HIGH PERFORMANCE AND ENERGY-EFFICIENT INSTRUCTION CACHE DESIGN AND OPTIMISATION FOR ULTRA-LOW-POWER MULTI-CORE CLUSTERS

Presentata da:

Jie Chen

Relatore:

Chiar.mo Prof. Davide Rossi

Esame finale anno 2021

Thanks to the support of my company GreenWaves Technologies and the help of my professors and my wife.

High performance and Energy-efficient Instruction Cache Design and Optimization for Ultra-low-power Multi-core clusters

Jie Chen

Department of Electrical, Electronic and Information Engineering

University of Bologna

A thesis submitted for the degree of

Doctor of Philosophy in

Electronics, Telecommunications and Information Technologies Engineering

2021

Abstract

High Energy efficiency and high performance are the key regiments for Internet of Things (IoT) end-nodes. Exploiting cluster of multiple programmable processors has recently emerged as a suitable solution to address this challenge. However, one of the main bottlenecks for multi-core architectures is the instruction cache. While private caches fall into data replication and wasting area, fully shared caches lack scalability and form a bottleneck for the operating frequency. Hence we propose a hybrid solution where a larger shared cache (L1.5) is shared by multiple cores connected through a low-latency interconnect to small private caches (L1). However, it is still limited by large capacity miss with a small L1. Thus, we propose a sequential prefetch from L1 to L1.5 to improve the performance with little area overhead. Moreover, we optimized the core instruction fetch stage with non-blocking transfer by adopting a 4×32-bit ring buffer FIFO and adding a pipeline for the conditional branch to cut the critical path for better timing. We present a detailed comparison of different instruction cache architectures' performance and energy efficiency recently proposed for Parallel Ultra-Low-Power clusters. On average, when executing a set of real-life IoT applications, our two-level cache improves the performance by up to 20% and loses 7% energy efficiency with respect to the private cache. Compared to a shared cache system, it improves performance by up to 17% and keeps the same energy efficiency. In the end, up to 20% timing (maximum frequency) improvement and software control enable the two-level instruction cache with prefetch adapt to various battery-powered usage cases to balance high performance and energy efficiency.

Contents

Chapter 1

Introduction

In recent years, with the growing Internet of Things (IoT) markets for smart homes, smart cities remote control and monitoring nodes raising rapidly, the requirement for near-sensor processing devices with high performance and low power is more vigorous, especially when powerful embedded systems and machine learning are involved in this field. The traditional applications, such as thermostats, water meter monitoring, have less necessity for high performance and are often based on the ultra-low-power Micro Controller Unit (MCU) powered by the battery. While for other camera or audio based applications, such as home security, office people counting, voice control, etc, the complex signal processing algorithms require high performance and much larger power. As a result, these devices are always connected to the source, bringing more expense for installation and implementation from the commercial point of view. Thus, to save the power and the implementation fee in commercial practice, designing the high-performance and ultra-low-power IoT signal processing devices with a long battery lifetime becomes our goal.

In general, most of the IoT devices could be divided into three categories, i) the ultra-low-power MCU, receiving and transmitting the data from sensors to the sever with little computing capacity, mainly for controlling and monitoring some specific environments, running with Real-Time Operating System (RTOS); ii) the powerful general embedded multi-core SoC used in real-time camera surveillance powered by the source; iii) Digital Signal Processing (DSP) devices with accelerators, aiming at specific usage cases to save power, such as headphones and earbuds with Audio Noise

Control (ANC). Between the two latter categories, we consider that there could be a usage case that needs high performance and could be powered by the battery. That is a non-real-time usage case with fast response, for example, a smart locker with face recognition powered by the battery. Since the face image detection occurs several times a day, the devices are in sleep most of the time. Thus we concentrate our design on these low-frequency, high-performance usage cases.

To chase this market, the MCUs are apparently not suitable with poor performance. The powerful Linux based embedded multi-core SoCs with various accelerators are capable of fast computation. However, it is difficult for them to control the working power envelope below 10mW and the price including implementation fee. Besides, the specific usage cases limit DSPs' markets. One possible solution to extend the battery lifetime with relatively high performance is using an optimized (simplified) general multi-core platform with specified accelerators to maximize the performance both from instruction-level and thread-level parallelism. Thus, the first version of Parallel Ultra Low Power platform (PULP) is proposed based on Near-Threshold Computing (NTC) [46], which means that it is better to have N $\times$ simple cores running at a lower voltage than one core running at nominal voltage. The PULPv1 achieves 60 GOPS/W at 0.5 V in ST Microelectronics UTBB FDSOI 28nm technologies [92].

During the continued optimization of PULPv1, there is one power issue of the instruction Cache (iCache) in multi-core cluster when running OpenMP [78] based applications. The iCache is based on Static Random-Access Memory (SRAM) and consumes more than 50% of the cluster's power. Thus, it is necessary to reduce the iCache's power. Since the cache memory is only a few KB (4KB), in PULPv2, we propose to replace the SRAM with latched-based Standard Cell Memory (SCM) with controlled Placement and Route (P & R) [107]. Thanks to the 2 - 4$\times$ less read and write energy compared to the SRAM, the iCache's power occupation in cluster reduces to about 25% and the energy efficiency reaches to 135 GOPS/W [95]. However, even though the private iCache is simple and fast with only one cycle access latency, its relatively small cache capacity and the data duplication degrades the performance and energy efficiency. To increase the cache capacity seen by each core, shared iCaches are proposed to decrease the miss rate [67], the single-port

(SP) iCahce and multi-port (MP) iCache. The former uses low latency (one cycle) interconnect to ensure each core can access all the memory banks. Nevertheless, congestion exists when several cores access the same cache bank, leading to extra power consumption. Besides, the logarithmic interconnect increases the critical path and limits the operating frequency. The latter iCache based on multi-port memories is proposed to tackle the congestion issue. However, the multi-port memories generate large extra area and power. Moreover, it still has limited operating frequency because of large ports congestion in P & R. As a result, the MP iCache is only suitable for small cache capacity with a few KB. In the end, the shared iCaches have worse scalability than the private cache when the number of cores increases.

As mentioned above, continuous performance and power optimization are the key factors in the multi-core PULP cluster to adapt to the battery-powered, low-frequency, high-performance IoT market. The approach proposed in this thesis to match all these requirements is to leverage hierarchy iCache design. Hierarchy iCache uses different levels to solve different above issues. In practice, the upper-level cache L1 is implemented as private, and the lower-level caches are implemented as shared [56]. This design provides high access rates for the high-level caches and low miss rate for the lower-level caches. This ideal was proposed to reduce the speed mismatch between the core and the main memory. In this research, we try to follow this typical design method and adapt it to our ultra-low-power, high-performance design requirement. Besides, hardware prefetching in the L1 cache is explored to improve the performance continuously. Finally, a practical Static Timing Analysis (STA) is performed to help improve the timing in iCache by removing the critical path to balance the high performance and low power in multi-core PULP clusters with good scalability.

In this scenario, it is necessary to launch a detailed comparison for each iCache design in terms of area, performance, power to address the work in this thesis. In order to analyze these characteristics, a based-line Register-Transfer Level (RTL) of PULP cluster with private cache is required to enable the exploration over various real-life IoT applications. Besides, the physical implementation is done for each cluster featuring different iCaches. A specific target of this thesis is to evaluate the iCache architecture's overall influence on the performance and energy for the multi-core cluster, analyzing the trade-offs between performance, energy and increasing the scalability in the multi-core ultra-low-power platform.

1.1 Background

1.1.1 Multi-core architectures

Scaling performance by increasing clock frequency and instruction throughput of single-core have been proven to be not viable anymore due to the power wall and instruction-level parallelism (ILP) wall [40]. Besides, many applications are better suited to thread-level parallelism (TLP) methods, and multiple independent cores are commonly employed to increase a system's overall TLP. A combination of increased available space (due to refined manufacturing processes) and the demand for increased TLP led to the development of multi-core architectures. As a result, the trend changes from pushing a single complex processor to integrating several computing elements into a single integrated circuit die happened decades ago for general-purpose computing, high-performance and also the embedded world (Fig. 1.1).

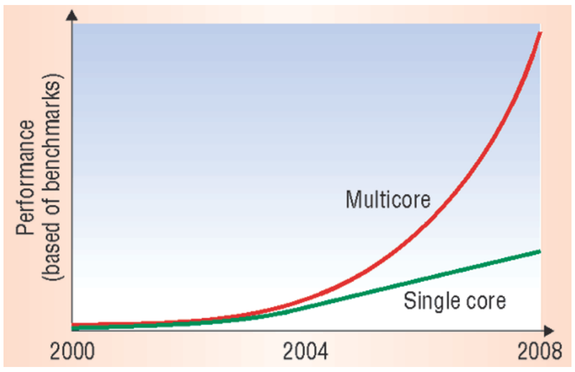


Figure 1.1: Performance comparison between a single core and multi-core processor [36].

Thanks to the Moore's Law, with a hundred billion transistors now available, even multi-core features with complex cores arise. Intel's Xeon Broadwell, consisting of 3.2 - 7.2 billion transistors, provides up to 22 cores (up from 8 in Xeon Sandy Bridge). Each core has a very aggressively complex 8-issue design (up from 6-issue in Sandy Bridge). AMD EPYC server solution employs a CPU with 4.8 billion transistors to provide 24 cores with two-way SMP support. Given the multi-core performance-per-area efficiency of small cores and the maximum outright single-threaded performance of large cores, ARM combines the two different cores to create asymmetric architecture. A strategy called ”big.LITTLE”, using several large cores paired with a few smaller simpler low-power cores, aims to decrease power consumption and prolong battery life instead of maximum multi-core performance for the phone or tablet.

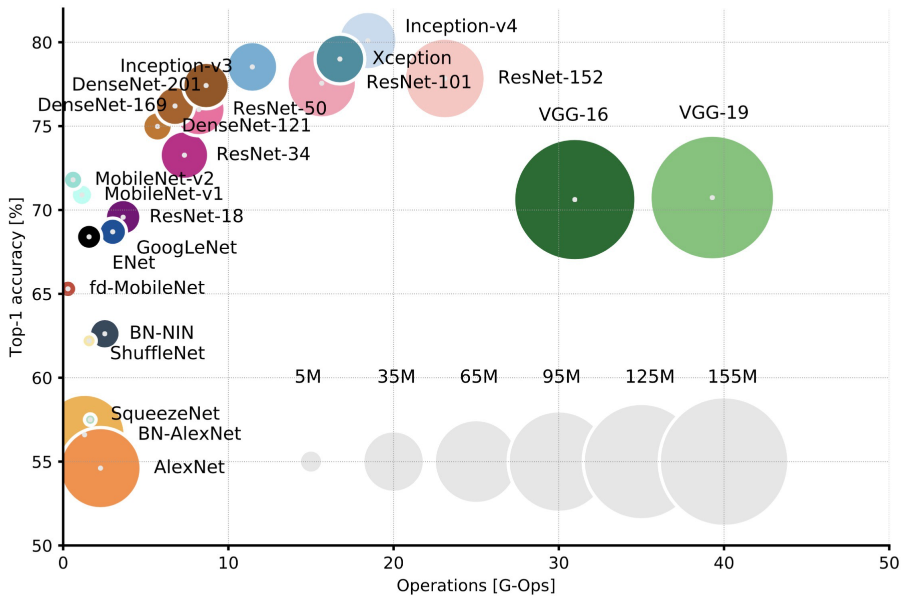


Figure 1.2: Top1 vs. operations, and size parameters. Top-1 one-crop accuracy versus the number of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters; a legend is reported in the bottom right corner, spanning from 5×10^6 to 155×10^6 parameters. Both these figures share the same y-axis, and the grey dots highlight the center of the blobs [14].

With the exploration of high-level (coarse-grained) parallelism, multi-core architecture can bring outstanding performance with the same power or get the same performance with less power, or a combination of both. Furthermore, by extending accelerator for specific applications, such as encryption and decryption engine, Graphic Processing Unit (GPU), and video coding engine, a heterogeneous platform can be designed to help unlock the energy efficiency potential, which is kept inaccessible by the burden of the Von Neumann fetch-decode-execute loop. However, the exploration of standard homogeneous multi-core baseline architecture to improve the performance and energy efficiency never stops. With the trends of exploiting machine learning algorithms for dedicated applications, more and more optimized efficient models are proposed. Some of the top-rated optimized models are shown in Fig. 1.2, from the result of the accuracy versus the number of operations, the most exciting part resides in top-left, which requires less performance ($\leq$ 10 G-Ops) while keeping the high accuracy ($\geq$ 75%). Besides, (Deep Neural Network) DNN relies on massive parallel multiply-and-accumulate (MAC) operations. It is suitable for multi-core low-end inference platforms both from performance and power consumption.

1.1.2 Low-power multi-core architecture

Multi-core with a simple core seems to be the best choice for low-power embedded end-nodes devices with relatively high performance, and for the parallelism maximization for intensive data analysis and for neural network inference applications. In the following subsections, two notable related works are described to highlight the low-power and high-performance characteristics of such architecture.

1.1.2.1 STMicroelectronics Platform 2012

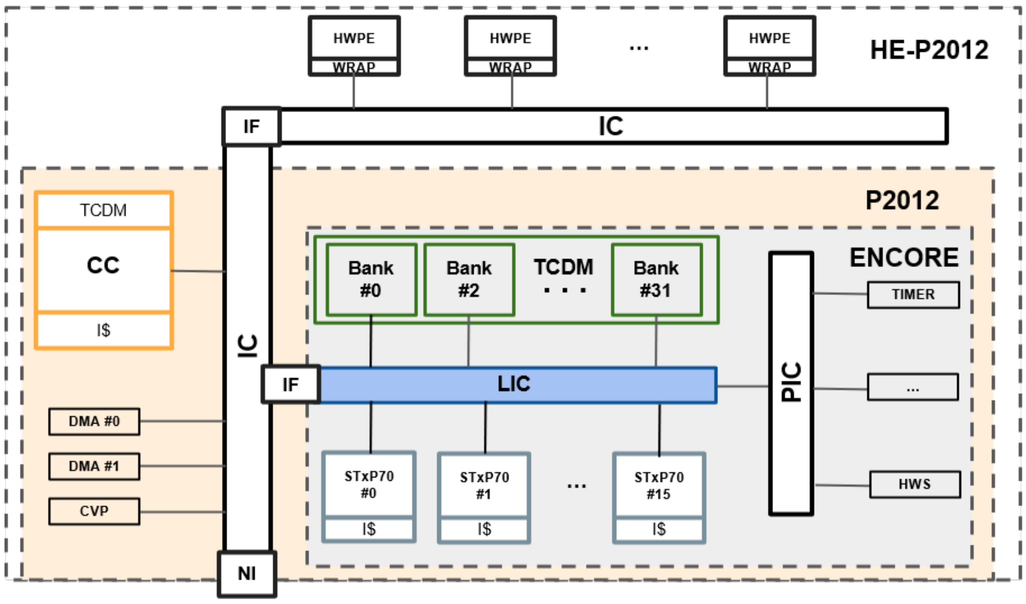


Figure 1.3: A simpified overview of P2012/STHORM cluster architecture.

Platform 2012 (P2012) is a general-purpose many-core accelerator for embedded design by STMicroelectronics composed of tightly-coupled homogeneous clusters [9]. With the design position of easy scalability, friendly implementation in a deep sub-micron process, it is easily programmable and is the basic platform for heterogeneous implementation. Each cluster features up to 16 processors and one control processor with independent instruction streams sharing a multi-banked Level one (L1) data memory, a multi-channel Direct Memory Access (DMA) engine, and specialized hardware for synchronization and scheduling. With the help of standard OpenCL and OpenMP parallel programming to provide the highest level of control on application-to-resource mapping and the addition of dedicated hardware (Intellectual Property) IPs, P2012 achieves extreme small area and energy efficiency by supporting domain-specific acceleration in the cluster level.

In P2012, each cluster, named ENCore, contains up to 16 STxP70-v4 cores sharing an L1 Tightly-Coupled Data Memory (TCDM) of 256 KB, 32 banks. Each core owns a 16 KB instruction cache, and there is no data cache. The asynchronous logarithmic interconnection (LIC) between the cores and the TCDM was designed to allow single-cycle access to the memory banks in case there is no contention [87]. Each cluster is equipped with a Hardware Synchronizer (HWS), which provides low-level services such as semaphores, barriers, as well as event propagation support with internal peripherals, such as Timer. The cluster also has a similarly designed peripheral interconnection used for communication between cores and peripherals (e.g., DMAs), memory outside of the cluster, and a Cluster Controller (CC) containing STxP70 core. As shown at the top, the cluster template can be enhanced with application-specific HardWare Processing Elements (HWPEs) interconnected to the ENCore with LIC to accelerate key functionalities in hardware to form a heterogeneous platform [20]. Compared with the original homogeneous P2012, it achieves up to 123 times speedup on the accelerated code region and saves 2/3 of the energy.

1.1.2.2 Parallel Ultra-Low-Power platform

1.1.2.2.1 Near-threshold Computing To explore the high energy-efficient computing platform, Near-threshold Computing (NTC) is proposed. Power consumption is a quadratic function of voltage and is proportional to CV^2f. As the voltage drops, we get significant power savings at the expense of performance. However, total power is a combination of static or leakage power and dynamic power. NTC's idea is that we need to find the minimum energy point while scaling down system voltage.

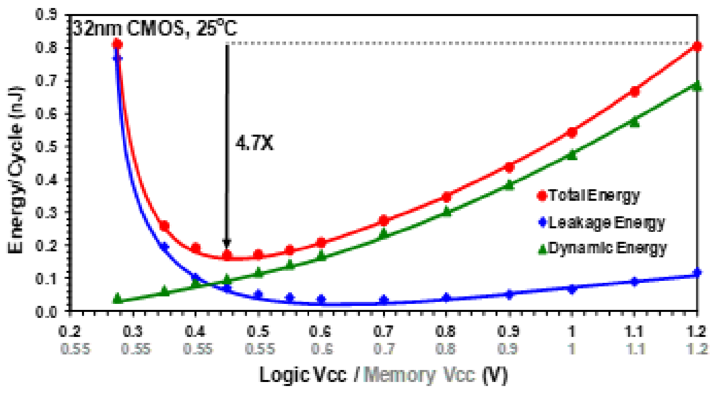


Figure 1.4: Measured power, performance and energy characteristics across wide voltage range for an IA-32 processor fabricated in 32nm [46]

In Fig. 1.4, as the voltage is dropped towards the transistor threshold voltage (Vt), the switching power decreases, but at the same time, the leakage current increases. This means that the optimal combination of leakage and switching power has to be found. Reducing the voltage below a certain power threshold increases the leakage faster than switching power decreases, and the performance will also be degraded. The optimum operating point is usually slightly above Vt and is called the near-threshold operating point or minimum energy point. However, NTC brings the promise of an order of magnitude improvement in energy efficiency. The frequency degradation due to aggressive voltage scaling may not be acceptable for high-performance applications for single-core [46]. Thus, to recover the performance degradation, parallel multi-core execution is proposed. Based on NTC, a multi-core platform combines extreme energy efficiency and high performance with parallel computing.

1.1.2.2.2 PULP architecture The Parallel Ultra-Low-Power Processing (PULP) platform is a general-purpose multi-core platform that achieves leading-edge energy efficiency and features widely tunable performance. PULP aims to satisfy the computational demands of IoT applications requiring flexible processing of data streams generated by multiple sensors, such as accelerators, low-resolution cameras, microphone arrays, vital signs monitors and so on. As opposed to single-core MCUs, a parallel ultra-low-power programmable architecture allows meeting the computational requirements of these applications without exceeding the power envelope of a few mW typical of miniaturized, battery-powered systems. Moreover, OpenMP, OpenCL, and OpenVX are supported on PULP, enabling agile application porting, development, performance tuning, and debugging (Fig. 1.5).

