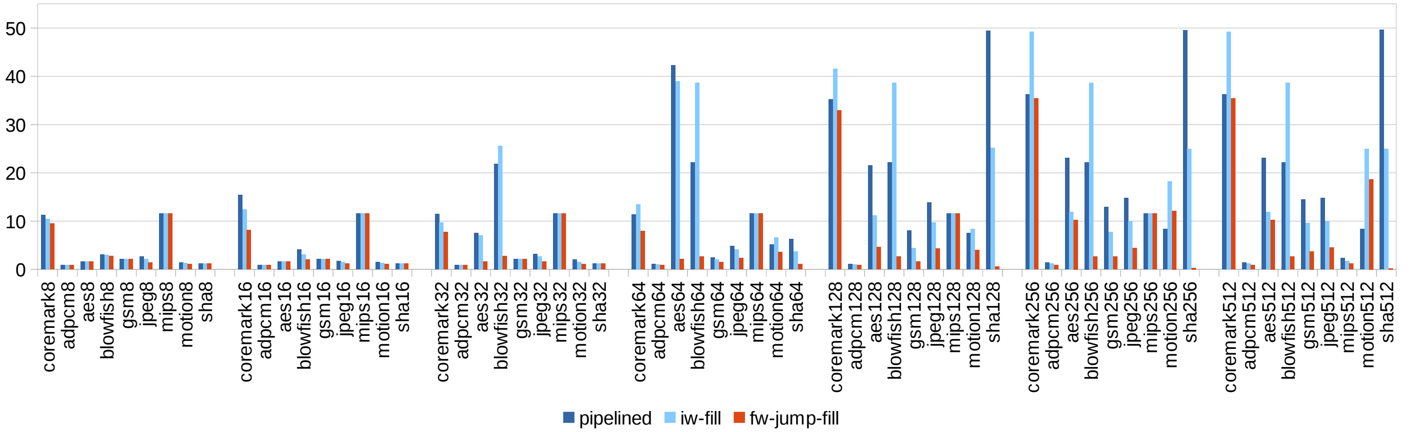

Fig. 6. Cycle count increase (%) compared to baseline across IRF variants and IRF size.

V. Evaluation

The processor core [5] used in the evaluation was designed with the *TTA-based Co-Design Environment* (TCE) [6] tools. The evaluation setup is illustrated in Fig. 1. The exposed datapath processor architecture utilizes a long instruction word control similar to *very long instruction word* (VLIW) architectures and thus benefits greatly from efficient low level instruction caching. The processor design used in the evaluation features a 50-bit instruction word with fields controlling each data bus. However, as the measurements do not relate to the datapath organization, the evaluation results can be generalized to other statically scheduled processor paradigms as well. The instruction memory cached by the IRF was implemented as a 8192x50-bit (51.2 kB) onchip SRAM. An overly large instruction memory would likely result in over-optimistic energy comparison in favor of the IRF, so the smallest 2^n-entry memory able to fit all the benchmarks was chosen.

To evaluate performance in control-oriented program code, EEMBC *Coremark* [7] was used. To emphasize another application domain of interest for IoT applications, we added eight fixed-point signal processing benchmarks from *CHStone* [8]. The benchmarks are characterized according to different instruction types in Table I. *Coremark* and *mips* have more program control instructions, whereas the other benchmarks are more straightforward, typical to signal processing. The benchmark programs were compiled using the compiler of the TCE toolset with the described compiler support implemented on top. Generation of the IRF instructions and hardware was integrated into the TCE tools and performed automatically. Hardware simulations were performed with Mentor ModelSim 10.4. Topographical synthesis results were obtained with Synopsys Design Compiler using a 28 nm FD-SOI process technology. SRAM timing, power and energy characteristics were obtained from Cacti-P [9].

To take into account the instruction storage access timings, conservative values of 0.40 ns access time for the data read port and 0.15 ns address hold time were manually inserted to the core's corresponding ports in Design Compiler. These values were obtained from Cacti-P for the 8192x50-bit memory.

A. Effect of IRF Fill Strategy on Cycle Count

Cycle counts normalized to the baseline are presented in Fig. 6. As expected, all the IRF variants incur a cycle count penalty compared to the baseline. In all but two cases, *motion* with IRF sizes 256 and 512, the *fw-jump-fill* incurs the least additional cycles. In these cases, as the IRF size is quite large, the compiler-analyzed prefill amount grows compared to those of smaller IRF sizes. As a result, the amount of prefill cycles of total execution time increases, but a branch in the benchmark causes the execution to exit from IRF, leaving most of the fetched instructions unused. In this case, the hardware presence assurance performs better, as the amount of stalls at the beginning of every IW in *fw-jump-fill* is far larger than the stalls incurred in the *pipelined* variant.

