A Hybrid Instruction Prefetching Mechanism for Ultra Low-Power Multicore Clusters

A Hybrid Instruction Prefetching Mechanism for Ultra Low-Power Multi-core Clusters

Maryam Payami, Erfan Azarkhish, Igor Loi, and Luca Benini, *Fellow, IEEE*

***Abstract*—The instruction memory hierarchy plays a critical role in performance and energy efficiency of ultra-low-power processors for the Internet-of-Things (IoT) end-nodes. This is mainly due to the extremely tight power envelope and area budgets, which imply small instruction-caches (I-Cache) operating at very low supply voltages (near-threshold). The challenge is aggravated by the fact that multiple processors, fetching in parallel, require plenty of bandwidth from the I-Caches. In this paper, we propose a low-cost and energy efficient hybrid instruction-prefetching mechanism to be integrated with an Ultra-Low-Power (ULP) multi-core cluster. We study its performance for a wide range of IoT applications, from cryptography to computer vision, and show that it can effectively improve the hit-rate of almost all of them to above 95% (average performance improvement of over 2×). In addition, we designed our prefetcher and integrated it in a 4-cores cluster in 28nm FDSOI technology. We show that system's power consumption increases only by about 11% and silicon area by less than 1%. Altogether, a total energy reduction of 1.9× is achieved, thanks to more than 2× performance improvement, enabling a significantly longer battery life.**

***Index Terms*—Ultra Low-power Embedded Multi-cores, Energy Efficiency, Instruction Cache, Instruction Prefetching**

I. Introduction

Prefetching is an extensively studied technique for data-memory latency hiding in high-performance processors. Yet, instruction prefetching is not as critical in such systems for two reasons. First, the data working-set in large-scale applications is much larger than the code working-set. Second, the code is cache-friendly, so a moderately sized instruction-cache (I-Cache) achieves very high hit-rates, and architects rightfully focus on the main bottleneck which is data caching [1]. In parallel ULP processors for the Internet-of-Things (IoT) end-nodes, the situation is reverted: data working sets are smaller and highly amenable to tiling, double buffering, and DMA management, since most workloads are linked to near-sensor data processing [2][3]. On the other hand, the on-chip area budget is much tighter in IoT end-nodes. This squeezes the size of the I-Caches, also because they are often implemented with low-density memories which can operate at low voltages (e.g. standard-cell based caches [4][5]). This area squeeze is further exacerbated by the fact that there are multiple processors fetching in parallel, so the instruction memory system must deliver plenty of bandwidth, requiring complex logic such as L0-buffers [6] and broadcast-interconnect [5], which further increase the area dedicated to the instruction memory. Therefore, in this context it is highly valuable in term of area and energy-efficiency to keep the I-Cache small, while at the same time exploit the latency-hiding opportunities offered by prefetching to reduce the impact of cache-misses [7]. Another factor is that the speed gap with the background on-chip L2-memory (dense SRAM or even embedded flash), where the entire code is stored, is not so huge, so the amount of prefetched information can be kept small (a direct consequence of Little's law [2][5]). This also works in the direction of making instruction prefetching highly effective.

Looking at the state-of-the-art instruction prefetching techniques, a simple yet effective method is called next-line prefetching (NLP) [1]. In this scheme, when a cache line is fetched, a prefetch for the next sequential line(s) is also initiated. One way to do this is called prefetch-on-miss. NLP exploits spatial locality for sequential access patterns, but for conditional/unconditional branches and function calls it is ineffective [8]. The Stream prefetcher (STP), similarly, follows a pattern or rule. As long as it persists, it issues a stream of prefetch requests [8]. Target-line [1] tries to prefetch non-sequential cache lines by maintaining a target prefetch table in hardware. Combining this method with NLP can take advantage of both mechanisms for sequential/non-sequential code. The main disadvantage of this method is a significant hardware cost for the table and the associated logic to perform the table lookups and updates. Also, the effective-address of the control-instructions need to be known even before their execution. This can be very costly in terms of logic delay and area. Plus, the compulsory misses do not benefit from it [1].