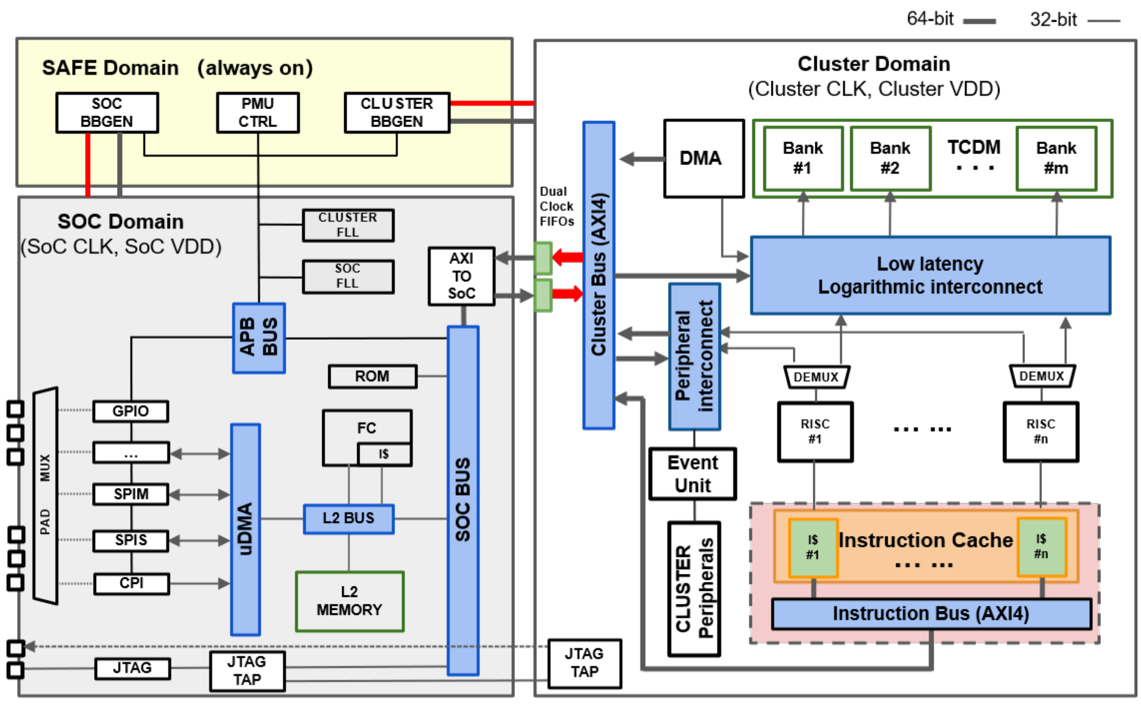


Figure 1.5: PULP block diagram [95]

Both the SoC and cluster domain feature with salable voltage and frequency. In the ultra-low-power parallel cluster, a tightly-coupled 1 cycle shared multi-bank L1 data memory is connected with each core through a low latency interconnect. The L1 data memory can be sized from 16KB SRAM to 32KB SRAM + 16KB latch-based Standard Cell Memories (SCM) heterogeneous memory architecture. Concerning SRAM memories, SCMs can achieve a lower density (by a factor of ∼3x), with the key benefit of providing much smaller energy per access (∼4x) [107]. There are N (2 - 16) configurable, simple one-issue cores in the cluster. The cores have a private instruction cache in PULPv1 and PULPv2 or shared instruction cache in PULPv3 connected to the cluster's Advanced eXtensible Interface (AXI) to access Level two (L2) memory in the SoC domain. One cycle hardware synchronization among all the cores to automatically manage idle cores' shutdown and support OpenMP and OpenCL pattern is implemented. Besides, an event unit to trigger idle cores is also employed to help power management. To exchange TCDM data with external L2 memory and peripherals, a lightweight, ultra-low-latency, multi-channel DMA is responsible for providing fast and flexible communication [91]. The DMA uses minimal request buffering and features a direct connection to the TCDM to eliminate the need for internal buffering, which is very expensive in terms of power. Finally, the RISC ISA-based (Reduced Instruction Set Computer - Instruction Set Architecture) core varies from OpenRISC (OR10N) to RISC-V (RI5CY) because of RISC-V compress instruction and easily extendable features (Fig. 1.6).

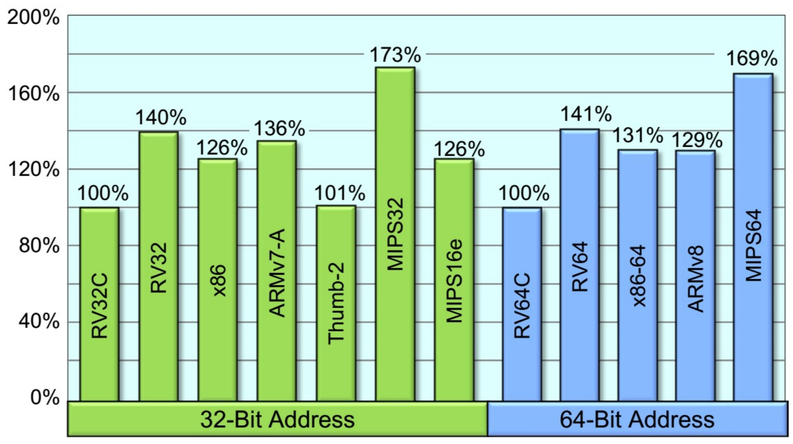


Figure 1.6: RISC-V Offers Simple, Modular ISA [55]

The cluster domain is responsible for parallel acceleration to improve performance, while the SoC domain is responsible for I/O and system control. A fabric controller (FC) takes control of all SoC by adjusting the (Power Management Unit) PMU and (Frequency Lock Loop) FLL, configuring the peripheral IPs, and cooperating with cluster to be responsible for data movement from peripheral to L2 memory with the help of micro DMA (uDMA). The always-on safe domain helps aggressively manage idle power with different power modes (deep sleep, sleep, idle and full active mode etc.). In the end, with the help of body-biasing to reduce leakage power, PULPv3 achieves 385 G-Ops/W [94].

1.1.3 Instruction fetch subsystem

1.1.3.1 Instruction fetch stage in RI5CY

The processing elements in the cluster are based on RI5CY, a small and efficient 32-bit, in-order RISC-V core with a 4-stage pipeline that implements the RV32IM[F]C instruction set architecture (Fig. 1.7). The core supports a custom extension to achieve higher code density, performance, and energy efficiency [33] [24]. It started its life as a fork of the OR10N CPU core based on the OpenRISC ISA. By inheriting custom ISA extensions including Hardware Loop (HWLP), bit manipulation, Single Instruction, Multiple Data (SIMD)-like instructions of vector operations, dot production for DSP, its performance speeds up to 9.5 times compared to the standard RV32IMC to 3.19 Coremark/MHz. Compared with OpenRISC ISA, RISC-V's fetch instruction implementation is more complicated. Firstly, since there is no delay slot in RISC-V ISA, the jump loses one cycle, and the next instruction is already being fetched and probably ready in the Instruction Fetch stage (IF). Secondly, there is no set-flag instruction and no branch prediction. As a result, the branch can only be taken after the execution stage (as shown in the red line) to re-target the Program Counter (PC). This combination path from the Execution stage (EX) to the instruction fetch interface outside the core will be on a critical path.

Thirdly, the instruction memory is word-aligned and does not accept misaligned accesses. With the combination of compressed instructions and normal instructions, the cross-word normal instruction needs to be assembled from two words (such as instr.2 and instr.3 in Fig. 1.8), except that if the lower half word is compressed such that there is no need to fetch next ready word. In this way, the core prefetcher needs to buffer at least two words. Therefore with the 32-bit fetch interface, four words are buffered in First In, First Out (FIFO) buffer to meet the consecutive pipeline requirement or four words (one cache line) for optimal performance and deal with

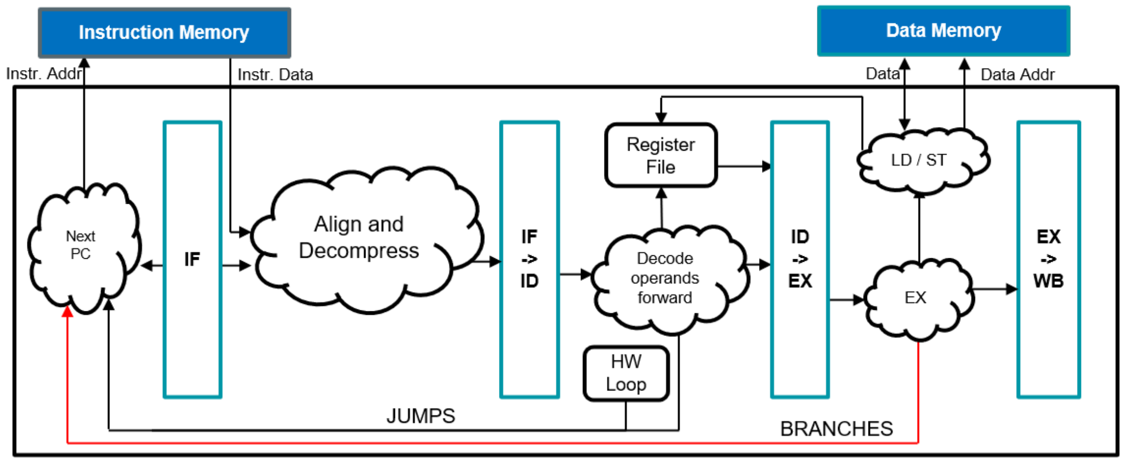


Figure 1.7: RI5CY block diagram

cache misses.

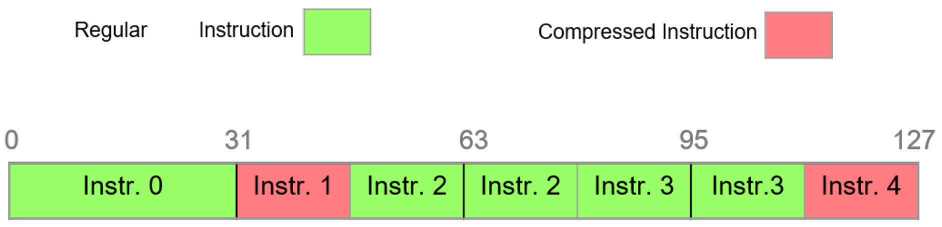


Figure 1.8: Misalignment of compressed instructions

1.1.3.2 Instruction fetch interface to iCache

As shown in Fig. 1.9, there are two channels between the IF and iCache - request and response channel. The core sends the fetch request with a target address. If the cache's TAG hits, it responds to the core with valid data in one cycle; if the iCache has a TAG miss, it waits AXI refill for at least 15 cycles before responding to the core.

Since the fetch request depends on the inner combination circuit generated by fetching valid and fetch data to issue the request as soon as possible. As a result, there are two types of long delay paths. One is from the fetch request ($fetch\_req$) to cache TAG lookup and the $fetch\_gnt$ generated by hardware loopback using handshaking protocol. The other is from $fetch\_valid$ or $fetch\_data$ to $fetch\_req$. As a result, the critical path degrades the systems maximum frequency and scalability.

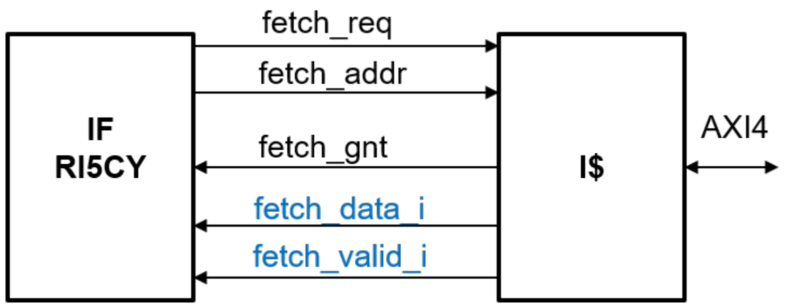


Figure 1.9: Instruction fetch interface. I$ stands for iCache.

1.1.3.3 Instruction cache structure

The L1 iCache connected to the core is used for instruction refill to hide the memory access latency. Its inner structure consists of the memories for instruction's TAG and DATA storage and the cache controller logic to manage the requests and the refills. It can be configurable in its total size, associativity, line size and replacement policy (including FIFO, Least recently used (LRU) or Random). The iCache has two main architectures:

- Private iCache: each core has its private iCache and a separate cache line refill path connected with AXI to the main memory, leading to high contention on external L2 memory. It is used in PULPv1 and PULPv2.

- Shared iCache: there is no difference between the private architecture in the data side except for the reduced contention L2 memory (only one line refill path exists). The shared iCache inner structure compromises a configurable number of banks, a centralized logic to manage requests and fairness mechanism to ensure the cores access to all cache banks. Thus, there are two ways: one is to use crossbar with round-robin scheduling that guarantees fair access to each banks with single port. The other is to use multiple port memories to eliminate the access congestion. Finally, in case of concurrent instruction missed from two or more banks, a simple bus handles line refills in round-robin towards the L2 bus. This structure is used in PULPv3 and Mr Wolf [85].

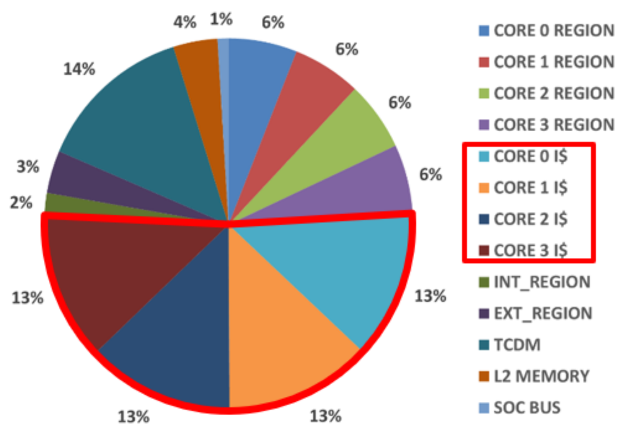


Figure 1.10: PULPv1 cluster power breakdown [92]

1.1.3.4 Instruction cache issue

In the PULPv1 with a private iCache based on SRAM, the iCache takes about 52% of the total cluster power shown in Fig. 1.10. Then, with the limitation of small cache capacity below 8KB, PULPv2 replaces the iCache's SRAM with SCM, reducing the power occupation of iCache to 25% [95]. Even though private cache is small and fast with only one cycle access latency, its relatively small cache capacity degrades the performance in terms of high capacity miss compared with the shared cache. However, the single-port shared cache (SP) suffers from high congestion when accessing the shared banks, especially with parallel programming such as OpenMP. Furthermore, the most critical issue is the scalability of the shared banks. When the core number increases to 8 or 16, the logarithmic interconnect leads to heavy congestion and timing issues. Moreover, due to the four pipelines RI5CY core's unconditional branch, the critical path from the core's EX to iCache worsens the situation.

To solve the single-port shared cache contention issue, on the one hand, we can increase the cache line size to reduce the core fetch frequency. The proposed cache line size can vary from 8 bytes to 32 bytes. On the other hand, a multi-port shared cache (MP) can be used, which keeps each core's read control logic private with n-port shared memory banks. Each private cache controller can access the memory banks simultaneously without contention, and a master controller is responsible for refilling from L2 memory and writing to the shared memory banks. The MP ensures

Cache type	Private	Shared	Shared	Target
		Single-Port	Multi-port	
Cache capacity	Small	Large	Large	Large
Performance	Low	High	High	High
Congestion	No	Yes	No	No
Area	Small	Small	Large	Small
Timing	Good	Bad	Bad	Good

Table 1.1: The pros and cons of different instruction cache architectures

one cycle hit access latency and shows the best performance while being suitable only for small cache sizes. However, the multi-port memories generate significant area overhead and lead to timing and power issues in P & R. Finally, how to solve this one cycle access from core fetch interface to instruction cache with large cache capacity, high performance, less congestion, small area, and good timing to meet the high-performance and low-power design requirement is the goal in our work (Table 1.1).

Chapter 2

Related work

Since the instruction access pattern has strong spatial and temporal locality, the instruction cache is very sensitive to increased access latency. Besides, the iCache's structure is more simple than the data cache with its read-only nature. However, iCache can still consume up to 50% of the overall system energy with the high-performance requirement. Thus, their optimization is a strict requirement for programmable ULP architectures [92]. Nevertheless, high performance means high power and energy, and various hardware architectural techniques for balancing the two factors are proposed [74] [73].

2.1 Ultra-low-power instruction memory

A standard methodology to reduce the energy consumption of the instruction memory in ULP systems is to adopt advanced memory technologies. Powell et al. [81] propose a circuit design named 'gated V$_{dd}$', which adds an extra transistor in the supply voltage path or ground path of the SRAM cell to form a 7T SRAM. It can reduce leakage power by gating the unused sections of a dynamically resizable iCache. In a Dynamically ResIzable instruction-cache (DRI iCache) [113], the key point is to turn off or to gate unused sections in iCache to estimate and adapt to the required iCache size dynamically (Fig. 2.1. The DRI iCache monitors itself in fixed-length sense *interval* with adaptive parameters, measured in a number of dynamic instructions (e.g., one million instructions). A miss counter counts the number of cache misses in each sense interval. At the end of each sense interval, the cache upsizes/downsizes, depending on whether the miss counter is lower/higher than a preset *miss-bound* value. A preset size-bound value is set to avoid thrashing. Thus, the 7T SRAM is shown in Fig. 2.2 by adding an extra transistor in the supply voltage path or ground path of the conventional SRAM cell to gate the power supply when the SRAM is unused. The results indicated that a wide NMOS dual-Vt gated-V$_d d$ with a charge pump reduces leakage most with minimal impact on cell speed and area.

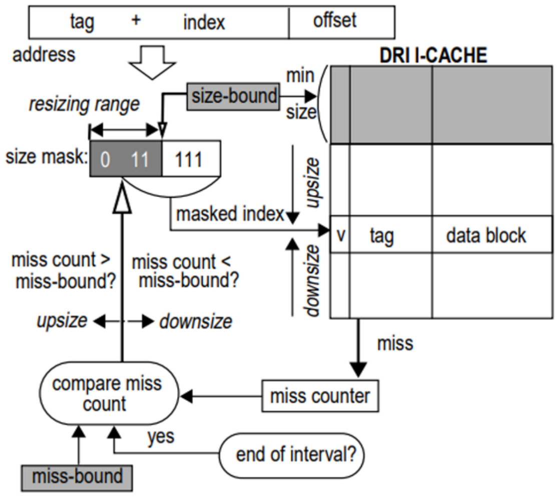


Figure 2.1: A direct-mapped DRI iCache's anatomy (same for set-associative caches) [92]

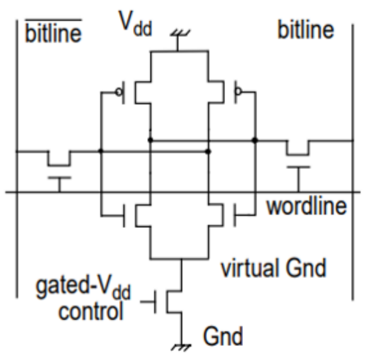


Figure 2.2: SRAM with an NMOS gated-Vdd [92]

Furthermore, the ultra-low-voltage operation of memory cells has become a topic of much interest due to its applications in very low energy computing and communications. As conventional 6T SRAM failed to archive subthreshold operation due to the effects of V_t variation [13], researchers have considered different configuration SRAMs for subthreshold operations having single-ended 8T [16] [110] or 10T bit-cells to improve stability. Ickes and al. [44] present a 10 pJ/cycle 32-bit microprocessor SoC with two 4KB custom 8T SRAMs (Fig. 2.3) and a small 1 KB (8 x 128B) instruction cache based on standard cell latches.

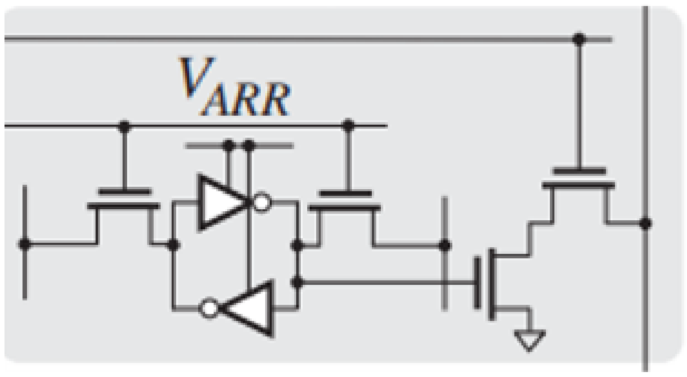
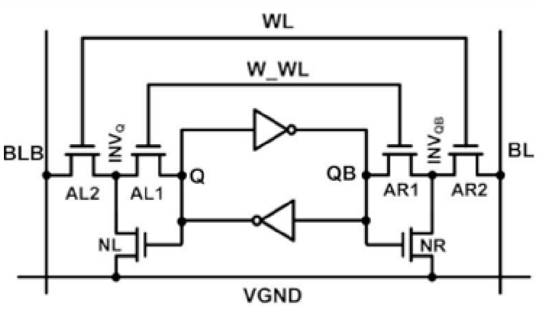


Figure 2.3: Zoom of 8T SRAM Architecture [44].