B. Area

Area comparison of the evaluation core with different IRF sizes is depicted in Fig. 7. On small IRF sizes, there is no notable differences in the core area. However, as the IRF size increases, the baseline and *pipelined* designs incur more area overhead, as the size of the presence bit status register increases. At IRF size of 512, *iw-fill* and *fw-jump-fill* both occupy 5% less area compared to the baseline.

C. Maximum Clock Frequency

In the baseline design, as the instruction fetch and decode stages are executed in the same pipeline stage, the next level instruction storage access timing becomes a part of the critical path. When the access time (data valid relative to clock edge) and the address setup time are large enough, the critical path

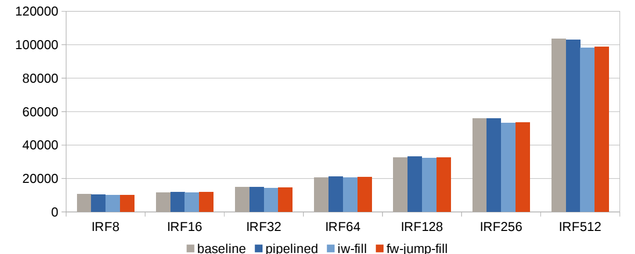

Fig. 7. Area (μm^2) comparison of the evaluation core with IRF variants.

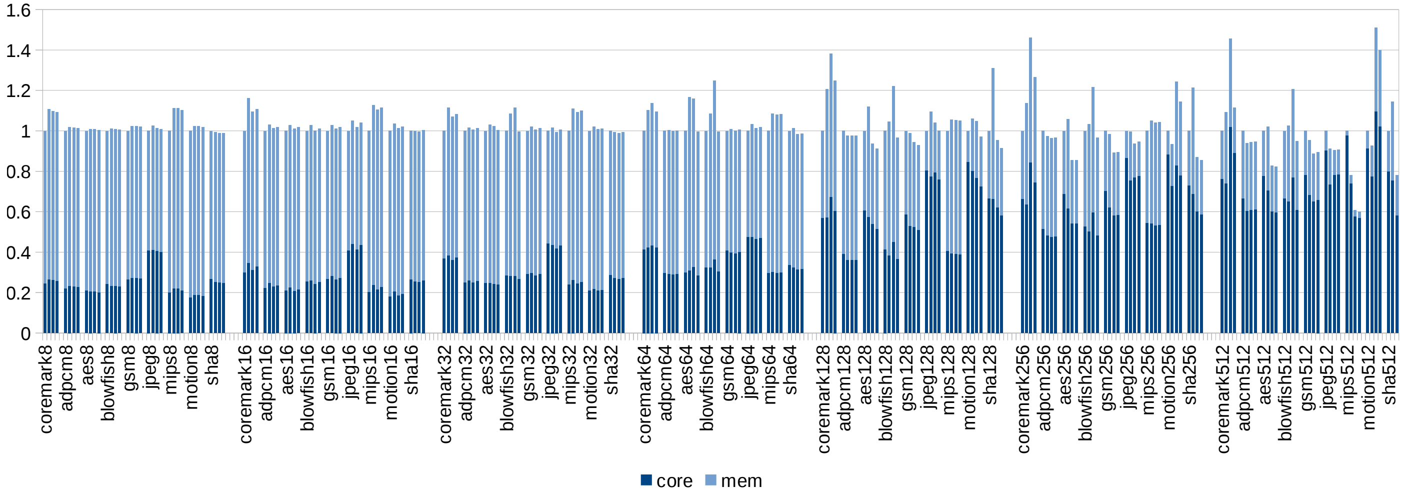

Fig. 8. Energy consumption compared to baseline across IRF sizes with instruction memory size of 51.2 kB. IRF variants per benchmark from left to right: baseline, *pipelined*, *iw-fill*, *fw-jump-fill*.

forms from the data read port through the instruction fetch and decode, looping back to the instruction address.