A Markov Prefetcher [8] monitors the address stream and builds a Markov prediction table in hardware. The learning phase of this scheme is usually long and its hardware cost is significant. A wrong-path prefetcher combines the target and next-line mechanisms, with the major difference that no target-line addresses are saved and no attempt is made to prefetch only the correct execution path [1]. Instead, in the simplest case both paths of conditional branches are always prefetched. This method can perform a potentially useful prefetch for a branch which is not taken, if the execution returns to it in the near future and it is then taken. No extra hardware is required above NLP, but cache pollution and the large amount of extra traffic are problematic.

Lastly, software-prefetching (SWP) [1] is based on the use of some form of explicit fetch instructions with very little added hardware. The only difficulty lies in correct placement of the fetch instructions in the code (also known as prefetch scheduling): It is possible to gain significant advantages by inserting a few fetch instructions manually in strategic portions

M. Payami, E. Azarkhish, and I. Loi are with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy (e-mails: maryam.payami@studio.unibo.it, {erfan.azarkhish, igor.loi}@unibo.it).

L. Benini is with the Department of Information Technology and Electrical Engineering, Swiss Federal Institute of Technology Zurich, 8092 Zurich, Switzerland, and also with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy (e-mail: lbenini@iis.ee.ethz.ch).

This project has received funding from the European Unions Horizon 2020 research and innovation program for the Oprecomp project (GA No. 732631).

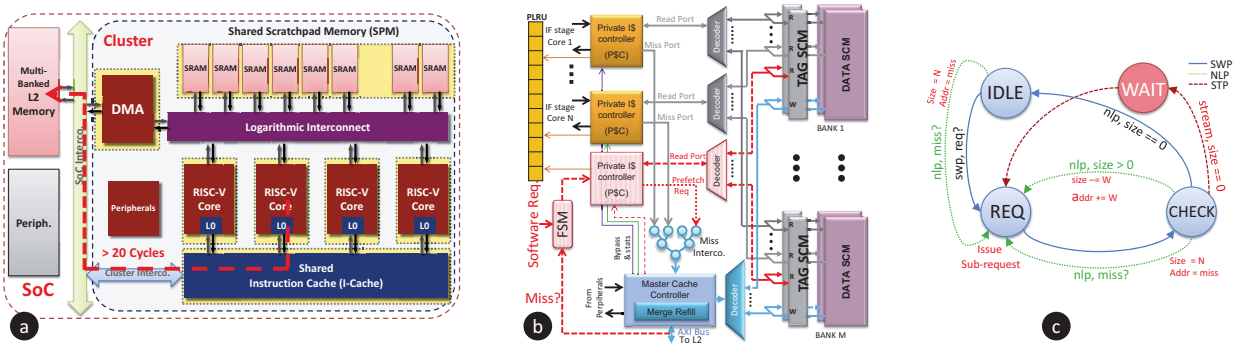


Fig. 1. (a) A processing cluster in the PULP platform, (b) block diagram of the multi-ported shared I-Cache in PULP, along with the prefetching augmentations, (c) The FSM for the proposed hybrid prefetching scheme.

of the program [9]. Also, this technique allows for prefetching complex access patterns and function calls.

In this paper we propose a hybrid-prefetching mechanism as a combination of 3 low-cost schemes (NLP, SWP, and STP), targeting tightly coupled multi-core clusters with simple in-order cores which are suitable for IoT end-nodes. Furthermore, a priority-based selection mechanism among the three schemes is implemented to ensure reactiveness and to avoid prefetching stale and useless information. We perform an extensive experimental validation with various classes of benchmarks along with an exploration of different approaches. Methodology and results are presented in Section II and Section III. Conclusions are drawn in Section IV. Further experiments and explanations can also be found in [10].