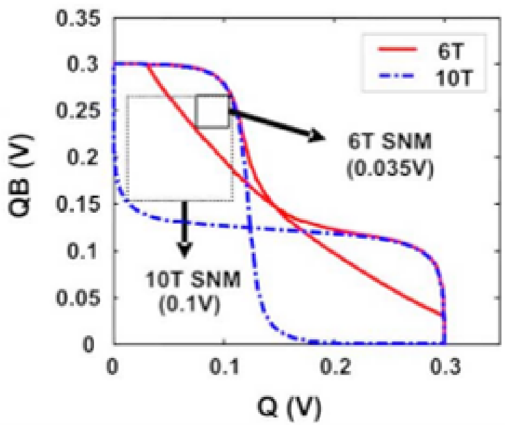
Chang et al. [17] propose a differential 10T bit-cell that effectively separates read and write operations to achieve high cell stability (Fig. 2.4a). In read mode, WL is enabled, and VGND is forced to 0V while remaining disabled. The disabled make data nodes ('Q' and 'QB') decoupled from bitline during the read access. Due to this isolation, the read Static Noise Margin (SNM) of the 10T cell is almost the same as the hold SNM of a conventional 6T cell, the read stability is remarkably improved in this 10T cell (Fig. 2.4b). During write mode, both WL and are enabled to transfer the write data to cell node from bitlines.

By boosting V_{WL} and V_{W_WL} by 100 mV (at 300 mV) and sharing a common VGND node with several SRAM cells, the 10T SRAM has good write ability and small area overhead. Besides, the leakage power of the proposed 10T bit-cell is close to that of the 6T. Based on the 10T SRAM optimized for sub-threshold operation, Myers et al. [75] introduce a Cortex M0+ based system with two 4KB 10T SRAM to achieve ultra-low power.

Moreover, Magnetic Random Access Memory (MRAM) is a promising emerging memory technology because of its advantages, such as non-volatility, high density, and scalability. In particular, Spin Orbit Torque (SOT) MRAM is gaining interest as it comes along with all the benefits of its predecessor Spin Transfer Torque (STT)



(a) 10T SRAM cell).



(b) Static Noise Margin (SNM) comparison of conventional 6T and our 10T cells.

Figure 2.4: Notion of 10T SRAM [17].

MRAM. Especially the split of read and write paths in SOT-MRAM promises faster access times and lower energy consumption than STT-MRAM. Oboril et al. [77] show that a hybrid combination of SRAM for the L1-data cache, SOT-MRAM for the L1 instruction cache and L2 cache can reduce the energy consumption by 60% while the performance increases by 1% compared to an SRAM-only configuration, targeting a 65 nm technology node.

In Spin Orbit Torque memories, the Magnetic Tunnel Junction (MTJ) cell stores the data as a resistance state value. An MTJ device consists of two independent ferromagnetic layers separated by a very thin (a few nm) barrier oxide layer such as magnesium oxide (MgO) (Fig. 2.5). One of the two ferromagnetic layers has a fixed magnetization with a fixed magnetic field. Hence, this layer is known as the fixed or reference layer. In contrast, in the second magnetic layer, called the free layer, the magnetization can be freely rotated based on the current direction flowing through the MTJ device. When the direction of the magnetic field of the free layer is parallel (P) to the fixed layer, the MTJ cell has a low resistance value. Instead, when the magnetization of the free layer is opposite or anti-parallel (AP) to the fixed layer, the MTJ cell has a high resistance value. These high and low resistance values represent logic' 1' and' 0' values.

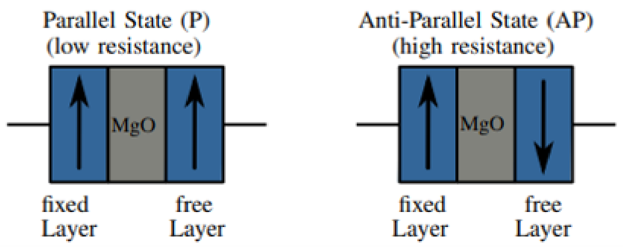


Figure 2.5: MTJ resistance according to the magnetization of the free layer [77].

This MTJ cell is the core part of a bit-cell in SOT-based memories as well as in STT-MRAM, as shown in Fig. 2.6. However, to eliminate the shortcomings of STT-MRAM, the SOT-MRAM bit-cell has an additional terminal to separate the (unidirectional) read and the (bidirectional) write paths which are perpendicular to each other. The terminals comprise a read line, a write line, a source line and a word line. The word line is used to access the required bit-cell during memory access via the NMOS-based access transistor. The comparison of the features among SRAM, STT-MRAM and SOT-MRAM are shown in Table 2.1. We can see that SOT-MRAM is suitable for cache memory with a smaller area, read latency and energy, especially for read-only instruction cache.

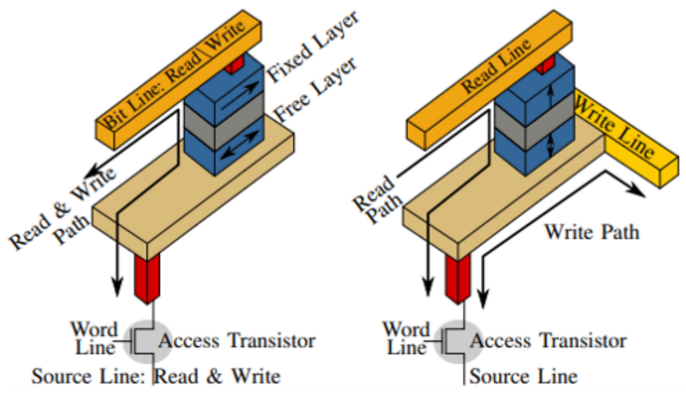


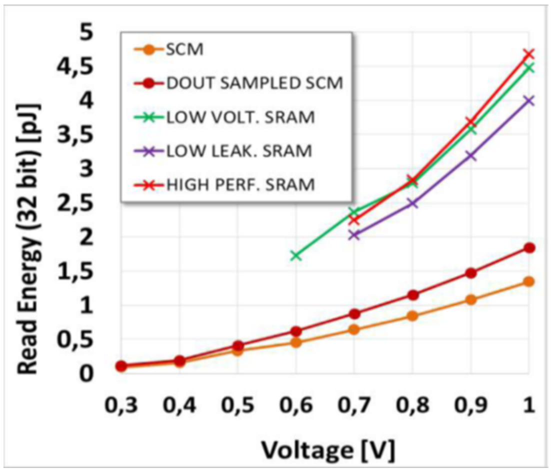
Figure 2.6: Bit-cell for STT-MRAM and SOT-MRAM [77].

	6T-SRAM	In plane STT-SRAM	SOT-MRAM
Data storage	Latch	Magnetization	Magnetization
Non-Volatility	no	yes	yes
Area [mm^2]	2.78	1.63	1.80
Read Latency [ns]	2.17	1.2	1.13
Write Latency [ns]	2.07	11.22	1.36
Read access Energy [pJ]	587	260	247
Write access Energy [pJ]	355	2337	334
Leakage power [mW]	932	387	354
Process	CMOS	CMOS + STT-MJT	COMS + SOT-MJT
Scalability	-	+	+
Endurance	++	+	+
Radiation immune	-	+	+
Bit failure Rate		-	?

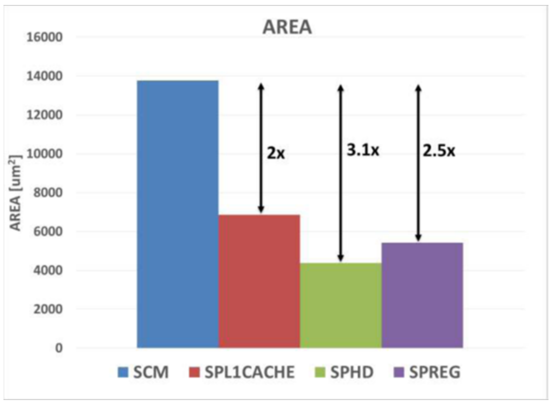
Table 2.1: Comparison of various memory technologies for a 512 KB memory [77].

Then, Garello et al. [32] demonstrate for the first time full-scale integration of top-pinned perpendicular MTJ on 300 mm wafer using CMOS-compatible processes for spin-orbit torque (SOT)-MRAM architectures with 62 nm devices. Kuan and Adegbija [61] show that an energy-efficient, highly adaptable last-level STT-RAM cache (*HALLS*) can reduce the average energy consumption by 60% in a quad-core system while introducing marginal latency overhead. In order to solve the backward of long write latency and high write energy of STT-MRAM to fit Last-Level Cache (LLC), HALLS allows the LLC's configurations to be dynamically adapted to executing applications cache configuration and retention time requirements. Thus, the STT-MRAM can be a viable option for mitigating the overheads of implementing the STT-RAM in LLC.

However, the subthreshold SRAM has a limited voltage range and large read energy, STT-MRAM has a slow write speed and high write power, and SOT-MRAM has a large area for high-density memory. Another solution for the small low-level cache is to use latch-based Standard Memory Cells (SCM) because of their design flexibility, ease of implementation, and robust operation at low supply voltages. Te-



(a) 10T SRAM cell).



(b) Static Noise Margin (SNM) comparison of conventional 6T and our 10T cells.

Figure 2.7: Notion of SCM [106].

man et al. [106] present a controlled Placement for the synthesis and place and route (P & R) to optimize the distinct and regular structure of an SCM array. Through careful floor planning, the specific structure of these entirely digital blocks can be manipulated in order to optimize their placement and minimize routing congestion and wire length. The proposed architecture for controlled SCMs includes dual-level write clock-gating, latch-based storage, and a NAND/NOR tree for read mux realization. These components are instantiated in RTL and placed during the floorplanning stage of the P & R flow. High-optimized SCM Macro are inserted, leading to a structured, non-congested layout with close to 100% placement utilization and reduced wire length. Based on that, PULPv2 [95] replaces SRAMs with SCMs in the instruction cache and increases the SoC energy efficiency by 38%. SCMs present extremely interesting features for small memory size, low-voltage, and energy-efficient designs, since (i) they can operate with very low voltage, even lower than 10T SRAMs optimized for low voltage [17], and (ii) their energy per access is significantly smaller than SRAMs (2 - 4×) (Fig. 2.7a). Nevertheless, although the controlled placement of standard cells memory array reduces area overhead [107], there is still 2× the area of the same size SRAMs-based memory (Fig. 2.7b). It is thus clear that since there is a strong motivation to use energy-efficient but low-density memories for instruction cache, there is a strong push to maximize the effective capacity of the caches through sharing schemes.

2.2 Improving Instruction Fetch Efficiency

Given a specific cache capacity, the miss rate of instruction fetch is a critical factor influencing performance and energy efficiency. Modern processors usually use two-way and four-way set-associative caches to reduce conflict misses. Besides, instruction prefetching is implemented inside or outside the core [100], including branch prediction [115] are effective methods to boost cache performance by reducing compulsory and capacity miss. Support of compressed instructions is also effective in decreasing capacity misses.

In this direction, we propose a hierarchical instruction cache with a 4-way set-associative design, which also benefits from the RISC-V' C' extension for compressed instructions. Sharing instruction memory is not new and has been mainly exploited in high-end computing platforms such as General Purpose Graphic Processing Units (GP-GPU). In GP-GPUs, all the compute units in each multiprocessor execute their threads in lock-step according to the order of instructions issued by the instruction dispatcher, which is shared among all of them [112].

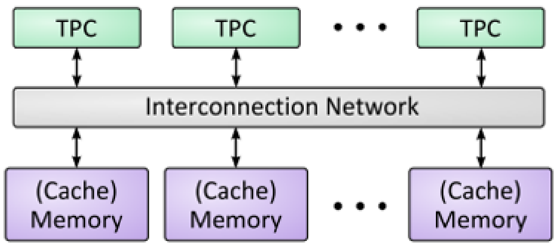


Figure 2.8: GPU with Thread Processing Cluster (TPCs) and Memory Banks [112].

Besides, all shared-memory multiprocessing architectures supporting OpenMP can benefit from energy reduction by reducing the instruction fetch miss rate with a larger cache capacity. Loi et al. [67] present two multi-banked shared instruction caches to solve the private cache's small capacity issue in PULP. The private cache achieves higher speed due to the simple design, but its performance is limited by the small cache capacity for each core, leading to a high miss rate. Moreover, the parallel program's redundant copies in the system may waste the bandwidth to the main memory, leading to more power consumption. By combing all the cache banks to share with all cores, the shared cache is proposed to solve the small cache capacity issue, which offers a low miss rate at the cost of minimum hardware area. However, the congestion of accessing the same cache banks may degrade the performance, and hardware complexity generates lower speed. Furthermore, multiple port memory is used inside the cache bank for TAG and DATA to solve the congestion issue. Nevertheless, with increased large hardware area and additional miss penalty, the performance of multi-port shared cache may drop compared with the single-port shared cache. As a result, it is suitable only for cache with a small memory size - a few KB.

2.3 Instruction Cache Prefetching

Without introducing a large area/energy penalty and boosting the performance, the importance of latency hiding techniques such as prefetching grows further. Prefetching means prefetching the next N-line to hide the latency of the core. *Prefetch on miss* and *tagged prefetch* are proposed by Smith [100]. On a miss, *prefetch on miss* always fetches the next line as well. It can cut the number of misses for a purely sequential reference stream in half. *Tagged prefetch* can do even better. In this technique, each block has a tag bit associated with it. When a block is prefetched, its tag bit is set to zero. Each time a block is used, its tag bit is set to one. When a block undergoes a zero to one transition, its successor block is prefetched. If fetching is fast enough, this can reduce the number of misses in a purely sequential reference stream to zero. However, it is unsuitable for non-sequential execution paths caused by jumps, conditional branches, and system calls. Despite these shortcomings, it is still an effective strategy to reduce cache misses by 20-50%.

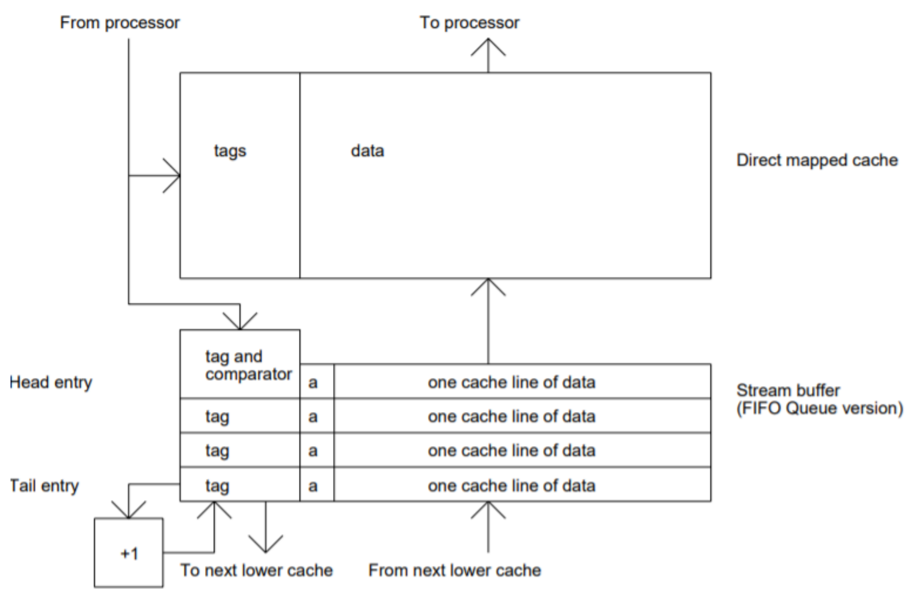


Figure 2.9: Sequential stream buffer design [52].

Unfortunately, the large latencies in the base system can make this impossible. Thus, Jouppi [52] presents a prefetching scheme using stream buffers in Fig. 2.9 to prefetch cache lines starting at a cache miss address. This technique is more effective than previously investigated prefetch techniques using the next lower level in the memory hierarchy when pipelined. On a cache miss, sequential cache blocks are prefetched into a separate FIFO stream buffer until it is filled to avoid L1 cache pollution. Next time, when the L1 cache sees a miss, the first entry of the stream buffer is checked, and, on a hit, a block is brought into the L1 cache. Jouppi also explored using multiple streaming buffers in parallel that can prefetch multiple intertwined reference streams.

Fetch-directed instruction prefetching (FDP) [88] separates branch predictor and instruction cache, so the branch predictor can run ahead of the instruction cache fetch (Fig. 2.10). The branch predictor produces fetch blocks into a Fetch Target Queue (FTQ), then by using Cache Probe Filtering (CPF) to remove useless prefetch blocks in the FTQ, it sends only useful prefetch addresses to a Prefetch Instruction Queue(PIQ) and the branch predicted fetch blocks could be accurately prefetched and thereby saving bus bandwidth to the L2 cache. This FDP relies on accurate branch predictors and a sufficiently large Branch Target Buffer (BTB) to cover the control flow.

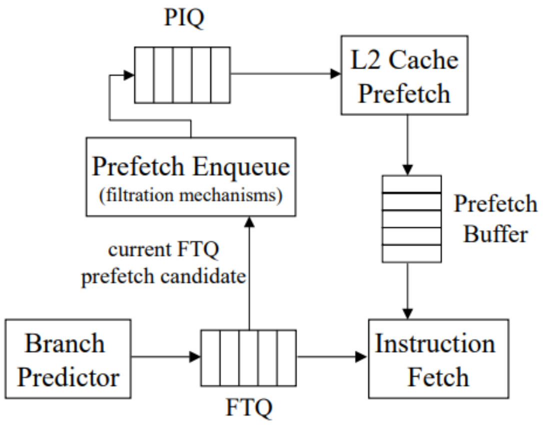


Figure 2.10: Fetch Directed Prefetching Architecture [88].

Proactive instruction prefetching (PIF) [26]-[27] is based on the fact that the stream of instruction cache misses is repetitive, and it eliminates the future instruction cache misses directly by tracing these temporally correlated streams. Based on that, RAS-directed instruction prefetching (RDIP) [60] correlates instruction cache misses with the program context captured from the Return Address Stack (RAS). It stores these misses in a *Miss Table* that is looked up using the signatures formed from the contents of the RAS (Fig. 2.11)). They simply XOR the bit-string representing the RAS state to 32-bit signatures. It brings 2% performance improvement compared to PIF with nearly 3X reduction in storage and 1.9X reduction in energy overhead. However, the main shortcoming of the temporal prefetchers is their high storage budget requirements, larger than 60 KB.

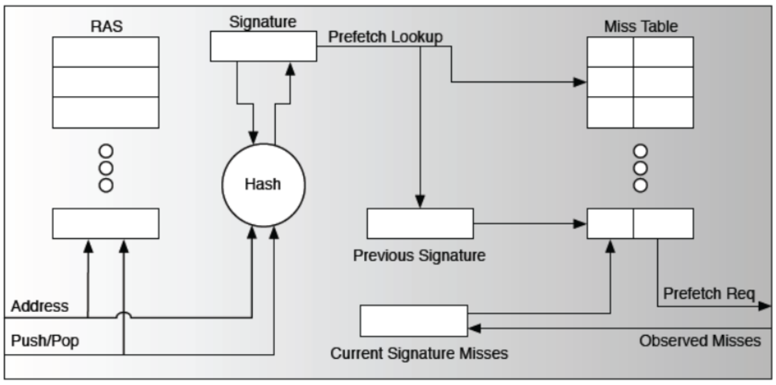


Figure 2.11: RAS-directed instruction prefetching architecture [60].

In the IPC-1 [41], numerous hardware instruction prefetchers were published. D-

JOLT [104] is a refinement of RDIP. It improves by implementing a new signature generation mechanism that generates the prefetches from a FIFO structure that stores the most recent function return addresses instead of the stack structure used by RDIP. They investigate the characteristics of the RDIP from the following three aspects: *Siggen*, *Histlen* and *Distance* (Fig. 2.12). By increasing the history length of the number of addresses used to generate a signature, D-JOLT can improve the prefetch accuracy while leading to less capacity efficiency. Besides, with a larger time distance of the signature associated with the generated miss address, D-JOLT can improve the prefetch coverage and be in time while losing some accuracy. With these observations, they propose a novel Siggen using a FIFO, which can use the correlations of the last function calls, and a hybrid combination of three prefetchers that consume three miss tables. Finally, D-JOLT consumes a large capacity to achieve high performance. With an 8KB entry miss table, it gives total storage of 125KB [90].