In the IRF variants with hardware presence assurance, with small next level storage access timings, the critical path goes through the data read port and the presence inspection logic to the global lock port. The global lock is connected to each function unit and is used to stall their execution in case of dynamic external events. Thus here the drawback of using a hardware presence assurance mechanism is clearly in decreased maximum operating clock frequency.

D. Energy-Delay Trade-Offs

Maximum clock frequencies after synthesis are listed in Table II. As the differences between the three variants were not large, the maximum critical path between the three variants at each IRF size (bold) was chosen as the operating frequency for evaluation. For the baseline, the differences were larger, so each IRF size for the baseline was evaluated with its own maximum clock frequency.

As indicated by Table II, the timing critical path of the baseline design is on average 33% compared to the IRF variants, due to the *instruction registers* added to the variants to cut the critical path. As seen in Fig. 8, although the baseline design executes all benchmarks in the least clock cycles, the energy consumption of the IRF variants is not significantly larger, 3-7%, on small IRF sizes. With IRF size of 512, the *fw-jump-fill* variant consumes 9% less energy compared to the baseline. This is due to not having the presence assurance logic. With *coremark* and *motion*, the variants consume more energy compared to the baseline because of preloaded instructions that do not get executed increasing the energy consumption. The best case with an IRF size of 512, *mips*, consumed 40% less energy compared to the baseline.

As the average energy consumption of the benchmarks at each IRF size is quite similar, it is interesting to calculate the *energy delay product* (EDP) to add weight to execution time of each benchmark in conjunction with the energy consumption. The EDP per benchmark for each IRF size is presented in Fig. 9. The cases with a higher EDP value compared to the baseline correspond to the relatively high energy consumption cases in Fig. 8. In almost all cases, the *fw-jump-fill* variant has the best EDP value. The best reduction, 47%, was achieved in *mips* with an IRF size of 512. The geometric mean over benchmarks is presented in Table III. The *fw-jump-fill* variant is the only one to achieve lower EDP value on each IRF size when averaged over the benchmark set.

VI. Related Work

Various IRF designs have been proposed in the past. In our work we picked various design choices we believe are interesting for a closer study and systematically evaluated them. Here we list the considered IRFs and their features.

IRFs were first proposed by Hines et al. [4]. They used a 32-entry IRF that was updated at the beginning of program execution and used in a dictionary fashion. Instructions referencing the IRF, containing 2 to 5 *packed* instructions, were fetched from cache and were used to address the IRF. From

TABLE II. Timing critical path (ns) after synthesis. Longest critical path between variants per IRF size in bold.

IRF size	baseline	pipelined	*iw-fill*	*fw-jump-fill*
8	1.00	**0.73**	0.71	0.72
16	1.09	0.76	0.76	**0.78**
32	1.09	0.80	**0.82**	0.80
64	1.10	**0.83**	0.82	0.82
128	1.19	**0.87**	**0.87**	0.86
256	1.16	**0.96**	0.90	0.94
512	1.23	**1.07**	0.90	0.99

TABLE III. Geometric mean of energy consumption/energy delay product compared to baseline.

IRF size	*pipelined* (%)	*iw-fill* (%)	*fw-jump-fill* (%)
8	104/79	103/78	103/78
16	105/79	103/77	104/77
32	104/83	103/83	102/80
64	106/89	107/91	102/80
128	109/94	105/89	99/77
256	104/103	104/102	99/88
512	97/100	97/100	91/86