II. Methodology and Design

To design an efficient instruction prefetcher, it is extremely important to first study the typical IoT endnode workloads [11]. Sixteen representative micro-benchmarks from four categories (Cryptography, Numerical, Artificial Intelligence, and Image Processing) are selected for analysis: *sha1*, *md5*, and *whirlpool* are cryptographic hash functions; *aes* is a symmetric encryption algorithm; *lu* performs LU decomposition on matrices; *matrixmult* and *strassen* perform matrix multiplication; *sobel* is an edge detection operator; *cnnconv* is the convolution operator in convolutional neural networks; *srad* performs speckle reducing anisotropic diffusion; *neuralnet* is a feed-forward network with 3 hidden layers with floating-point arithmetic; *fft* computes the fast-fourier-transform; *svm* is a fixed-point implementation of support-vector-machines; *fast* is an efficient corner detection algorithm; finally *singleloop* and *multifunc* are two synthetic benchmarks with a loop of NOP instructions and with multiple function calls in a loop, respectively. We analyze these benchmarks in terms of instruction memory access patterns and roughly categorize them into two groups: Group-1 (*sha1, aes, md5, lu, matrixmult, strassen, whirlpool, sobel, cnnconv, singleloop, multifunc*) have large loop bodies either naturally or through the use of loop unrolling and function inlining for improved performance and compiler scheduling (The unrolled codes will be identified by "*_u*" suffix). This group generate a lot of capacity misses in the cache. Group-2 (*srad, neuralnet, fft, svm, fast*) have many function calls, mostly resulting in conflict misses in the cache. External link libraries and software emulation of floating-point/fixed-point arithmetic are examples which can cause this behavior. We execute the benchmarks on the platform individually, and multi-programmed workloads are out of the scope of this paper. By analyzing these representative instruction streams, we propose a hybrid mechanism based on next-line prefetching (NLP), stream prefetching (STP), and software prefetching (SWP). We demonstrate that for Group-1 a combination of NLP+STP can effectively remove most of the misses thanks to their large loop bodies. But for Group-2, software assistance is required to achieve high hit-rates. Therefore, a combination of NLP+SWP is proposed with software prefetch requests explicitly inserted in the code. This is particularly beneficial for function calls, because their code can be brought into the cache right before execution with a prefetch request inserted slightly before each call. It is worth mentioning that cold-misses can be easily eliminated as well by insertion of an SWP command right before the main loops [10]. In our design, the prefetched blocks are stored inside the cache, eliminating the need for a separate storage, and prefetched lines are treated differently from normal cache-lines to reduce the risk of cache pollution, as explained later.

We integrate our prefetcher in the PULP platform [3], an ultra low-power multi-core targeting energy-efficient computation for the IoT applications. As shown in Figure 1a, each processing cluster in PULP includes a set of light-weight RISC-V based cores with 4 pipeline stages and in-order execution (no branch prediction, predication, or multiple issue is supported) for energy efficient operation. The processors share a Scratchpad Memory (SPM) for computation and a DMA engine for bulk transfers.

A multi-ported I-Cache is shared among these cores, enabling flexible execution of different code sizes, as opposed to private caching mechanisms [5]. Figure 1b illustrates an overview of this shared I-Cache which is based on private cache-controllers (P$C) with shared data and tag banks, flexibly designed based on standard-cell memories (SCM) with controlled placement [4]. Cache hits are independently handled by the P$Cs in a single cycle, while miss requests from multiple P$Cs are redirected to a master cache controller, with the possibility to be merged if they target the same cache line. A cache miss might take several cycles as the L2 memory is situated out of the clusters (See Figure 1a). The replacement policy is pseudo-least-recently-used (PLRU), and associativity is a generic parameter. A more detailed description of this cache architecture is presented in [5].