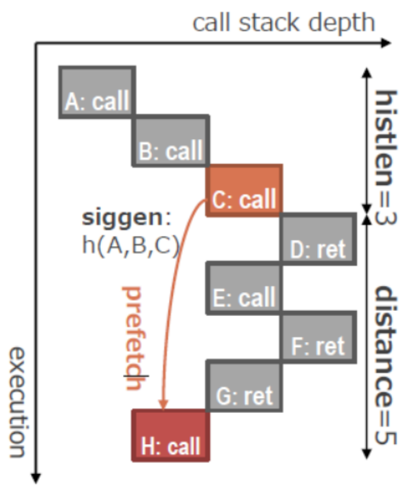


Figure 2.12: Three main characteristics of the RDIP analyzed by D-JOLT. Each block represents a dynamic instruction sequence with a call/return instruction at the end. The *Siggen* is the algorithm to generate a signature. The *histen* is the number of addresses used to generate a signature. The *Distance* indicates how much time has elapsed since the signature associated with the miss address was generated. This figure shows the prefetch block address from the call instruction of C is H, using h(A, B, C) [104].

FNL-MMA combines a Footprint Next Line (FNL) prefetcher and a temporal correlation prefetcher (Multiple Miss Ahead (MMA)) on the I-Shadow cache, which is a small tag-only cache. FNL is an enhanced next line prefetcher with extra trace recording tables that estimate if a cache line is worth prefetching. MMA selects the look-ahead distance to avoid the late prefetches in the Next Predicted Miss prefetcher (NPM) for non-continuous prefetch blocks. With an 8K entry miss table, it takes total storage of 97KB [90].

EPI [89] introduced the concept of entangling the cache line to be prefetched (destination) with a source cache line such that the destination would be prefetched when the source cache line is encountered. For an cost-effective EPI [89], only the head (first cache cache) of a basic continuous block for **src** and **dst** is paired. Fig. 2.13 shows the basic implementation of the EIP in the right-top part, a *Basic block* computes the head (first cache line) and size of a dynamic basic block by simply comparing the current address with the head address plus size. Once a new basic block is detected, it is stored in the *Entangled table* (left part of Fig. 2.13).

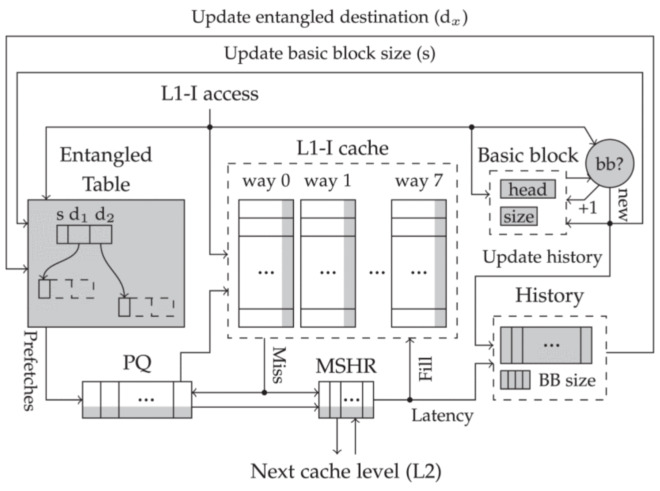


Figure 2.13: Overview of the Entangling I-prefetcher with hardware extensions shown in gray. The Basic block registers head and size keep track of the current basic block. History is a small circular queue and each entry records basic block information: head, size and timestamp of the first L1I access. Entangled is a cache-like structure and each entry consists in a src-entangled cache line, its basic block size, and a compressed array of dst-entangled cache lines (for the advanced optimization techniques, a confidence counter is associated to each destination). The L1I, PQ and MSHR are extended with information on timing (the timestamp when the request was issued) and on the src-entangled (position of the source in the Entangled table and an access bit indicating if the access stems from a demand access or a prefetch) [90].

To build entangled pairs for timely prefetching, first, it stores the recent history of the basic block heads together with the timestamp of their first access to L1 iCache to a small circular queue called *History buffer*. Second, it calculates the latency of demand L1 iCache miss with the help of start and end timestamps. The start timestamp is held in the miss status holding register(MSHR) with two extra fields: one access bit for demand misses and a pointer to *History buffer*. The access bit is to distinguish a demand miss or prefetch miss to track the latency of the prefetches to compute the actual latency on a late prefetch. The prefetches issued in PQ also have timestamps, and their access bit is set to 0. PQ also exchanges with MSHR when the prefetches miss in iCache. When a late prefetch happens, the corresponding access bit is set to 1. The end timestamp can be obtained with the time of the cache fill. When the access bit in MSHR is set to 1, and the history pointer is valid, it means a *src-entangled* cache line is found, and then the *Entangled-table* is updated. If the access bit is 1 and the history pointer is not valid, no *src-entangled* is searched for. In the end, once there is a hit in the *Entangled-table*, the current and *dst-etangled* basic blocks are prefetched. Other optimizations, such as adding confidence, merging spatio-temporal basic blocks and compressing destinations to improve the performance-area trade-off, are added. Typical EIP [89] models highly associative structures (e.g., a +1000-entry history buffer and a 34-way Entangled table which gives +8K entries). Its total storage requirements are 127.9KB. However, the size of the cost-effective EPI can reduce to 40KB.

Besides these large metadata prefetch strategies, other types of prefetchers interact with hardware structures, such as the branch predictor (e.g. BTB directed), to gain insights into the program's execution ahead of time. As a result, they require intrusive changes in the processor design. One of such prefetcher is SN4L [4]. It is based on FDP and comprises three predictors: selective next-four-line (SN4L), discontinuity (Dis), and BTB prefetching. SN4L improves next-four-line prefetcher using a usefulness filter. Only previously useful cache lines among the next four will be prefetched. Dis records the 4-bit offset of the branch instruction from the last two demanded instructions upon iCache misses. On iCache accesses, Dis looks up DisTable and generates prefetches if the missing line is found. Otherwise, the BTB will be consulted. Finally, BTB prefetching is employed to reduce BTB misses. BTB prefetching can be activated when an iCache fill is processed. The predecoder identifies the PC-relative branches and installs those detected branches unconditionally into the BTB array. Because of the nature of the prefetching scheme, the register-relative indirect branches cannot be prefetched by this mechanism. SN4L prefetcher only requires 2.06KB of storage [90]. MANA [5] is a refinement of SN4L-Dis-BTB that uses an 8-bit vector for consecutive prefetchers (previously proposed by PIF [26]]). It offers a good performance-area trade-off, and it is representative of state-of-the-art BTB directed instruction prefetchers. Here is the summary of performance and storage for all kinds of state-of-art iCache prefetch [90]. We can see that FDP based SN4L+Dis+BTB and MANA have a large benefit of good performance-area trade-off without large meta-data.

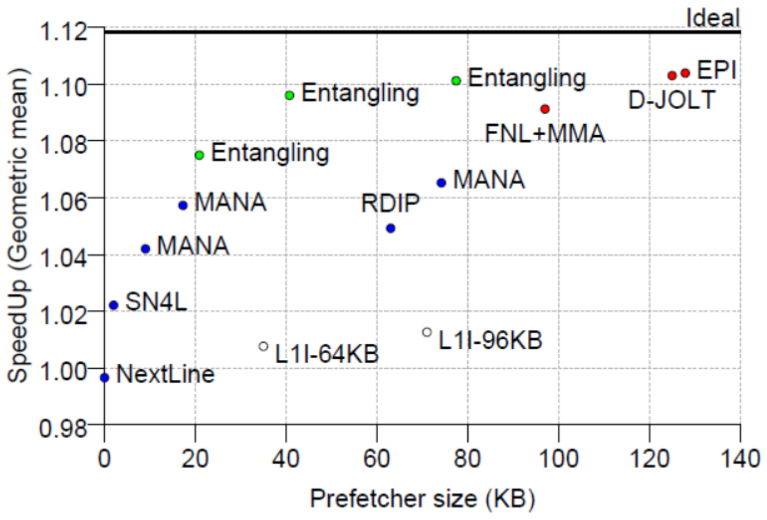


Figure 2.14: IPC vs mempry requirements [90].

In the end, from the view of industry perspective, Yasuo et al. [45] propose an effective FDP-base frontend design with only 195 bytes of hardware overhead. The authors point out that academic research always uses a less-than-optimal frontend baseline with a small BTB. At the same time, the industrial or commercial solutions always keep accurate branch predictors and large BTB with FDP. To overcome its previous issues, it has two enhancements, taken-only branch target history and post-fetch correction. It outperforms the 1st Instruction Prefetching Championship (IPC-1) winners with a 128KB storage budget.

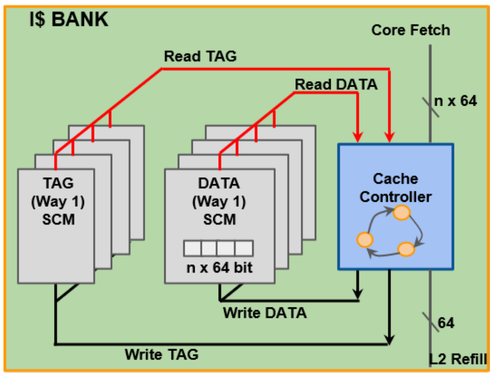
Nevertheless, for a ULP multi-core cluster with limited instruction cache capacity, these prefetchers all have the same issue of significant metadata overhead because of extensive address tracking and analysis, which brings much extra area and power. Thus, it is necessary to balance the performance with the cache power. Besides, keeping a large or small BTB for each fetch access for each core in the ULP cluster also brings large dynamic power. In this work, we employ a simple sequential next-line prefetch (4 instructions) to hide the L1 to L1.5 latency without jeopardizing energy efficiency. We expose this feature to allow software-controlled enable/disable of the prefetcher to adapt to application characteristics and trade-off performance and energy efficiency.

2.4 State-of-the-arts ICache in ULP cluster

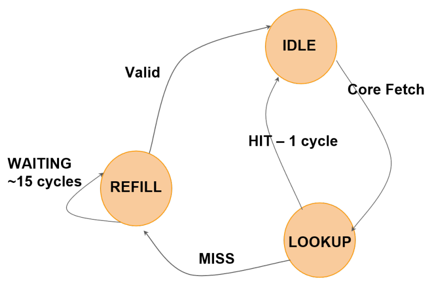
2.4.1 Private Instruction Cache

The baseline cluster features private instruction caches (Fig. 1.5). Each private cache bank comprises 3 elements, a TAG, a DATA array implemented using SCMs and a cache controller using a request-grant handshake protocol with a pseudo-random (PRAND) replacement policy. To refill from L2, each cache bank sends the request to the AXI bus independently with AXI interconnect nodes. Fig. 2.15a shows the details of the private cache bank. We can see that the core controller receives the fetch request from the RI5CY core with a cache line size of $n \times 64$ bits. Here we choose a 128-bit cache line to reduce the core fetch frequency since it is enough for a 32-bit RI5CY core. At the same time, the cache controller will read TAG and DATA to check if the fetch address is inside. If a cache hits, then the data is read back to the core with one cycle. If the cache miss, the controller goes to refill state and waits for AXI bus refill for about 15 cycles. Once controller get the data, it sends to the core (Fig. 2.15b, 2.15d). As a result, when executing parallel applications, each core fetches the same instructions while there is a miss, then all the cores will suffer from a miss and ask for AXI refill with the same instruction for about 15 cycles. This is a waste of AXI bandwidth with redundant refills.

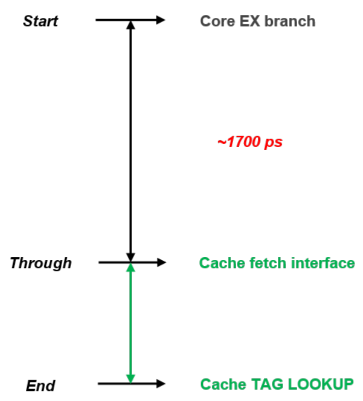
The private cache has one cycle latency when hit and its critical path in the cache controller is from core's $fetch\_req$ to cache bank's tag memory LOOKUP shown in arrow 1 in Fig. 2.15d). As we mentioned in section 1.1.3.1, it starts from the core's EX stage to determine branches. This path delay is about 1700 *ps* with the synthesis in GF22FDX technology, which limits the maximum frequency of the core fetch subsystem.



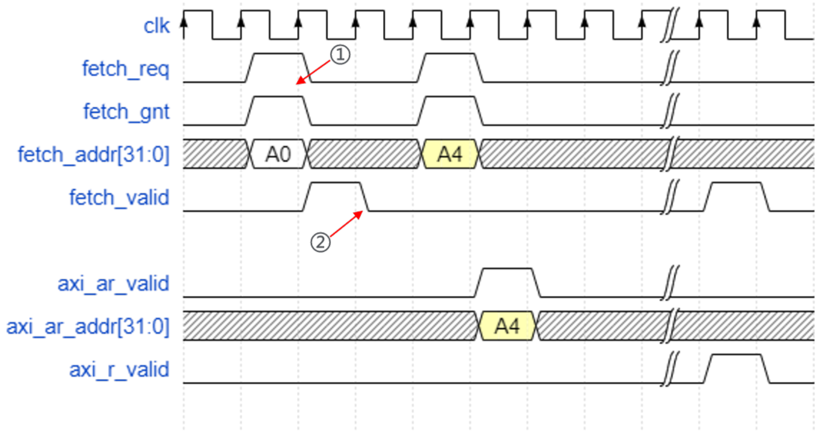
(a) Cache bank



(b) Cache controller finite-state machine (FSM)



(c) Notion of critical path.



(d) Instruction fetch timing diagram.

Figure 2.15: Private cache bank subsystem

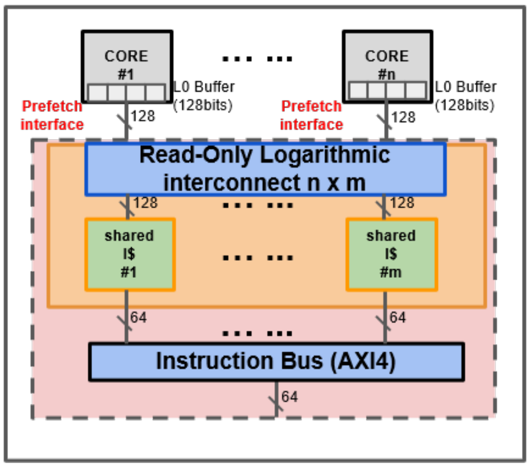
In conclusion, the private caches are fast (i.e., small critical path) and simple (i.e., low-power). Data replication and high miss penalty for large footprint applications are major drawbacks for private instruction cache, which decrease their performance and energy efficiency.

2.4.2 Shared Instruction Cache

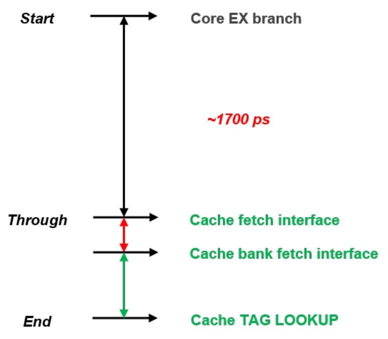
Shared instruction cache is shown in Fig. 2.16a. It uses the same cache banks as the private cache Fig. 2.15a, and it sends instruction refill to L2 memory only once when the cores access the same shared banks (AXI bus refill only once). Data replication is avoided since cache banks are shared among the cores, but two or more cores may compete to access the same cache bank. In such a condition, a round-robin arbitration policy ensures that only one core can access the cache bank and keeps the others stalled. As a result, each core has the same probability of accessing the cache bank. To better spread access among multiple banks, a read-only interconnect with n input and m output is employed to ensure fair access to the cache banks.

2.4.2.1 Logarithmic interconnect

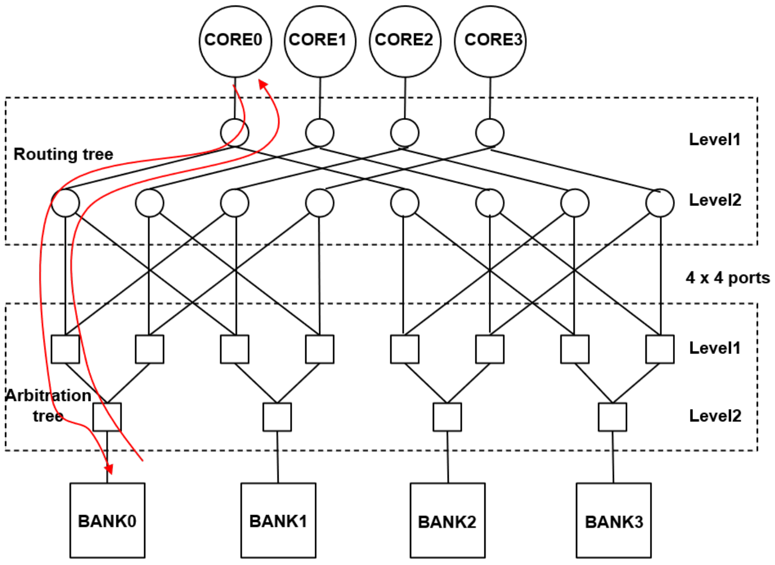
The logarithmic interconnect is a parametric, fully combinational Mesh-of-Trees (MoT) interconnection network to support high-performance, single-cycle communication between processors and memories in L1-coupled processor clusters [87]. As shown in Fig. 2.16c, a combinational path is created based on a network of routing primitives (circles blocks) and arbitration primitives (square blocks). The former is used to create independent routing paths (routing trees) from the cores to the arbitration tree and vice-versa. The latter is used to arbitrate concurrent requests (arbitration tree) and route them to the memory banks and vice-versa. The interconnect ensures access to large shared memory for cluster cores. However, it creates more logic delay in the critical path from core's $fetch\_req$ to memory's tag lookup (shown in red arrow Fig. 2.16c). Therefore, the delay needs to be evaluated in detail. Post-placement & routing results show that the delay of n processors and m memory banks with 32-bit data size ($n \times m$), the delay is expressed by a certain Fan-out of 4 (FO4), which is a measure of time independent of CMOS technologies, the gate delay of a component with a fan-out of 4 [39]. Our target shared instruction cache, with an 8 × 8 configuration, has 32FO4 (19FO4 from cores to memory and 13Fo4 from memory back to cores), which means that the combinational logic delay for the critical path from core's *fetch_req* to memory's tag lookup will increase 19FO4. This increased delay will push the pressure of the core's fetch timing and reduce the maximum frequency, as shown in Fig. 2.16b. Finally, the maximum performance will decrease with the shared instruction cache.



(a) Shared cache



(b) Additional critical path introduced by LIC



(c) Mesh of trees 4x4: empty circles represent routing switches and empty squares represent arbitration switches.

Figure 2.16: Single-ported Share Cache subsystem

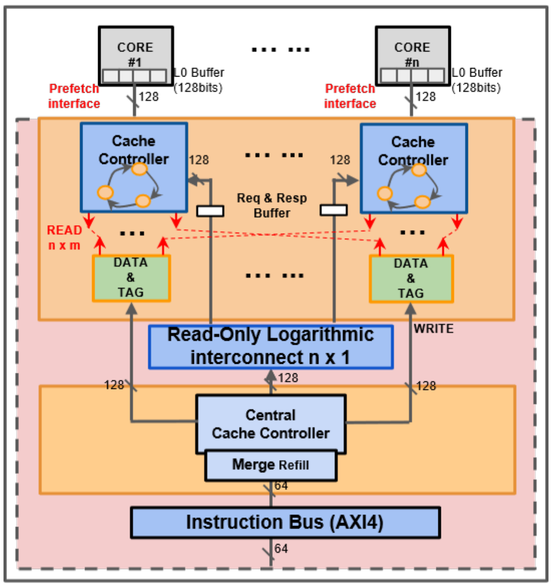
When one of the core's fetch requests goes to the cache controller of each shared memory bank, if a miss happens, the cache controller asserts the respective refill request and continues accepting incoming fetches from other cores while waiting for the refill response from L2. The shared cache controller can also track more than one pending refill by additional FIFO to track pending misses and restore the correct order from the response coming from the AXI BUS. If incoming fetches ask the same missed address after the L2 refill, they will be served by the TAG and DATA memories instead of refilling in L2. The merge of the same miss refill largely reduces the redundant L2 refill frequency, which is the purpose of shared caches. The shared cache can reduce n times the L2 refill frequency where n is the number of cores, especially in parallel applications.