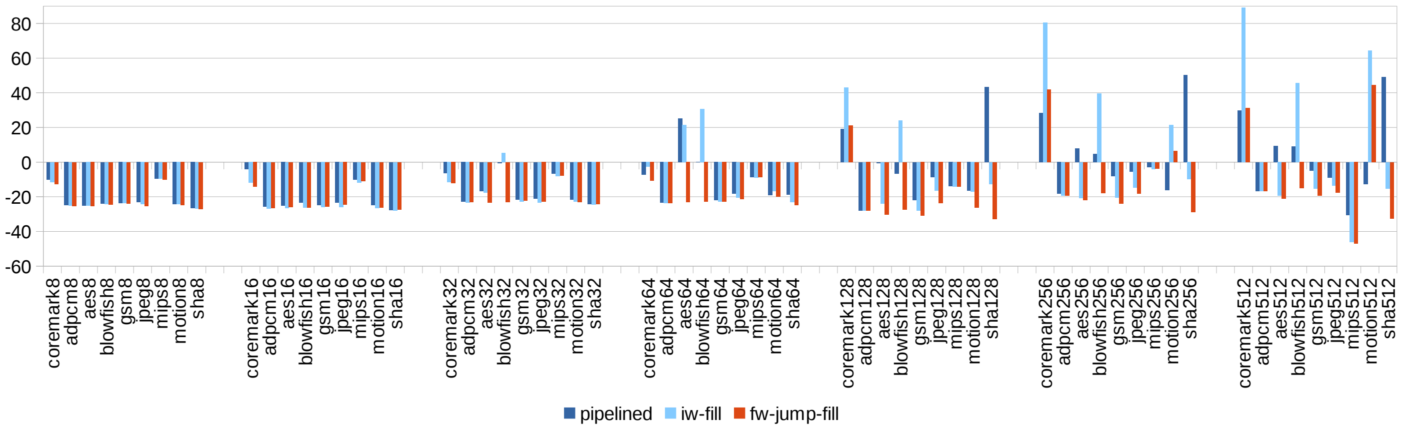

Fig. 9. Energy delay product compared to baseline across IRF sizes with instruction memory size of 51.2 kB.

this design, we evaluated executing instructions directly from the IRF or from the next level in hierarchy (bypassing).

Hines et al. later added compiler support to dynamically update the instruction register contents [10]. Also this approach is evaluated in this paper. The authors used dynamic program profiling to determine the instructions to be placed into registers and used register windows to indicate instructions that could simultaneously exist in the IRF. In our evaluation we used static (compiler-based) analysis of instructions without dynamic profiling.

Black-Schaffer et al. [11] added indirection in addressing the instruction register file in a VLIW processor. Each subfield of the instruction had an IRF, with a size customized to the instruction field. Each *control & index memory* (CIM) entry contained a complete VLIW instruction in the form of indices addressing the IRFs. All instructions were executed from the IRFs via the CIM and could not be bypassed. Simultaneous loading and execution of instruction was enabled with presence bits, an approach evaluated in the IRF variants of this paper.

VII. Conclusions

In this paper, we designed, implemented and systematically evaluated four instruction register files to evaluate the effect of fetch strategy and implementation to processor performance, energy consumption and area. Minimal branch delay and the least cycle counts in all benchmarks were achieved by having instruction fetch and decode in the same pipeline stage, at the expense of a low maximum clock frequency. Separating them into two stages increases the cycle counts, but allows for a 1.5x increase in clock frequency and results in better energy delay product.

In ensuring that instructions are fetched before their execution, hardware presence assurance limits the clock frequency at relatively large IRF sizes. Software presence validation mitigates the issue and consumes less overall area, but requires additional compiler analysis.

In the best case, careful compiler analysis along with optimized hardware implementation and software presence assurance achieved a total core energy saving of 9% on average with EEMBC Coremark and CHStone benchmarks when compared to the baseline instruction register file. Best individual benchmark reduction was 40%. Energy delay product on average was 23% better and in the best case 47% better compared to the baseline.

Future work involves studying further variations of IRF designs with, e.g., capabilities to prefetch code blocks ahead of time.

Acknowledgment

The authors thank the following sources of financial support: Tampere University of Technology Graduate School, Business Finland (FiDiPro Program funding decision 40142/14), HSA Foundation, the Academy of Finland (funding decision 297548) and ECSEL JU project FitOptiVis (project number 783162).

References

[1] O. Silven and K. Jyrkkä, "Observations on power-efficiency trends in mobile communication devices," *EURASIP Journal on Embedded Systems*, vol. 2007, no. 1, p. 056976, Mar. 2007.

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

[3] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 23-25 2010, pp. 21–21.

[4] S. Hines, J. Green, G. Tyson, and D. Whalley, "Improving program efficiency by packing instructions into registers," in *Proceedings of the Annual International Symposium on Computer Architecture*, Washington, DC, June 19-23 2005, pp. 260–271.

[5] J. Multanen, H. Kultala, P. Jääskeläinen, T. Viitanen, A. Tervo, and J. Takala, "Lotta: Energy-efficient processor for always-on applications," in *Proceedings of the International Workshop on Signal Processing Systems*, Cape Town, South Africa, Oct. 21-24 2018.

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

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

[8] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, Oct. 2009.

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

[10] S. Hines, G. Tyson, and D. Whalley, "Reducing instruction fetch cost by packing instructions into registerwindows," in *Proceedings of the International Symposium on Microarchitecture*, Barcelona, Spain, Nov. 12-16 2005, pp. 19–29.

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