Integrating our proposed prefetching techniques requires

minor modifications to the baseline I-Cache illustrated in Figure 1b: a small finite-state-machine (FSM), a new P$C, and one port only to the tag array (all highlighted in red color). When a software prefetch request arrives at this FSM, depending on its size, it is issued to the corresponding P$C line-by-line (each line is 16Bytes). If another prefetch request arrives while serving a request, the FSM drops the current request and starts the new one. This mechanism is therefore "preemptive" to avoid issuing stale requests causing cache pollution. The block diagram of this FSM is shown in Figure 1c. The newly added P$C consults the tag array before issuing any request and ignores prefetch requests to the already existing blocks. Plus, if no free ways are available, it looks up the LRU table (shown in Figure 1b) to choose a victim block for replacement. A prefetch request is then sent to the master controller through the miss-interconnect shown in the same figure. Unlike the processors, the prefetcher does not use the prefetched blocks, so it only needs a read-port to the tag banks. This also simplifies its P$C state-machine. Each SWP command requires two processor instructions for setting the address and size of the prefetch. This can be achieved through ISA extension of the processors, however, in this work we have added two memory mapped registers to the processor, writing to which triggers the prefetch FSM. This is easier to achieve and does not need compiler support. One interesting benefit of the PLRU replacement appears when used with the prefetcher. Prefetched blocks can be treated differently from normal blocks, by not updating their LRU counter when they are brought by the prefetcher. This way prefetched blocks are more prone to being replaced than the normal blocks and prefetcher induced pollution can be controlled.

A next-line prefetcher (NLP) can be easily integrated, as well, by a small modification to the FSM, illustrated in the Figure 1c. The AXI interconnect to L2 memory is constantly monitored and if a miss is observed there, N next lines starting from that miss address are prefetched. The NLP is extended to a stream prefetcher (STP) by inserting a wait-state in the FSM in Figure 1c. After completion of one burst request, a new prefetch request is automatically started after waiting for a specific number of cycles. With proper choice of a parameter called wait-cycles (the number of cycles to remain in the wait state) it is able to perform prefetch-before-miss. This is explored in Section III. To summarize, the single FSM illustrated in Figure 1c combines the three proposed schemes with different priorities and preemption. SWP has the highest priority and STP has the lowest. This is to ensure reactiveness and to avoid prefetching stale and useless information. NLP and STP can be enabled/disabled at run-time with their parameters (burst-size, wait-cycles) adjusted through dedicated configuration registers. This can be achieved either by the user program itself (requiring code modification), or using an additional runtime, allowing user programs to run unmodified. SWP, however, always requires code modification with prefetch request commands explicitly inserted in it.

III. Experimental Results

Our baseline scenario consists of the cycle-accurate RTL model of PULP with one processing cluster, 512 KB of L2 memory, and the required peripheral devices. All program codes are stored in L2 and it takes at least 20-cycles for a core to access it through the path highlighted in Figure 1a. Since L2 is shared by many cores and also used for data, its access time can increase further, unpredictably. The processing cluster has 4 RISC-V cores, 1 KB of 2-way set-associative I-Cache with PLRU replacement, and 8 SPM banks of 2 KB, and the required interconnects and peripherals. A small cache configuration is chosen with the main goal to optimize area and energy efficiency compared to large caches without prefetching. Performance has been measured in cycle-accurate simulation of the PULP platform executing complete benchmarks in QuestaSim 10.2c. Logic synthesis has been performed in Synopsys Design Compiler J-2014.09-SP4 in topographical mode using the 28nm-FDSOI technology by STMicroelectronics at 125°C, 0.9V, Slow-Slow process corner. A clock frequency of more than 500MHz was achieved, not limited by the I-Cache or the prefetcher (the critical-path was in the data interface of the cores towards the shared SPM). Power consumption was extracted at the 25°C, 0.9V, Typ-Typ corner using Synopsys Primetime H-2013.06-SP2.

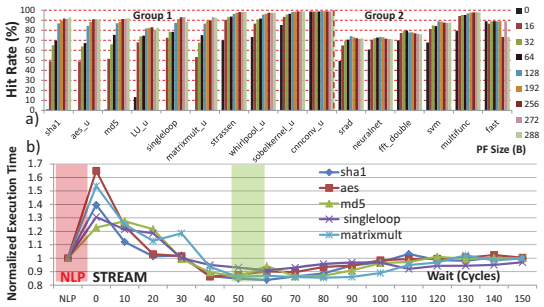


Fig. 2. a) Effect of the burst size (B) of NLP on the average hit-rates, b) The effect of STP wait-cycles on the execution time of different benchmarks.