Consequently, the shared instruction cache benefits from large cache capacity to avoid data replication while minimizing the access cycle (single-cycle-latency) and area overhead. The single-ported shared instruction cache features a read-only low-latency crossbar that uses a round-robin arbitration policy for each core's fetch request. Since one cache bank can serve one refill request each time, it causes congestion when several cores access the same cache bank for parallel applications. Moreover, a long path is present between the instruction fetch stage of the core to the cache banks through the interconnect, which limits the maximum frequency.

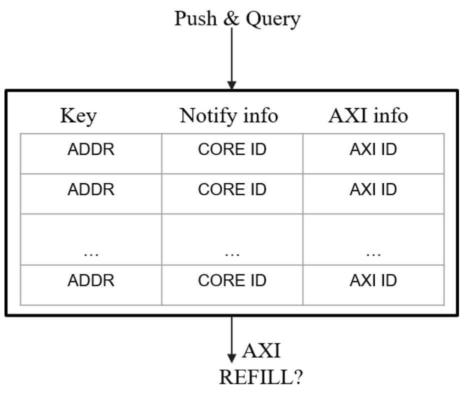
2.4.3 Multi-ported Instruction Cache

A new shared instruction cache is proposed to share only TAG and DATA memories while keeping a private cache controller for each core to solve the congestion issue of a single-ported shared cache. This can be realized by using multi-ported SCMs. Fig. 2.17a shows the detailed view of this cache. We can separate the multi-ported shared cache into two parts or two levels. Level 1 is composed of a private cache controller and multi-ported banks. Level 2 has a central cache controller cooperating with a merge refill unit. These two levels are still connected with the same logarithmic interconnect with $n \times 1$ (n private controllers and one central controller). Finally, a request and response buffer is employed to cut the critical path between levels to reduce the timing pressure of level 1 to level 2.

In the first level, now there can be m memory banks, and each of them is composed of TAG and DATA and a private cache controller. Besides, it has a single write port (written by the central cache controller after a miss) and n read ports, one for each private cache controller (n cores) and used for normal cache access (hit or miss). Each cache controller has a direct and private path (no contention) to read TAG and DATA memories with multi-ported memory banks. Therefore, this cache is similar to the private cache while using different ways to deal with cache misses.

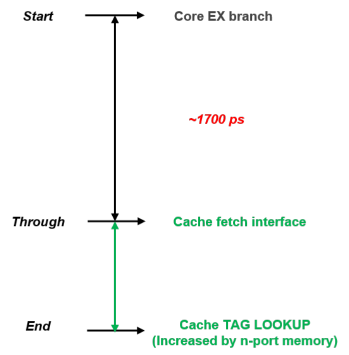


(a) Shared cache


Push & Query
AXI REFILL?

Key	Notify info	AXI info
ADDR	CORE ID	AXI ID
ADDR	CORE ID	AXI ID
...	...	...
ADDR	CORE ID	AXI ID

(b) content addressable memory



(c) Additional critical path introduced by multi-port memory

Figure 2.17: Multi-ported Share Cache subsystem

In the second level, different from the private cache, instead of directly sending n refill request to AXI BUS when each cache controller has a miss with the same address, the multi-port cache handles the refills by a dedicated central cache controller, which is in charge of merging the refills with the same address, sending refill to L2 as well as updating the TAG and DATA memories. The refill merge is realized by a CAM (content addressable memory) like a small cache, all refills to L2 with the same address can be merged into 1 refill by referring CAM. As shown in Fig. 2.17b, the refill address is written into CAM and generates a unique AXI ID (entry ID), and the address will be used as a key. If the same address hits in the CAM, no refill request is generated, and only the CORE ID field is updated, meaning that the cores linked to the hit refill address in the CORE ID field will be notified after the L2 response. If there is a refill address miss, the first empty CAM entry is allocated, and a corresponding refill request is sent to L2.

The multi-ported shared cache may have the best performance with free access to memory banks without contention and minimum L2 refill traffic. However, with the request and response buffer between levels, its miss penalty is two more cycles than the single-ported share cache. This can result in performance decline compared with single-port shared cache in extreme cases: all instructions are missed. Besides, the multi-port memory banks introduce a large area and create wire congestion in place & route of back-end. This also augments the memory read latency, as shown in Fig. 2.17c.

In conclusion, multi-ported shared cache benefits from large cache capacity and avoiding. Nevertheless, even if the heavy congestion for cores' access is avoided, n-ported TAG and DATA memories still have a series of area overhead issues. As a result, it is suitable only for cache sizes up to a few KB.

2.5 Thesis Outline

In this section, we describe the organization of the remainder of the present thesis work.

In chapter 3, we compare different architectures for iCache targeting tightly coupled clusters in detail. The analysis involves private iCaches per core, a shared iCache with single-port memories, as well as a shared iCache with multi-port memories. With synthetic micro-benchmarks and real program workloads, we can figure out an effective iCache architecture configuration to support high performance in a multi-core cluster by varying cache capacity, cache associate, and cache line block size. We summarise the three architectures' characteristics and drawbacks to provide clear evidence for the next exploration.

In chapter 4, we propose a two-level iCache to balance a multi-core cluster's performance and energy efficiency, which combines the private iCache for each core as L1 with 1 cycle latency and single-port shared iCache as L1.5 with 2 cycle access latency. It benefits from simple L1 and large cache capacity of shared cache while with a relatively large area. On average, when executing a set of real-life IoT applications, our multi-level cache improves performance and energy efficiency by 10% concerning the private iCache system and improves the energy efficiency by 15% and 7% with a performance loss of only 2% concerning the shared iCache. Besides, the relaxed timing makes two-level iCache an attractive choice for aggressive implementation, with more slack for convergence in physical design.

In chapter 5, we exploit adopting sequential prefetch between caches to load as soon as possible the instruction will be used in the future since two-level iCache with limited small L1 cache capacity has the performance drawback compared with shared cache, up to 25% performance drop in some large library based applications. With low power consumption requirements, efficient prefetch should be considered to reduce L1 capacity miss without introducing much power. Thus a dual-port read TAG memory is used for refill and prefetch lookup access in the L1. The result shows that it constantly improves the performance by 7% compared with the no prefetch one. Compared with private cache, it improves the performance by 15% with an energy efficiency loss up to 7%. Compared with shared cache, it almost keeps the same performance and improves energy efficiency by 7%.

In chapter 6, we focus on the timing optimization of the core instruction fetch stage in section 1.1.3.2. It is necessary to analyze and optimize the critical paths in the request and response channels. By removing the critical paths through the implementation of 4×32 -bit ring buffer FIFO and one cycle delay of the conditional branch, we achieved a much higher maximum frequency to improve its scalability against shared caches. Finally, we have 20% maximum frequency improvement and up to 17% performance improvement compared to private and shared caches on average.

Finally, we summarize in chapter 7 the main research contributions of the present thesis work and future research direction.

Chapter 3

Evaluation of state-of-the-art Instruction Caches in PULP

In this chapter, we adapt the three state-of-the-art instruction caches to our target PULP cluster. To give a comprehensive summary of the characteristic of each cache, we run the RTL simulation of the target cluster with the three caches. By varying the configurable parameters in terms of cache size and cache set-associate, we give the best configuration for each iCache for further exploration.

To efficiently analyze and assess the pros and cons of all architecture, we developed a programming environment for efficient data-parallel acceleration based on the OpenMP programming model. This software environment allows us to accurately control the instruction size to create synthetic tests to stress a specific corner case and assess the best and worst operating conditions for the three iCaches. After finding the best corner for each cache, we further validate the caches with real-life applications, including signal processing and CNN kernel-based parallel program with OpenMP.

3.1 Software and Program paradigm

To fairly measure the performance and energy efficiency, we fix the PULP cluster with 8 cores and change only the instruction cache (shown in Table 3.1). By varying the cache set-associate and cache size, we can find the best operating configuration for our target ULP cluster. In Table 3.1, the private instruction cache features N

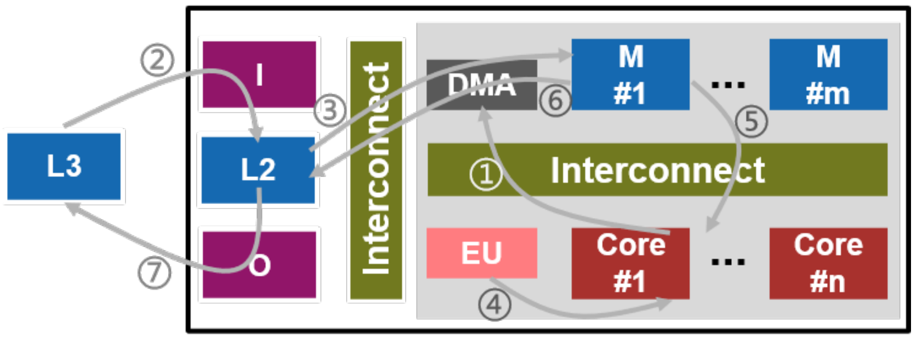
Mnemonic	Type	Hit	L2	Description	Set-associate
		Cycles	Penalty		
PR	Private	1	15	N Bytes I$ bank x 8 cores	W-way
SP	Shared	$\geq$1	17	8 x N Bytes I$ banks, 1-port	W-way
MP	Shared	1	19	2 x M Bytes I$ banks, 8-port	W-way

Table 3.1: Instruction cache architecture configurations

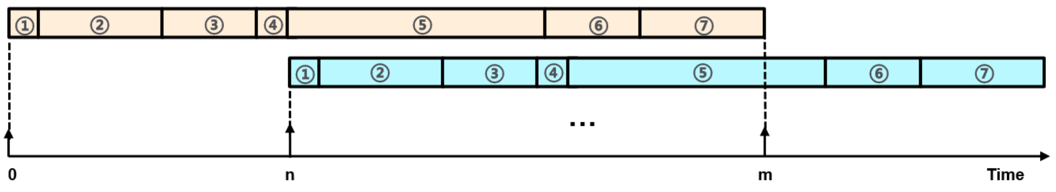
Bytes cache bank for 8 cores, where N can be 512 or 1024, and the same N is used for the single-port shared cache. Therefore, there are two combinations of the total memory of 4KB and 8KB. While for multi-ports shared cache, we choose 2 banks with M Bytes (M can be 2048 or 4096 to have a total of 4KB or 8KB memory) because it has no memory access congestion. Besides, all caches' set-associate W varies among 1-way, 2-way, 4-way. In conclusion, we have six combinations for each cache to make a detailed comparison and find the best combination for the target ULP cluster. Finally, since the PULP has no data cache to avoid redundant data copies with explicit data copy, we use the following multi-core program methodology to activate its computing capacity fully.

3.1.1 Program methodology

Fig. 3.1 shows the basic program sequence with explicit data transfer in PULP. First, the cluster's cores execute the instruction load from L2, and the master core (can be any one of the cores) initiates a DMA transfer to load task data from L2 to cluster's TCDM and go to sleep. Second, if the data size is large and does not fit the L2 size, an external L3 memory device is used to load data to L2 with the help of SoC I/O. Next, the DMA is responsible for transferring the data from L2 to TCDM. Once it's finished, it sends a finished event to the event unit, and the event unit wakes up the cores in step 4. Then, the cores start processing the data in parallel with the help of OpenMP in step 5. In the end, once all the cores finish data processing, the master core asks the DMA to copy the data back to the L2, and I/O copies it back to the L3 in steps 6 and 7 (Fig. 3.1a). The data transfer is in the pipeline (Fig. 3.1b) with the help of several DMAs while the cores are processing the data to hide the latency of huge memory transfers (shown in step 2, 3) and to maximize the performance.



(a) Notion of program sequence



(b) Data transfer pipeline to hide huge data transfer latency, at least save (m-n) time for the second data processing

Figure 3.1: Program sequence with PULP cluster for parallel acceleration

Thanks to the ultra-low-latency lightweight DMA, it supports multiple outstanding transactions required to hide the huge latency of external memory accesses without a large internal memory for the temporary storage of incoming or outgoing data [91]. Based on the pipelined DMA transfer program mythology, an automatic tool to deploy DNNs on low-cost PULP MCUs with typically less than 1MB of on-chip SRAM memory - DORY is proposed [burrello2020dory]. DORY abstracts tiling as a Constraint Programming (CP) problem: it maximizes L1 memory utilization under the topological constraints imposed by each Deep Neural Network (DNN) layer. Then, it generates ANSI C code to orchestrate off- and on-chip transfers and computation phases. The generated C code helps users to complete the work described in Fig.3.1. Finally, we combine the generated C code with our programs to improve programming efficiency and hide the complex explicit data transfers for DNN applications.

3.1.2 Synthetic tests

The synthetic tests are simply doing parallel vector multiply for 8092 data. We use Loop unrolling algorithm 1 to control the instruction size of the synthetic tests accurately. The $BUFFER\_SIZE$ is always fixed to 8192 while $STEP$ changes among 32, 64, 128, 256, 512 and 1024 which represent instruction size for 0.375KB, 0.75KB, 1.5KB, 3KB, 6KB and 12KB respectively. Finally, we run the RTL simulation of the 6 tests, and all the tests have the same number of operations for cluster cores.

Algorithm 1 Loop Unrolling

```
Require: $BUFFER\_SIZE = 8192$
Require: $TOTAL\_CORE = N$
Require: $STEP = M$
  $start \leftarrow core\_id \times BUFFER\_SIZE\ /\ TOTAL\_CORE$
  $end \leftarrow start + BUFFER\_SIZE\ /\ TOTAL\_CORE$
  for $i = start; i < end; i+ = STEP$ do
      $c[i] \leftarrow a[i] \times b[i]$
      $c[i] \leftarrow a[i+1] \times b[i+1]$
      ...
      $c[i + STEP - 1] \leftarrow a[i + STEP - 1] \times b[i + STEP - 1]$
  end for
```

3.1.3 Benchmarks

Furthermore, we used a series of benchmarks based on full-fledged optimized OpenMP implementation [69] applications and four CNN applications which make use of the Auto-Tiler library, a CNN library that manages large data transfer between SoC domain and Cluster domain automatically for the embedded system [37] to analyze in-depth the behaviour of each architecture. Each application features a different behaviour in terms of access patterns to the instruction memory subsystem and diversified memory footprints and execution time.

The detailed characteristics of each application are shown in table 3.2, including each code section size in KB and each number of 32-bit instructions in execution. Since cache performance is influenced strongly by code locality and code size, we

APP	Size [KB]	Class	Description
BFS	59.2	Short-Jump	Breadth-First Search
CT	28.2	Long-Jump	Color Tracking
FAST	28.6	Long-Jump	Machine-generated corner detection algorithm
SLIC	26.1	Long-Jump	Simple Linear Iterative Clustering
HOG	33.3	Library	Histogram of Oriented Gradients
SRAD	31.6	Library	Speckle Reducing Anisotropic Diffusion
FFT	41.2	Library	Fast Fourier transform
CIFAR10	37.5	CNN	Object Recognition
MNIST	37.9	CNN	Handwritten digits Recognition
KWS	30.1	CNN	Key word spotting
CNNDronet	71.3	CNN	Detector for Real-Time UAV Applications

Table 3.2: Benchmark details

classify the applications into 3 groups. The Short-Jump class includes *BFS* and *CNN* kernel-based applications, which are loop-based applications with most loop bodies smaller than four cache lines. The Long-Jump class groups all the loop-based applications with loop bodies greater than four cache lines or based on extensive control flow instructions, including *CT*, *FAST*, and *SLIC*. In the end, the Library class contains *HOG*, *SRAD*, and *FFT*, which use libraries to manage non-native data types, such as float, and fixed-point arithmetic, generating significant stress in cache [34].

All the above applications are reasonably complex and long-lived (millions of instructions in most cases), so it takes too much time to explore with RTL simulation. For this reason, we analyze the performance based on measures coming from RTL-equivalent FPGA implementations, mapped on Xilinx Zynq ZCU102 FPGA using Vivado 2019.2. The FPGA emulation allows executing at up to 50 MHz, enabling near-to-real-life execution time. The performance analysis is based on statistics collected by hardware counters implemented inside the instruction cache. Then we calculate the miss rate of each application and use the miss rate to power LUT to estimate the absolute power and energy efficiency for each cache architecture.

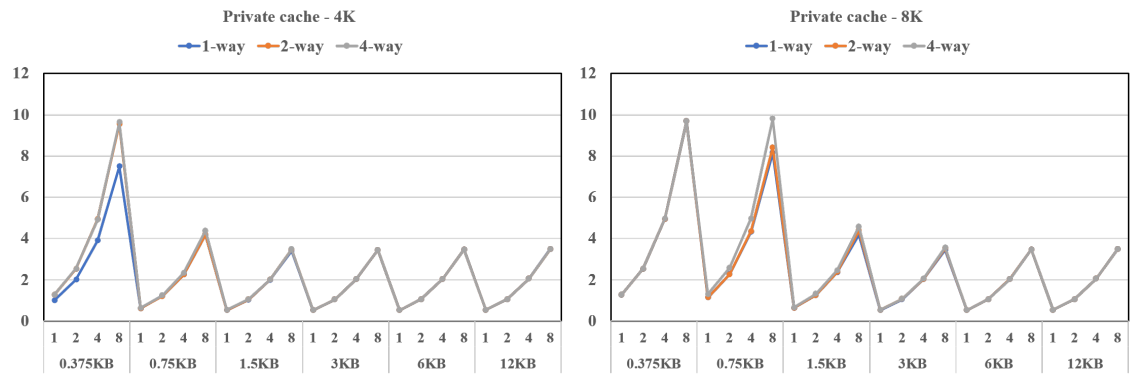
3.2 Performance Results

3.2.1 The performance of synthetic tests

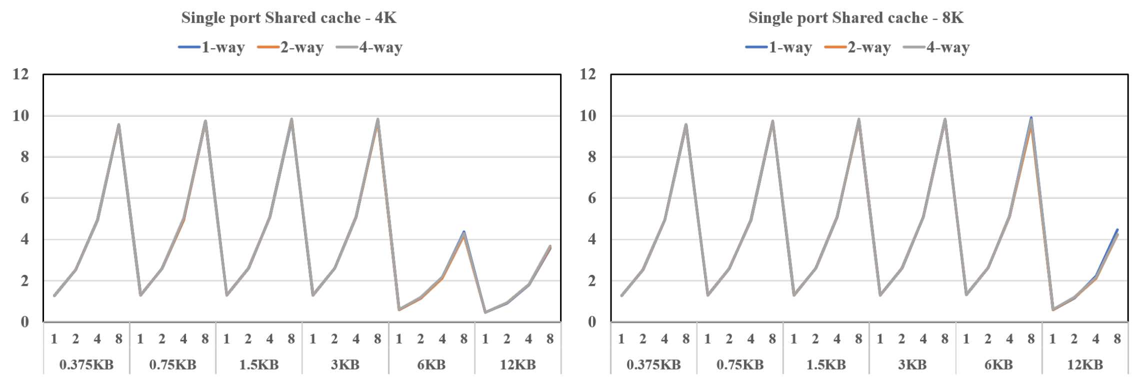
We simulate the above 6 synthetic tests with the same number of operations on different cache architectures with configurable cache set-associates and cache sizes. At the same time, we also simulate the tests with different cores (1, 2, 4, 8 cores) to address our target parallel multi-core architecture.

The performance of each configuration of simple private cache is shown in Fig. 3.2a. All the data is normalized to private cache running with the baseline configuration - 0.375KB instruction size, one-way, and single core. We use PR-4K-1W-1C-0.375KB to express it for easy understanding, and it is the same for shared caches. So it is expected that PR-4K-4W-8C-0.375KB achieves a 10 times speed-up compared with the baseline. The left is the private cache with 0.5KB cache capacity, which means 4KB in total (PR-4KB). The private cache with 1KB cache capacity is on the right, which means 8KB in total (PR-8KB). i), We can see that the performance decreases according to the instruction size's increase. There is a performance drop when instruction size crosses the boundary of cache capacity. For PR-4KB, when the instruction size of 0.75KB is large than the 0.5KB cache capacity, there is about 40% performance drop for the same n-way set-associative. The performance drop enlarges until 50% with the augmentation of instruction size. With the significant cache capacity miss, PR-4K-1W-1C-12KB only has half the performance of PR-4K-1W-1C-0.375KB.

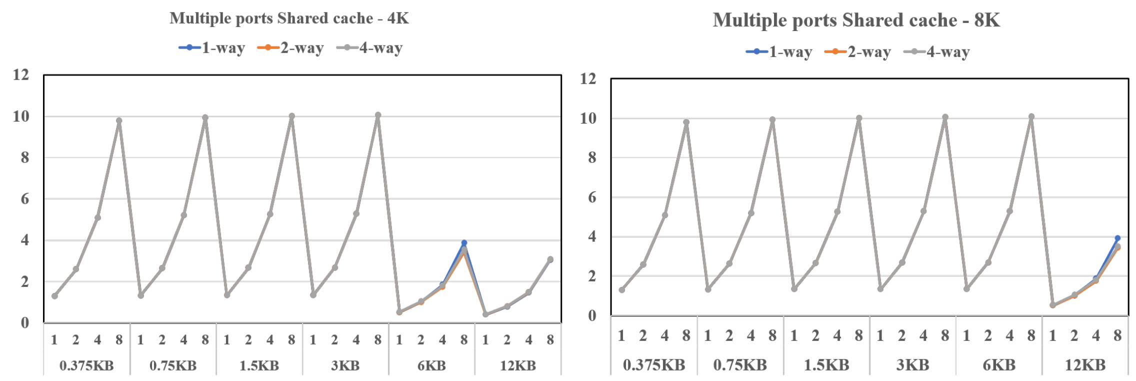
Secondly, We summarize the performance improvement brought by multiple cores in a parallel architecture. We can see about 2x, 3.8x, and 7x performance improvement when running with 2, 4, and 8 cores on average, respectively, compared with the performance running with a single core for PR-4K-1W-0.375KB. It is a significant improvement of thread-level parallelism by simply increasing the number of small cores. However, the performance improvement is not linear. There are two main reasons. First, when the program allocates the resource for each core and synchronizes among the cores, it introduces extra code and leads to a performance drop. Second, the private caches have L2 refill miss congestion when the instruc-



(a) Private cache



(b) Single-port shared cache



(c) Multi-port shared cache

Figure 3.2: Comparison of three types of instruction cache architectures' performance. 1) The core number varies from 1 to 8. ii) The cache is among 1-way, 2-way, 4-way set-associate. iii) The instruction size varies from 0.375KB to 12KB. All the results are normalized to the performance of the 1-core, 1-way private cache with 0.375KB instructions.

tion size exceeds the cache capacity, especially for PR-4K-12KB. We can see that the performance improves only 1.9x, 3.5x, and 3.5x when running with 2, 4, and 8 cores, respectively.

Thirdly, we compare the relationship between performance and set-associate. For PR-4KB-0.375KB, we can see that 2-way or 4-way set associate private cache achieves the best performance, improving about 30% on average compared with the 1-way set associate private cache. This is because when cache capacity is relatively small, there is more conflict miss. However, when compared 4-way with 2-way set-associate private cache, there is slight performance improvement, less than 0.5% for PR-4KB. PR-8KB is the same except for PR-8KB-0.75KB, where 4-way set-associate cache improves about 15% compared with 1-way and 2-way set-associate cache.

The performance for each configuration of shared caches is shown in Fig. 3.2b and Fig. 3.2b. All the data is normalized to PR-4K-1W-1C-0.375KB. Firstly, the shared caches have 8x the private cache capacity. For SP-4KB and MP-4KB, we can see about 1.8x, 2.5x, and 2.53x performance improvement on average when running with 0.75KB, 1.5KB, and 3KB instruction size, respectively, compared with PR-4KB. It is the same performance improvement for 1.5KB, 3KB, and 6KB with SP-8K and MP-8K, which means that shared caches are suitable for larger applications. However, we can still see the same performance drop after 3KB instruction size for SP-4KB and MP-4KB and after 6KB instruction size for SP-8K and MP-8K, which drops about 50% on average because of the large cache capacity miss.

Secondly, With parallel acceleration, the performance of shared caches still improves 1.9x, 3.5x, and 7.5x running with 2, 4, and 8 core on average, respectively, when the instruction size is less than the cache capacity. Thirdly, when the instruction size exceeds the shared caches' capacity with large cache capacity miss - SP-4KB and MP-4KB running with 6KB and 12KB instruction size, we can see that MP-4KB results in a performance drop of about 11.8% compared with SP-4KB on average. It is the same for SP-8KB and MP-8KB running with a 12KB instruction size. The reason is that the two-level structure multi-port cache has two more cycles of L2 miss penalty than the single port shared cache ($(19-17)/17 \approx 11.8\%$ according to Table 3.1).

In conclusion, for performance comparison, the shared cache with large cache capacity can adapt to tests with wide instruction size with less cache capacity miss.

The set-associate cache can solve the cache conflict miss. However, it has limited performance improvement. Therefore, 2-way or 4-way cache is suitable for our target cluster platform.

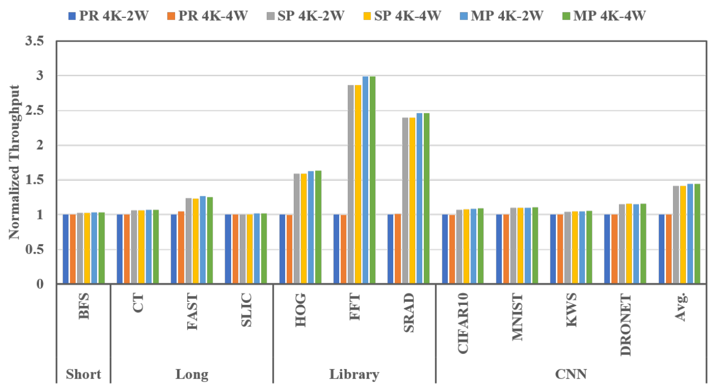
3.2.2 The performance for real-life applications

We simulate the above complex applications with different behaviour in terms of access patterns to the instruction memory. The result can give us a comprehensive and direct view of each cache architecture and help to fix the best choice in our target cluster system. We still vary the cache set associate and cache size. However, with the conclusion of the synthetic tests, we take only 2-way or 4-way set-associate and run with all 8 cores to fully take advantage of parallel acceleration with 4KB or 8KB cache size.

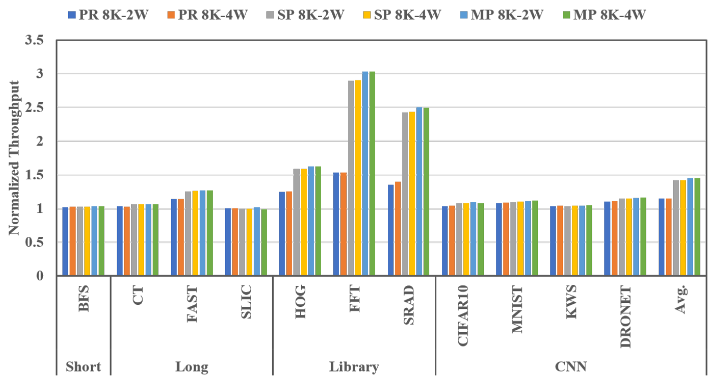
Fig. 3.3a and 3.3b show the performance results for all caches with 4KB and 8KB cache capacity normalized to the private cache with 2-way cache set-associate (PR-4K-2W) for each application. Firstly, we note that the set-associate influences little on the cache performance, the caches with 4-way set-associate improve the performance by about 1% on average compared to the caches with 2-way set-associate.

Secondly, caches with larger cache capacity always achieve better performance since they reduce the miss rate as shown in Fig.3.3c). Especially for private cache, the miss rate of HOG, FFT and SRAD decrease about 50% when using PR-8KB, thus the performance of PR-8K increases about 50% for the two applications compared to PR-4K. However, for shared caches, the performance improvement is trivial (about 1%) since all application's miss rate is closely to 0.

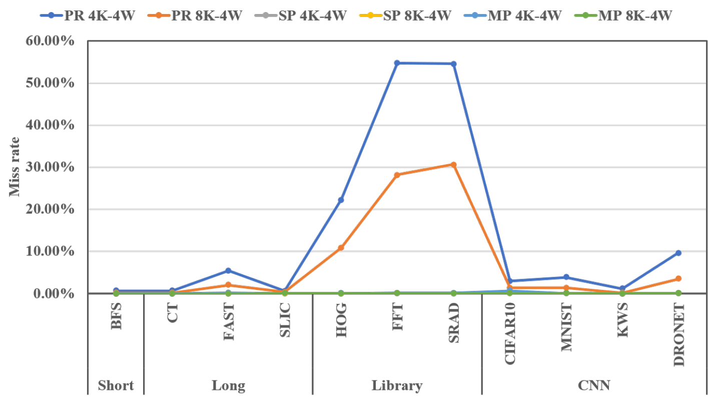
Thirdly, the larger the miss rate of the applications for private cache, the large the performance improvement of shared caches. For applications with a low miss rate (less than 1%), including BFS, CT, SLIC and KWS, the performance improvement of shared caches is about 5% on average. For applications such as FAST (5.4%), MNIST (3.8%), CIFAR10 (2.92%), DRONET (9.3%), the performance improvement of shared caches are about 12% on average. For the high miss rate applications such as HOG (22.9%), FFT (54.7%) and SRAD (54.5%), the performance improvement of shared caches can be 2.5x - 3x on average because of the less L2 refill penalty.



(a) Performance with 4KB cache capacity



(b) Performance with 8KB cache capacity



(c) Miss rate of cache with 4-way set-associate

Figure 3.3: Comparison of three types of instruction cache architectures' performance. 1) The core number is fixed to 8. ii) The cache is among 2-way, 4-way set-associate. iii) For each application, its results are normalized to the performance of PR-4K-2W.

In conclusion, larger cache capacity and 4-way set-associate can increase the system performance. However, shared caches have limited performance improvement since their miss rate is close to 0. Therefore, a 4KB cache size cache is suitable for our target cluster platform. Besides, a 4-way set-associate is employed to ensure stable performance even if caches with smaller associativity consume less power and energy.

3.3 Results of physical implementation

This section presents a comprehensive physical exploration of the area, timing, and power for all the configurations shown in Table 3.3 to characterize the power and energy efficiency of alternative instruction cache architectures. All caches have 4KB cache capacity and 4-Way set-associate to ensure the stable performance with the small area shown in section 3.2.2.

Mnemonic	Type	Refill Cycles	Description	Set-associate
PR	Private	15	512B I$ bank × 8 cores	4-Way
SP	Shared	17	8x 512B I$ banks, single-port	4-Way
MP	Shared	19	2x 2048B I$ banks, 8-port	4-Way

Table 3.3: Instruction Cache Architectures Explored in this work.

3.3.1 Area and Timing results

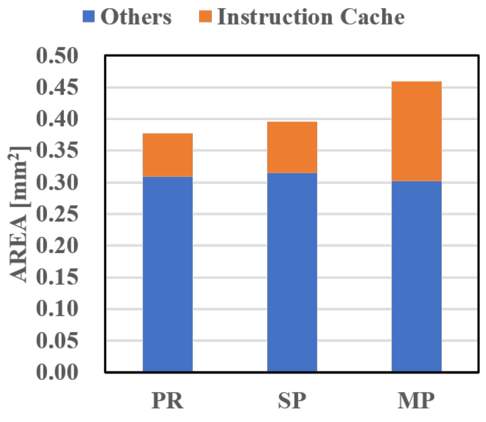


Figure 3.4: The 8-core cluster area breakdown for the different cache architectures and configurations. Instruction cache area contribution is placed on top.

One of the main points for designing a shared cache is to provide a large capacity with a small area overhead. Fig. 3.4 illustrates the silicon area costs for all cache architecture configurations. Thanks to the private cache's simpler structure, it is smaller than other caches. *SP* has little area increase compared to *PR* and *MP* has the largest area that is more than twice the cache area of *PR* because of the multiple reading ports. As a result, *MP* has poor scalability when the number of the cores increases.

Type	Cluster Maximum frequency [MHz]		Speed up compare with *PR* [%]	
	8-core	16-core	8-core	16-core
PR	378	363	0	0
SP	350	320	-7	-12
MP	357	306	-5	-16

Table 3.4: Timing result

Table 3.4 shows the results of the static timing analysis for all the cache architectures, implemented in clusters of 8 cores or 16 cores to emphasize the timing issue of the shared caches. Not surprisingly, the clusters featuring shared caches (*SP*, *MP*) have worse maximum frequency due to the long critical path between the core and the interconnect and the high congestion of the multiple reading ports, respectively. For a system with 8 cores, the caches with simple L1 private cache achieve the best timing, keeping about 6% timing improvement compared to shared caches. When we increase the system core number to 16, we observe that *MP* and *SP* result in about 12% and 16% maximum frequency drop respectively compared to *PR*. For the *MP*, 16-port memory banks cause serious wire congestion, leading to worse timing results. It is the same issue for *SP*, with more channels and more levels logarithmic interconnection, the critical paths get worse with interactions of *request* and *response* channels.

3.3.2 Power results

Obtaining the power of the whole cluster system with different instruction cache for real-life applications running for millions of clock cycles on a post place and route database is not feasible, both due to the long simulation time and size of the VCD traces required to annotate the switching activity of the design. What's more, real-life applications often have complex and unpredictable behaviours, making it difficult to understand the functional and power behaviour of the cache. To model the presented caches and provide more insight into their power and energy consumption behaviour, we propose a methodology based on a synthetic benchmark where we artificially modulate the instruction locality.

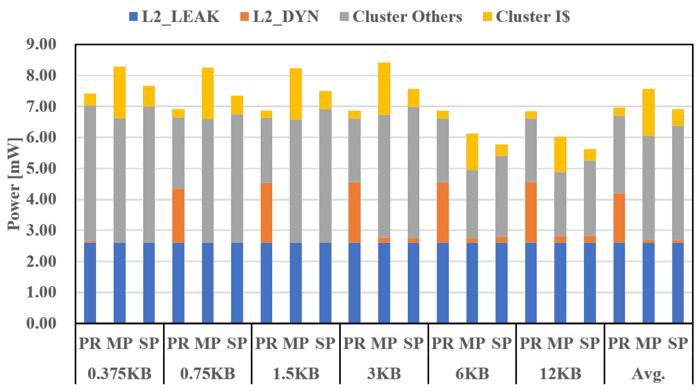
Moreover, since the applications executed by the cores have an IPC close to 1, the only stalls of the cores are those caused by cache misses, decreasing the whole cluster system's activity and reducing its power consumption. On the other hand, the power consumption of the L2 memory augments with the miss rate due to the increasing number of refills. In order to take into account this significant contribution, we characterized the energy consumption of every refill from L2 and the overall L2 leakage power and added it to the system power. We used this methodology to characterize the behaviour of the seven architectures, summarized in equation 3.1. We obtain the parameters $L2\_leakage\_power$ and $L2\_per\_read\_energy$ from the L2 SRAM's datasheet. The parameters $cycles$ and $L2\_miss\_refill\_number$ can be read from the hardware counters implemented inside the instruction cache with the cycle-accurate simulation. We only need to find the $cluster\_power$ with a specific system frequency to have the total energy.

$$\begin{aligned}
&Total\_energy \\
&= (cluster\_power + L2\_leakage\_power) \cdot time + L2\_read\_energy \\
&= (cluster\_power + L2\_leakage\_power) \cdot cycles/frequency \\
&+ L2\_miss\_refill\_number \cdot L2\_per\_read\_energy
\end{aligned} \tag{3.1}$$

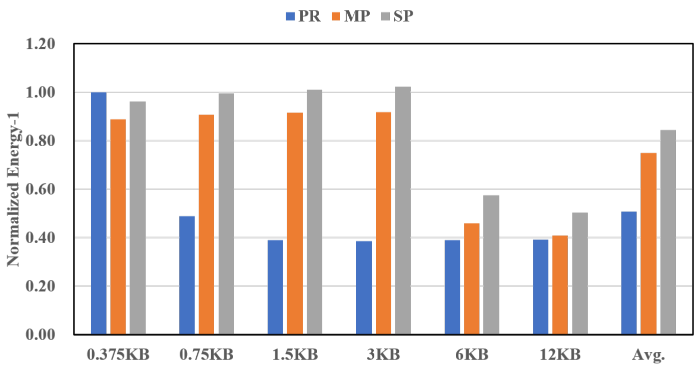
3.3.2.1 The power and energy efficiency of synthetic tests

The power of all synthetic tests measured on different architecture configurations is shown in Fig. 3.5a. The *PR* configuration is the one consuming the least power in all configurations, thanks to the simple and straightforward implementation for loop body below 0.5 KB. For larger loop bodies, the power of the *PR* configuration drops due to the high miss rate. As we can see, the cluster power, including I$ power, decreases while the L2 dynamic power increases largely since the cores stall and are waiting for L2 refills. In general, *SP* features the smallest power consumption thanks to its relatively small area and large cache capacity.

Fig. 3.5b demonstrates the energy efficiency for each cache normalized to PR-0.375KB. On average, the *MP* and the *SP* have about 24% and 33% energy efficiency improvement respectively compared to the *PR*. Besides, they can attain twice the energy efficiency of *PR* for loop body between 0.5 KB and 4 KB. However, for loop body below 0.5 KB, the *PR* still has at least 4% better energy efficiency than the shared caches, and it is suitable for most of the applications with small for loop bodies.



(a) Power

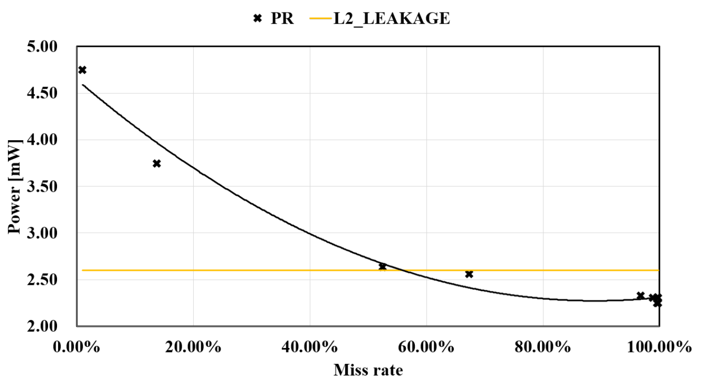


(b) Normalized Energy efficiency

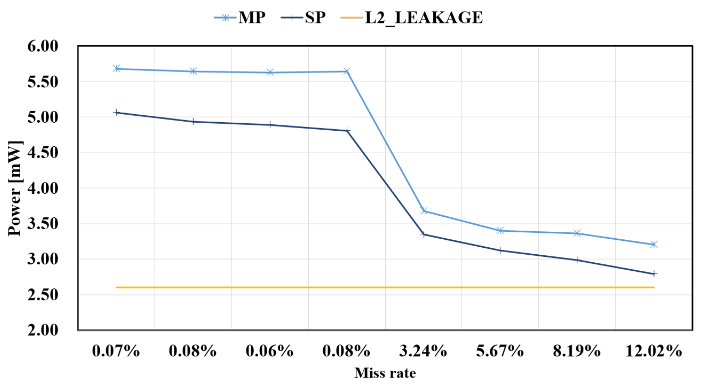
Figure 3.5: Power, energy efficiency of the synthetic tests normalized to *PR* with 0.375 KB cache size, 200MHz.

3.3.2.2 Power model - Look-Up Table

Since the Fig. 3.5 shows a significant dependency between the power consumption of the cluster and the miss rate of applications caused by the cores' stalls. Hence, in order to model the power consumption of the different cache architectures, we created a high-level model according to the relationship between the power consumption of the cluster and the miss rate to calculate the cluster power rapidly. Fig. 3.6 shows the relationship between miss rate and cluster power. We separate the schema into private and the shared caches.



(a) Private cache.



(b) Shared caches

Figure 3.6: $Power_{Average} \propto (1/MissRate)$ inverse linear regression, 200MHz.

In Fig.3.6b, we separate the function into two parts - miss rate is below and larger than 1%. When the miss rate is lower than 1%, the cluster power almost remains stable since the IPC is close to 1, and the cluster system is fully active. Nevertheless, when the miss rate is larger than 1%, it shows the inverse linear regression. Besides, the maximum miss rate of shared caches is below $100\%/8 = 12.5\%$, thanks to the

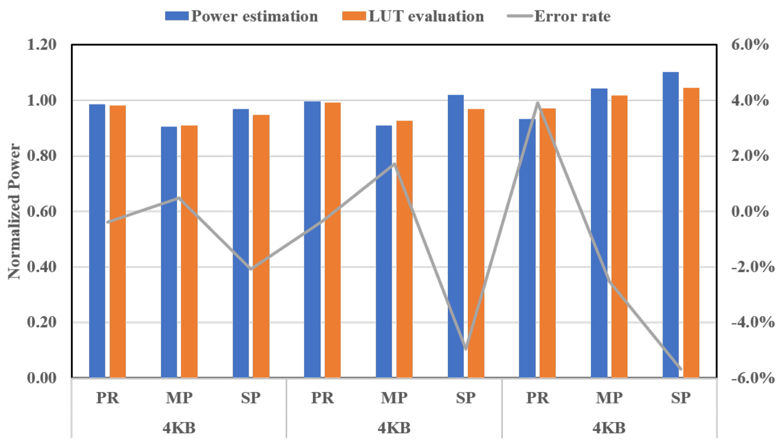


Figure 3.7: Error rate between power estimation and model evaluation, the power results are normalized to *PR*, 200 MHz.

parallel computing architecture. In the end, when each cache reaches the maximum miss rate, its power remains stable, containing the static leakage power and stable cache L2 read power.