First, the burst size of the NLP mechanism is changed from 0 (disabled) to 288 Bytes. The average hit-rate is plotted in Figure 2a. For Group-1, on the average, 65% improvement in hit-rate is achieved with the cryptography benchmarks (*sha1*, *md5*, *aes*) gaining the highest benefit. For Group-2, however, only 15% improvement is obtained at the best case. The *fast* kernel also does not gain any benefit because of having too many branches. It is, therefore, better to disable the prefetcher for it. Total execution-time was found to improve by over 2× for Group-1 and only by 1.2× for Group-2. Another glance at Figure 2a reveals that for some benchmarks in Group-1, there is still some room for improvement, and hit-rate is not yet close to 100%. This is because of the "prefetch-on-miss" strategy which triggers the prefetcher only when a miss has already happened. In this case it is possible to enable the STP and adjust its wait-cycles to completely remove the unnecessary misses. The effect of wait-cycles on STP's performance (combined with NLP with a typical burst size of 256 Bytes) is illustrated in Figure 2b. As can be seen, with zero wait cycles, the STP works even worse than NLP due to over-prefetching and polluting the cache, however with wait-cycles around 50 to 60, hit-rate is improved to over 95% and execution-time is further reduced.

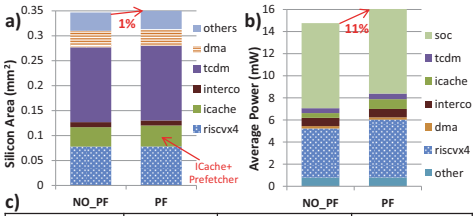
For Group-2, on the other hand, a combination of SWP and NLP was found effective. Software prefetch requests were manually inserted in the source code, before the function calls. The code size increase due this operation was less than 11% in

the benchmarks. Moreover, NLP was enabled to aid SWP with prefetching the sequential code sections. This way both the original sequential misses and most of the additional misses due to the code size increase can be eliminated. Please note that the manually inserted SWP instructions only increase the size of the sequential blocks, and these blocks are perfectly prefetched by NLP. More than 91% hit-rate was achieved with total execution only 8% higher than a 16 KB warm cache with 100% hits. This is interesting given that the area of the 16 KB cache is approximately 16× larger (2.8× larger cluster).

Effect of the hybrid prefetcher on silicon area and power consumption is illustrated in Figure 3. As can be seen, the cluster area is increased by less than 1% (mainly due to the FSM and the additional read port to the tag array), and the total power consumption increase in the system (including the L2 memory and the peripherals) is around 11%, on the average. The power increase mostly occurs in the processors and the L2 memory, because of fewer stall cycles and the increased L2 traffic. Total energy reduction and performance improvement are shown in Figure 4, for each individual benchmark at its best prefetching configuration. On the average, energy-efficiency is improved by 1.9×, thanks to the average improvement of 2.1× in performance and very marginal increase in power consumption. This enables a significantly longer battery-life in energy constrained systems. Note that enlarging the cache is not as energy- and area-efficient as prefetching. An 8 KB cache without prefetching achieves similar performance as a 1 KB cache with prefetching, with 29% more power consumption and 2× larger cluster area. Also, further increase in cache size (to 2 KB) in presence of the prefetcher was found unnecessary, as performance only improves by less than 5% with more than 10% increase in total power (See Figure 3c). Another observation is that if we unroll *strassen*, its baseline performance will improve by 2.3×, and if we use prefetching with this unrolled version, another 35% improvement is achievable (less than 10% power consumption increase). This highlights that even though unrolling increases the code size it has a very positive impact on compiler scheduling, and if augmented with a proper prefetching mechanism it can be very beneficial for energy efficiency. To sum up, our proposed scheme allows simple in-order processors to issue multiple outstanding instruction fetch transactions towards the L2 memory, and provides them with latency hiding capabilities, similar to out-of-order processors with large issue widths. This way, stall-cycles are reduced and energy efficiency is improved to a great extent [10].