To validate the power model, we analyze the error rate between LUT and exhaustive power estimation on three of the smallest real-life applications considered in this work. We selected several relatively small applications and estimated the error rate between power estimation using PrimeTime and LUT evaluation model, with a maximum error smaller than 6% (Fig. 3.7). Since we only care about the relationship of each cache, all power results are normalized to PR. Finally, we use this accurate power estimation to make a detailed comparison. This model poses the basis to evaluate larger real-life applications that cannot be executed in a reasonable time on post-layout netlists.

3.3.2.3 The power of real-life applications

The energy efficiency for each cache normalized to PR is demonstrated in Fig. 3.8. On the left, for the low miss rate applications, the *MP* always has the worst energy efficiency than *PR* and *SP* due to its large area. Then, since the miss rate is low, the *PR* benefits from its simple architecture and attains the best energy efficiency in BFS, CIFAR10, KWS, CT and SLIC. However, on average, the *SP* achieves the best energy efficiency since it has a larger cache capacity with little extra area compared

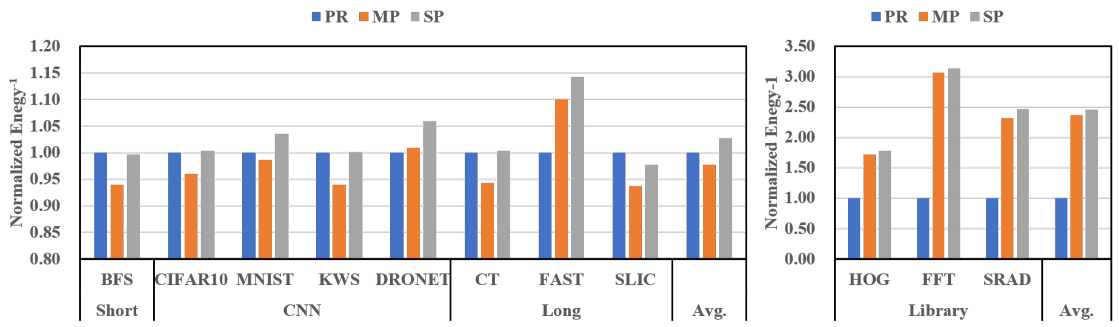


Figure 3.8: Energy efficiency of the applications normalized to *PR*, 200MHz. The low miss rate applications in the left and the high miss rate applications in the right.

to *PR*. It is the same for the high miss rate applications on the right for the *SP*. Now, the shared caches‘ energy efficiency is about 2.4× as large as *PR*'s because of 8× capacity.

3.4 Conclusion

In this chapter, we explored three different instruction cache architectures for energy-efficient and cost-effective tightly coupled clusters of processors for end-node IoT devices, including one traditional private cache and two shared caches (one featuring a crossbar between the processors and the memory banks, and one exploiting memory banks with multiple ports). All the designs are based on latch-based memories instead of SRAMs to purchase low power and high energy efficiency. We conducted an exploration running a series of synthetic tests with loop-unrolling, several signal processing and CNN based applications featuring diverse instruction memory access patterns on the same cluster, configured with different instruction cache architectures. The results reveal that the shared caches can execute in a more energy-efficient way for a much wider class of applications with much less silicon area compared to private caches. When executing the high miss rate applications, they can bring up to 3× performance and 2.5× energy efficiency improvement compared to private cache. However, due to the logarithmic interconnect for single-port shared cache and multi-port memory banks for multi-port shared cache, they all suffer from critical timing issues, leading to scalability when the number of cores increases.

Chapter 4

Two-level instruction cache

4.1 Overview

Many modern processors utilize multiple levels of cache, with small, fast primary caches backed up by larger, slower caches. The usage of second-level cache or multi-level cache compromises cache capacity (hit rate) and cache access latency. In chapter 3, we have proven that the larger cache capacity, the better the hit rate in multi-core clusters. Nevertheless, with the same technology, the larger the cache capacity, the larger the memory access time leading to more processor pipeline stages [6]. Therefore, the first level cache is often on the critical path of simple single-issue microprocessors featuring a flat pipeline [79]. Moreover, DSP-oriented processors designed for high energy efficiency typically feature a prefetch buffer to support compressed instructions and reduce the pressure on the memory hierarchy (low power) and even more complex instruction fetch stages to avoid stalls (high efficiency). However, this extends the critical path towards the instruction cache [34]. Finally, this path is further extended in shared caches due to the need for an additional crossbar or multiple ports [66] towards the cache banks. These considerations, joint with the well-known routing bottlenecks for deep sub-micron technology nodes, potentially pose significant limitations on future software-programmable parallel processor clusters' performance, energy efficiency, and scalability.

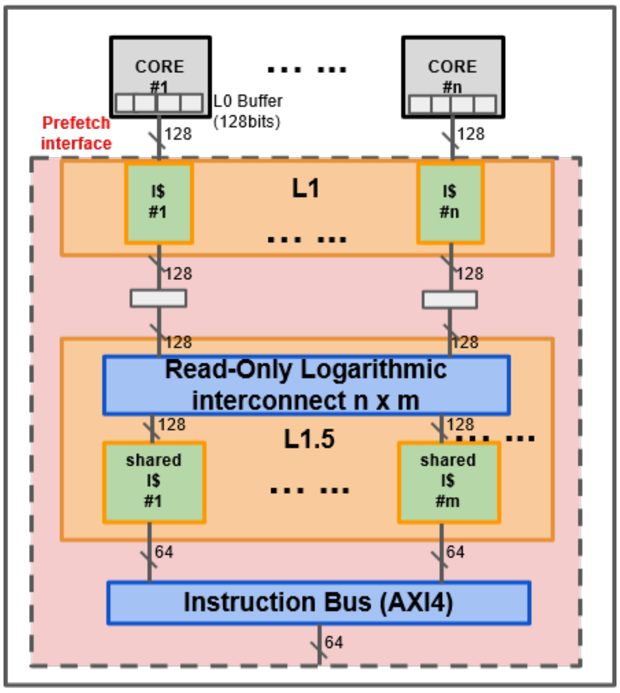
The main contributions of this work are: firstly, we propose a two-level instruction cache architecture that combines the small, fast private cache (L1) with one cycle access latency with large single-port shared cache (L1.5) with two-cycle access latency through the logarithmic interconnect in tightly coupled CMP cluster; secondly, we analyze the two-level cache and compare it with the above three main cache architecture from the aspect of performance and energy efficiency with the synthetic tests and real-life IoT applications.

The rest of the chapter is organized as follows: The details of our two-level cache are described in section 4.2, and the comparison results with private cache and shared caches with synthetic tests and application are shown in section 4.3, and section 4.4 concludes the chapter.

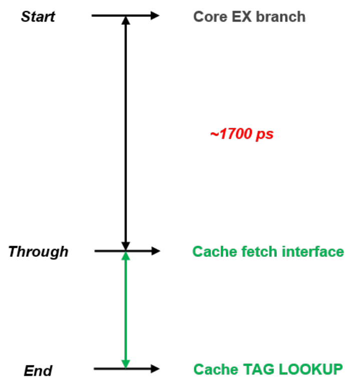
4.2 Architecture

In this section, we present a two-level instruction cache shown in Fig. 4.1a which combines small private instruction caches (Level 1) with a tightly coupled (1 clock cycles latency) shared instruction cache (Level 1.5). The two caches are connected through a single clock latency interconnect described in section 2.4.2.1. L1.5 shared cache provides a large capacity to reduce the miss rate of L1 private cache. Small L1 cache avoids long critical paths from the core to the interconnect and back to the core described in section 2.4.2 and benefits from the low-latency access time from the core to the L1 cache and from the L1 cache to the L1.5 cache.

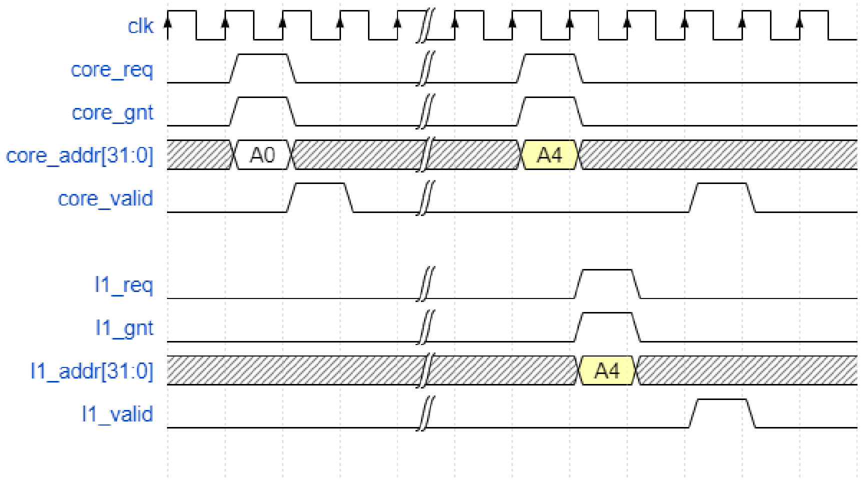
The fetch request is issued by the core's Instruction Fetch unit (IF), which first checks its L1 private cache. If there is a miss, a refill request is sent from the L1 cache to the L1.5 shared cache, crossing the low-latency interconnect instead of fetching in the L2. As described in section 2.4.2.1, the read-only logarithmic interconnect can increase the memory access latency. Thus the proposed hierarchical cache features a configurable request buffer and a response buffer (that's to say. they can be enabled with a System Verilog parameter) to reduce the interconnect latency. Since in the presented cluster implementation, the response buffer is sufficient to reduce the critical path, the request buffer has been disabled. On the other hand, this buffer is a powerful knob to improve the scalability of the system towards high-end clusters optimized for frequency or featuring a larger number of cores (for example, 16), taking advantage of the hierarchical structure of the cache.



(a) Two-level instruction cache. Level 1 is private cache and Level 2 is shared cache with only 2 cycles latency (including 1 cycle response buffer) when hit.



(b) L1 critical path, the same with private cache.



(c) Two-level instruction cache timing. The miss penalty for core is at least 3 cycles when miss in L1 and hit in L1.5.

Figure 4.1: Two-level instruction Cache subsystem

In the proposed implementation, the access time of the L1 cache is one cycle, of the L1.5 cache is two cycles (including one cycle for response buffer) when the cache hit separately (Fig. 4.1c). In the case of L.5 congestion, it can take more than two cycles. However, the congestion on the memory banks is largely reduced

Mnemonic	Type	Hit Cycles	L1.5 Penalty	L2 Penalty	Description	Set-associate
PR	Private	1	-	15	512 Bytes I$ bank x 8 cores	4-way
SP	Shared	≥1	-	17	8 x 512 Bytes I$ banks, 1-port	4-way
MP	Shared	1	-	19	2 x 2048 Bytes I$ banks, 8-port	4-way
HIER	Private	1	≥3	19	512 Bytes L1 I$ bank x 8 cores	4-way
	Shared	≥1	-	17	2 x 2048 Bytes L1.5 I$ banks, 1-port	4-way

Table 4.1: Instruction cache architecture configurations, including two-level cache.

by adopting a banking factor of 2 and the L1 filter cache.

Table 4.1 shows the hit cycles and miss penalty for each cache including the two-level cache (shown as *HIER*). Compared to private cache's 15 cycles of L2 miss penalty, two-level cache reduces it to about three cycles when hits in L1.5, which significantly ameliorates the performance. However, compared to SP, a two-level cache introduces another cache level, bringing additional area and power. It also leads to inefficient fetch with two more cycles when the instruction size is between L1 and L1.5 with a relatively low L1 hit rate according to the equation of $average\_fetch\_cycles = L1\_hit\_rate + (1 - L1\_hit\_rate) * 3$. It is even worse when compared with multi-port cache since *MP* has no refill congestion. However, the two-level cache's L2 penalty is the same with MP and leads to the worse performance with a very large instruction size compared to *SP*.

In order to analyse the performance and power of all the caches, we run the same synthetic tests and real-life applications described in section 3.1.2 and 3.1.3 to summarize in detail the characteristic of the two-level instruction cache.

4.3 Evaluation

Before comparing all the caches, we need to configure the two-level cache for each level. From the results of chapter 3, we choose the cache associativity between 2-way or 4-way, and cache capacity is fixed to 4KB. As shown in Table 4.1, we utilize a 4-way set-associate for both L1 and L1.5 cache to reduce miss rate and ensure performance. We use the same GF22FDX technology to implement the design and the same power model method to compare our proposal with previous work in detail.

4.3.1 Performance Results

4.3.1.1 The performance of synthetic tests

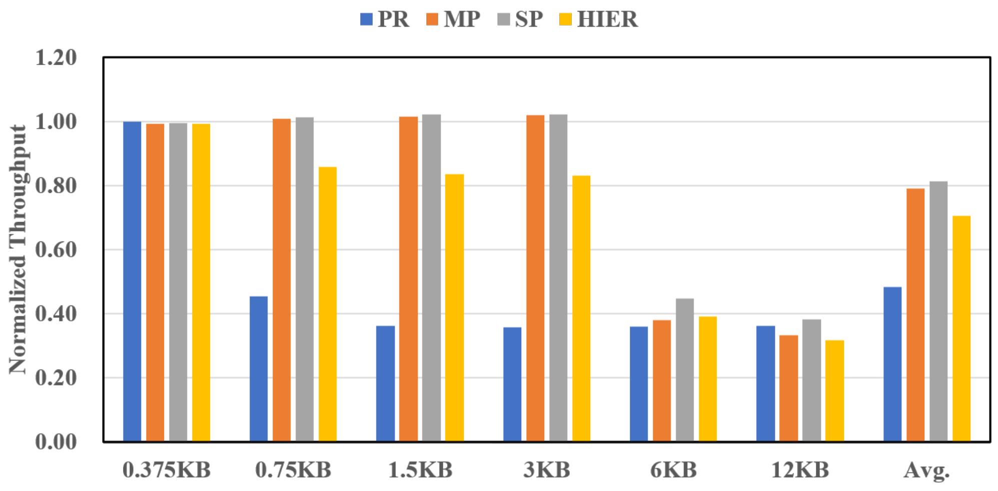


Figure 4.2: Throughput of the applications normalized to *PR* with 0.375 KB instruction size, 200MHz.

We summarize in Fig. 4.2 the throughput of all synthetic tests measured on the different architectural configurations normalized to *PR*-0.375 KB. There are four points that we can see. First, the performance drops significantly when over the limit of total cache capacities. We can see that for the *PR*, the performance drops dramatically starting from 0.75KB, which is the same for the shared and two-level cache. Second, two-level and shared cache architectures always show better or equal performance when compared to the related private configurations. We should notice that shared caches always have larger L1 cache capacities (8x) than private cache. Third, shared cache architectures always achieve better performance compared with two-level cache because the *HIER* has a smaller L1 cache capacity, and there are about two more refill cycles when there is a miss in L1 while hit in L1.5. So when the instruction size is between the L1 and L1.5 cache capacity, it loses about 16% throughput while the L1 miss rate is higher than 50%. In the end, when instruction size is much larger than cache capacity, such as 12KB, the *PR* attains better performance than *MP* and *HIER* since it has a smaller L2 penalty.

4.3.1.2 The performance of real-life applications

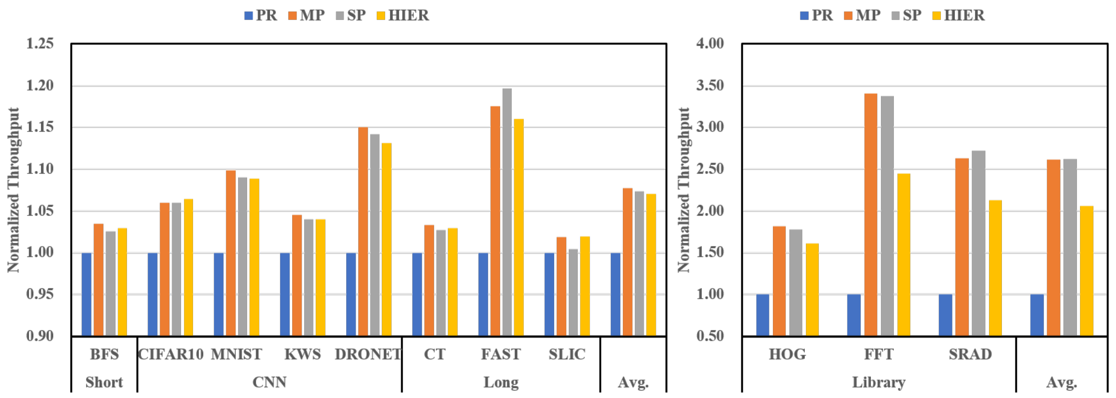


Figure 4.3: Throughput of the applications normalized to *PR*, 200MHz. the low miss rate applications in the left and the high miss rate applications in the right.

The throughput of each application for all caches normalized to the *PR* is shown in Fig. 4.3. On average, the flat shared cache configurations perform better than the two-level and the private caches, joining the benefits of a larger cache capacity with respect to the private, and a smaller L2 penalty with respect to the two-level cache (table 4.1) which causes a drop in performance of only 2%. However, the two-level cache reduces the *PR*'s performance drop and can compete with shared cache for the low miss rate applications. For the high miss rate applications, which are rarely in the IoT applications, the *HIER* has twice the performance of *PR*. Nevertheless, it still loses about 20% performance when compared with shared caches.

4.3.2 Physical Implementation Results

4.3.2.1 Area and Timing results

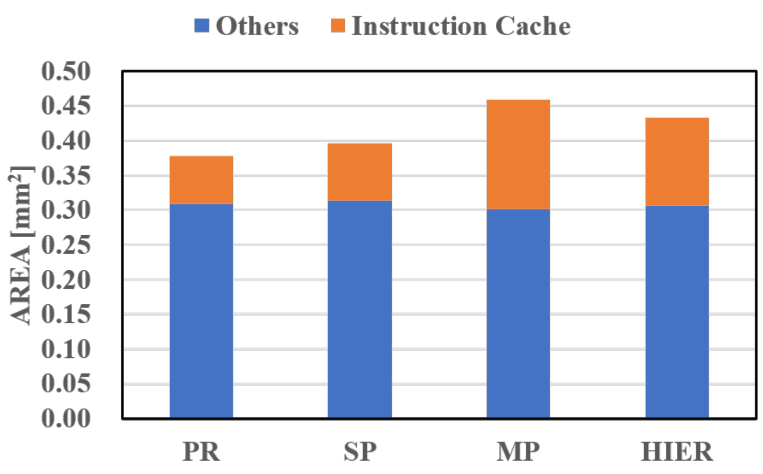


Figure 4.4: The 8-core cluster area breakdown for the different cache architectures and configurations. Instruction cache area contribution is placed on top.

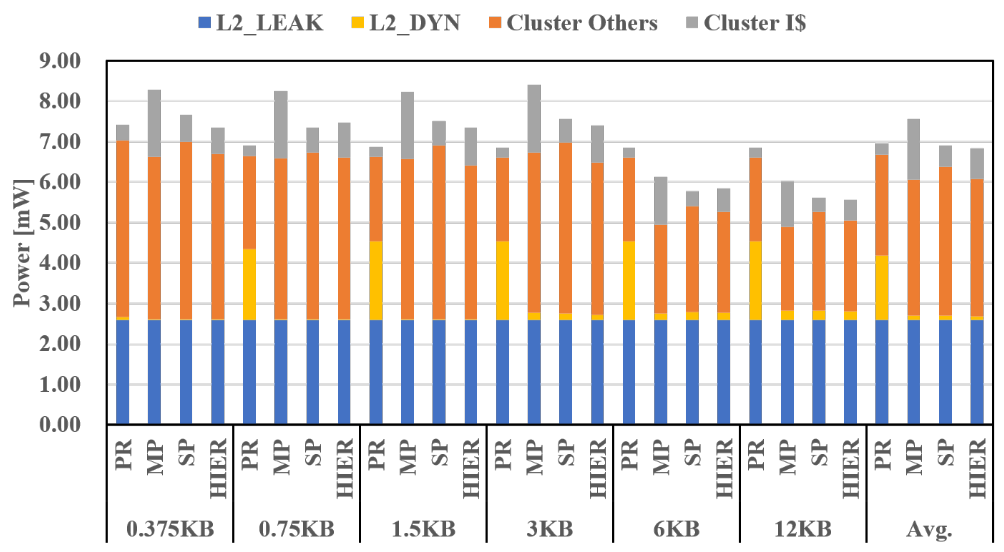
Fig. 4.4 illustrates the silicon area costs for all cache architecture configurations. There is no surprise that the *HIER* has a larger cache area than that of the *PR* and *SP* since the *HIER* has both L1 and L1.5. However, its area is still smaller than *MP* which demonstrates that the *MP* is only suitable for small cache capacity.