IV. Conclusions

In this paper we proposed a low-cost hybrid instruction prefetching mechanism based on SWP, NLP, and STP to be integrated with ultra low-power multi-core platforms with simple in-order cores. We studied a wide range of applications and grouped them into two categories: the ones with large computation loops, and the ones with many function calls. We showed that for most of the benchmarks in the first group a cooperation between NLP and STP can improve the ICache hit-rate to over 95% with an average performance improvement of over 2×. While for the second group, SWP+NLP can be effective, leading to a similar improvement. By synthesizing our designs using the state-of-the-art technologies we showed



c)

I-Cache Size (KB)	Prefetching?	Cluster Area (mm²)	SoC Power (mW)
1	Yes	0.35	16.5
2	Yes	0.39	18.3
8	-	0.66	21.3
16	-	0.98	25.8

Fig. 3. (a) Silicon area (mm^2) of the cluster (b) system power consumption, breakdown with/without the prefetcher. (c) Synthesis area and power consumption for cycle time of 2 ns.

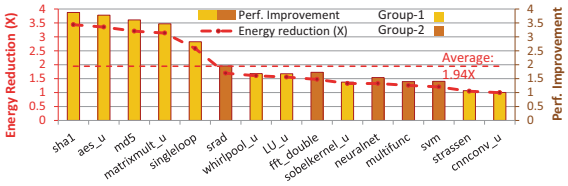


Fig. 4. Total energy reduction (left-axis) and performance improvement (right-axis) due to prefetching, sorted in decreasing order of energy reduction.

that the increase in system's power (about 11%) and cluster's silicon area (less than 1%) are negligible. Overall, our proposed prefetching scheme allows for an average energy reduction of 1.9× over the range of studied applications. The future directions include studying the effectiveness of the proposed schemes for larger (IoT gateway) system configurations, and complex parallel workloads including operating systems and virtual memory. We will also study compiler modifications to automatically insert software prefetch commands in the code.

References

[1] B. Falsafi and T. F. Wenisch, *A Primer on Hardware Prefetching*, M. Martonosi, Ed. Morgan & Claypool, 2014.

[2] S. Pei, M. S. Kim, and J. L. Gaudiot, "Extending amdahl's law for heterogeneous multicore processor with consideration of the overhead of data preparation," *IEEE ESL*, vol. 8, no. 1, pp. 26–29, March 2016.

[3] D. Rossi, F. Conti, A. Marongiu *et al.*, "PULP: A parallel ultra low power platform for next generation IoT applications," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015, pp. 1–39.

[4] A. Teman, D. Rossi, P. Meinerzhagen *et al.*, "Power, area, and performance optimization of standard cell memory arrays through controlled placement," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 4, pp. 59:1–59:25, May 2016.

[5] I. Loi, D. Rossi, G. Haugou *et al.*, "Exploring multi-banked shared-L1 program cache on ultra-low power, tightly coupled processor clusters," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, ser. CF '15, 2015, pp. 64:1–64:8.

[6] M. Gautschi, P. D. Schiavone, A. Traber *et al.*, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on VLSI Systems*, vol. PP, no. 99, pp. 1–14, 2017.

[7] A. Panda and K. S. Chatha, "An embedded architecture for energy-efficient stream computing," *IEEE ESL*, vol. 6, no. 3, pp. 57–60, 2014.

[8] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, vol. 49, no. 2, pp. 35:1–35:35, Aug. 2016.

[9] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 2:1–2:29, Mar. 2012.

[10] M. Payami, "Instruction prefetching techniques for ultra low-power multicore architectures," Master's thesis, Univ. of Bologna, Bologna, 2016.

[11] J. Pallister, S. J. Hollis, and J. Bennett, "Identifying compiler options to minimize energy consumption for embedded platforms," *The Computer Journal*, vol. 58, no. 1, pp. 95–109, 2015.