Type	Cluster Maximum frequency [MHz]		Speed up compare with *PR* [%]	
	8-core	16-core	8-core	16-core
PR	378	363	0	0
SP	350	320	-7	-12
MP	357	306	-5	-16
HIER	372	354	-1	-2

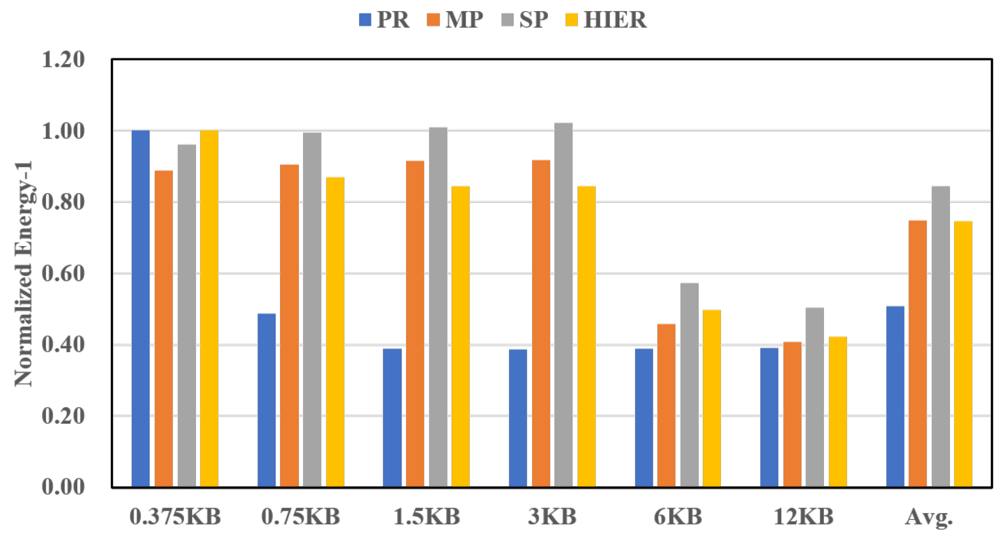
Table 4.2: Timing result

Table 4.2 shows the most important design target of the *HIER*. As expected, by benefiting from the small L1's simple and fast architecture, the *HIER* has almost the same maximum frequency as the *PR*. Thus, the *HIER* keeps the balance between scalability and performance.

4.3.2.2 The power and energy efficiency of synthetic tests



(a) Power



(b) Normalized Energy efficiency

Figure 4.5: Power, energy efficiency of the synthetic tests normalized to *PR* with 0.375 KB cache size, 200MHz.

Fig. 4.5a shows the power of all synthetic tests measured on the different architectural configurations. As we can see, *MP* still has the largest power while *PR* has the least power because of the design area. Then, the *HIER* has the smallest power since it benefits from L1's local fetch and L1.5's large capacity. From the view of L2 dynamic power, the *HIER* is more like a shared cache with much less L2 refills.

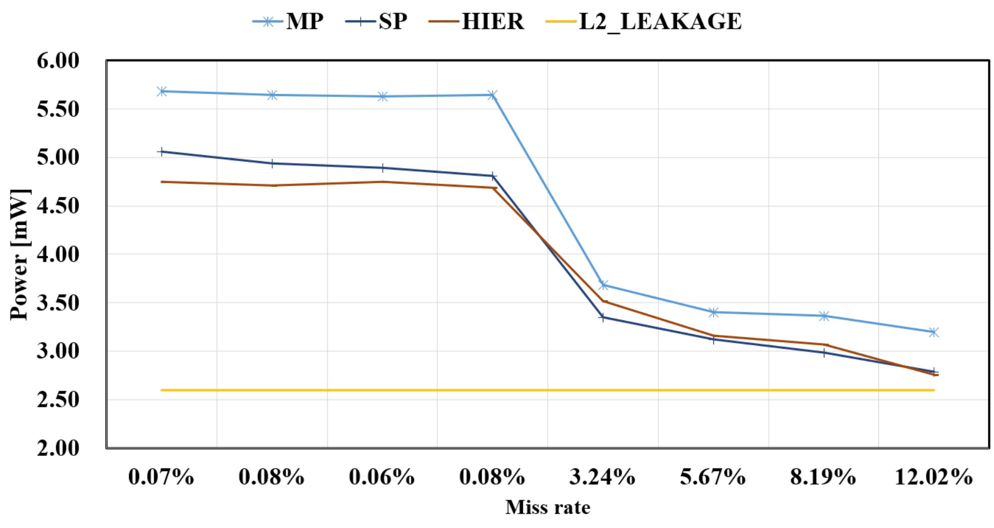


Figure 4.6: $Power_{Average} \propto (1/MissRate)$ inverse linear regression, 200MHz, updated with the *HIER*.

Fig. 4.5b shows the normalized energy efficiency of all synthetic tests measured on the different architectural configurations. There are two points that need to be noticed. First, on average, *HIER* improves about 20% the energy efficiency compared to *PR*. Second, the *PR* and *HIER* attain better energy efficiency than shared cache when the instruction size is less than the L1 cache capacity.

By exploiting the same power model method, Fig. 4.6 updated the LUT with the power of *HIER* running with synthetic tests. We can see that the *HIER* acts like a shared cache. Besides, it has less power than the shared caches when the tests' miss rate is low.

4.3.2.3 The power and energy efficiency of real-life applications

The energy efficiency of each cache normalized to PR is shown in Fig. 4.7. On the left, the low miss rate applications, the *HIER* has at least 7% energy efficiency improvement compared to other caches thanks to its large L1.5 capacity and simple L1 structure. Joining the architectural efficiency and power considerations, we can conclude that for most applications, featuring a miss rate lower than 5%. Very common in the near-sensor processing domain, the energy efficiency of the proposed two-level cache surpasses both the private and the shared caches. We should note that a significant energy saving for the shared cache configurations (both flat and hierarchical) comes from the merged refill requests to the energy expansive L2 memory. Compared to the *PR*, the *HIER* provides significantly more robustness with respect to applications with a large footprint and long branches. This can be clearly noted in SRAD, where the shared caches deliver more than 2x better energy efficiency than the private cache. For high miss rate applications, the *HIER* provides smaller performance (20%) than the flat shared caches due to a large number of L1.5 refills, leading to a 20% energy efficiency drop. However, it still has about twice the energy efficiency of the *PR*.

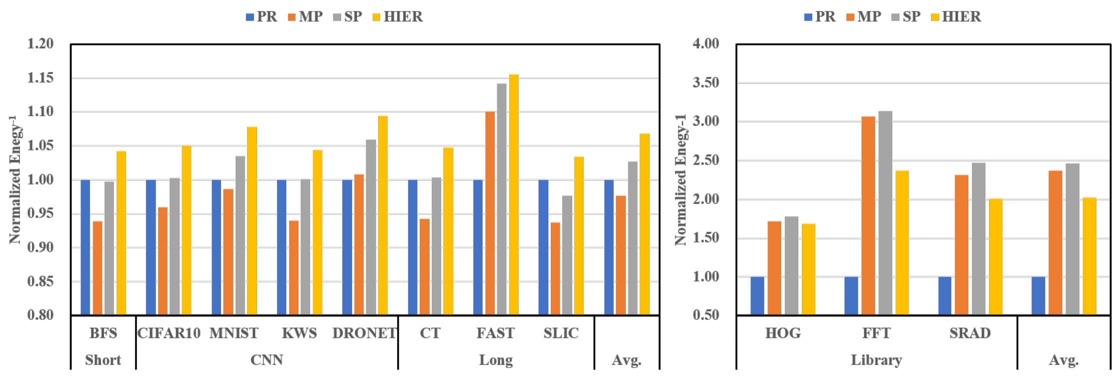


Figure 4.7: Energy efficiency of the applications normalized to *PR*, 200MHz. The low miss rate applications in the left and the high miss rate applications in the right.

4.4 Conclusion

In this chapter, we proposed a hierarchical instruction cache architecture for energy-efficient and cost-effective tightly coupled clusters of processors. Exploiting a small level-zero private cache tightly coupled to a single-clock latency L1.5 shared cache, the proposed architecture joins the benefits of private caches (low-power consumption) with large capacity typical of shared caches (execution efficiency). We benchmark the proposed architecture on a wide range of real-life IoT workloads, showing that for the low miss rate applications, the proposed architecture improves the energy efficiency by 10% with respect to private caches, and from 7% to 15% with respect to flat shared caches. In applications with a high miss rate, which is rare in IoT applications, the new cache organization still performs better than private caches with the same total cache capacity, thanks to larger shared L1.5. It has an acceptable performance loss with respect to the flat shared caches. In conclusion, the proposed cache provides strong advantages in implementation effort with respect to the flat shared caches, improving the scalability of multi-core systems both in terms of the number of cores per cluster and maximum operating frequency. Finally, more effect will be paid to solve the issue of high miss rate applications, such as adding prefetching features between L1 and L1.5.

Chapter 5

Prefething in L1 iCache

5.1 Overview

Prefetching is a well-known approach to mitigate the impact of cache misses in order to reduce the latency of memory operations in modern computer systems. The prefetching can be implemented in each level of the caches or inside the cores, such as modern Intel processors [111] which have four prefetchers per core, namely two L1 data cache prefetchers and two L2 prefetchers. In the chapter 2, we conclude that the correlation-based prefetching can largely improve the performance and solve the non-sequential issue. However, these prefetchers consume large metadata storage (more than 20KB) due to the vigorous instruction execution trace. The FDP-based prefetcher depends on the accurate branch prediction with the help of BTB with relatively small size around 7KB for industry products [45]. However, since our multi-core cluster features simple RISC-V core to do parallel acceleration with OpenMP, implementing each core with BTB may improve the non-sequential execution performance while bring large extra access power.

The main contribution of this work: Firstly, we employ a simple sequential next-line (4 instructions) prefetching with cache probe filtering (CPF) to hide the L1 to L1.5 latency without jeopardizing energy efficiency. We expose this feature to allow software-controlled enable/disable of the prefetcher to adapt to application characteristics and trade-off performance and energy efficiency. Secondly, we analyze the two-level cache and compare with the previous main cache architectures from the aspect of performance and energy efficiency with the synthetic tests and real-life IoT applications. The rest of the chapter is organized as follows: The details of our prefetching scheme in two-level cache architecture is described in section 5.2. The comparison results with synthetic tests and application are show in section 5.3, and section 5.4 concludes the chapter.

5.2 Architecture

With the target of energy efficiency design in mind, we carefully choose the prefetch strategy to avoid leading to worse performance and more power. Since the prefetch only improves performance and brings additional power inevitably, according to the equation $Energy = Power \times Time = Power \times (Cycles/Frequency)$, with a fixed frequency. The performance (1/Cycles) improvement brought by prefetch should be higher than the extra power generated by the additional area and activities.

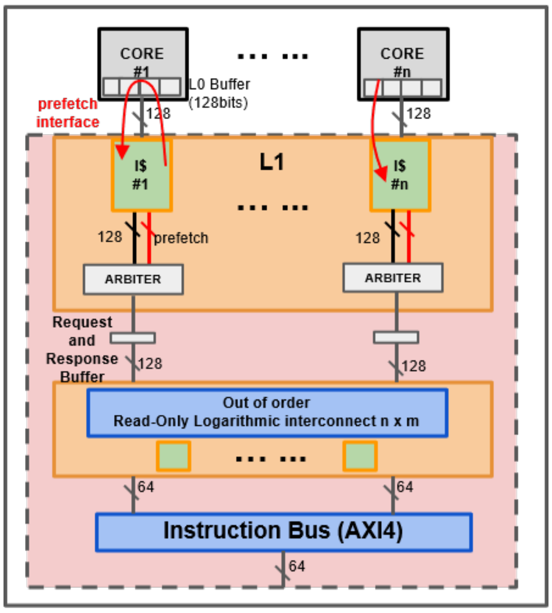


Figure 5.1: Two-level cache with L1 prefetch, the critical path shown in the red arrow.

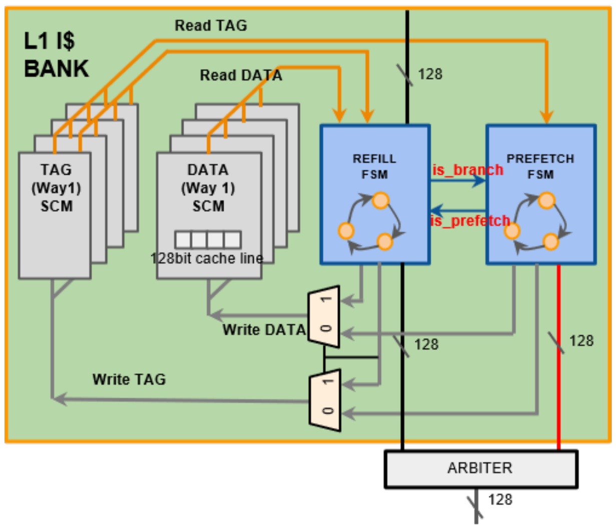


Figure 5.2: Details in the L1 cache. i) The fetch and prefetch control unit access to TAG with dual-port memories and DATA. ii) The fetch control unit has the priority to write memories.

A simple prefetch scheme is used to meet our ULP requirement, as shown in Fig. 5.1. We utilize L1 next-line (4 instructions) prefetch to largely hide the latency from core to L1 to reduce L1 capacity miss (Fig. 5.1). We choose always prefetch since the prefetch-on-miss is insufficient to hide the latency for small L1. Besides, we use dual-port TAG memories, implemented with latches for parallel cache LOOKUP between prefetch and refill operations. Fig. 5.2 shows the dual-read-port TAG for prefetch LOOKUP. Thanks to the efficient prefetch, the bandwidth (BW) from L1 to L1.5 increases little only when branches happen. Nevertheless, there is a speed mismatch between core fetch and prefetch because of the different refill latency of each memory level. As a result, we analyze two cases shown in Fig. 5.3:

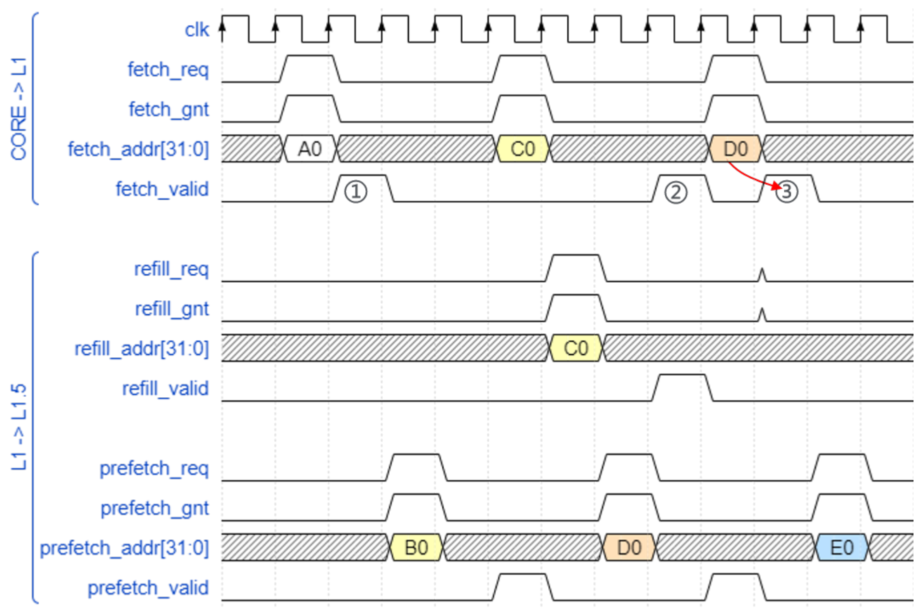


Figure 5.3: Timing diagrams of L1 to L1.5 prefetch. The upper diagram is the core fetch to L1 with one cycle latency when hit (first fetch) and three cycles latency when miss and hit in the L1.5 (second fetch). The below diagram shows the L1 refill and prefetch to the L1.5 with two cycles latency when hit. Once there is a core fetch, prefetch starts in the next cycle. L1 refill and prefetch share the interconnect's bandwidth.

- Prefetch is faster than core fetch, shown in the first *fetch_valid*. In this case, prefetch waits for the next valid core fetch to trigger again. If the next core fetch hits, we say prefetch succeeds. If not, we know that there is a branch. The prefetch control unit waits for the branch's valid fetch and restarts from the new address shown in the second *fetch_req*. The branch address is 0xC0 instead of 0xB0, and prefetch restarts from next address - 0xD0.

- Prefetch is slower than core fetch. Even though the prefetch is valid, core fetch still has a miss since prefetch is in doing. However, suppose the fetch knows a valid prefetch is in doing. In that case, the fetch can Wait for the Unfinished Prefetch (WUP) or use the prefetch data directly to save at least one cycle (red arrow) for TAG LOOKUP indicated by the signal *is_prefetch* in Fig. 5.3. The third core fetch indicates a miss from the TAG LOOKUP while finding a valid prefetch, then the refill to L1.5 is cancelled, and the cache responds directly. In conclusion, with these strategies, we can improve performance constantly.

Furthermore, we store only valid prefetch cache lines in the cache to avoid cache pollution. So before storing prefetch data to cache, we compare the recent buffered prefetch address with the current buffered fetch address, then if there is a branch, which means $((Current\ fetch\ address \neq Prefetch\ address)$ or $((Current\ address + 16) \neq Prefetch\ address))$, as shown by the signal *is_branch* in Fig. 5.2, we drop the wrong prefetch cache line. Corresponding with the previous two cases, i) Prefetch is faster than fetch, and we always store prefetch data. In this case, we may store the wrong prefetch data when a branch happens. ii) Prefetch is slower than fetch, and we are always sure that we store the useful data to cache. In real practice, we found that the prefetch is rarely slower than core fetch with the relatively large 128-bit cache line. In the end, there is no conflict to write the cache between refill and prefetch control unit with two MUXs giving priority to the refill.

5.2.1 Out-of-order interconnect

In Fig. 5.2, the refill and prefetch control unit issue a fetch to the next level through an arbiter to share the BW, and this useful prefetch increase BW a little only when branch happens. Instead of increasing 2× ports for the interconnect, which will bring more congestion and delay, we support out-of-order transfer for the refill and prefetch sharing one port. This BW sharing is shown in Fig. 5.3. For the second fetch in address 0xC0, there is a miss in the L1 cache, and the refill issues a request. After that, the prefetch in address 0xD0 starts directly without waiting for the finish of the refill. However, it is difficult to distinguish the valid responses for refill and prefetch from the shared banks since they can come in any order. Thus, we add one bit ID for each transfer in the most significant bit (MSB) of address and valid response data in the arbiter. For example, if the transfer address MSB is asserted, we need to wait for the valid response data with the MSB and vice versa. If the two responses come in the same cycle, we omit the prefetch data without disturbing the normal refill. In real practice, when there is no miss in the L1 cache (always sequential fetch without branch), the BW is almost 100% occupied by the total 2×8 requests from the refill and prefetch. With the presented prefetch scheme, we improve the performance without influencing normal fetch with minimal area overhead, including prefetch control unit, additional read port for TAG, and out-of-order interconnect

Mnemonic	Type	Hit Cycles	L1.5 Penalty	L2 Penalty	Description	Set-associate
PR	Private	1	-	15	512 Bytes I$ bank x 8 cores	4-way
SP	Shared	≥1	-	17	8 x 512 Bytes I$ banks, 1-port	4-way
MP	Shared	1	-	19	2 x 2048 Bytes I$ banks, 8-port	4-way
HIER	Private	1	≥3	19	512 Bytes L1 I$ bank x 8 cores	4-way
	Shared	≥1	-	17	2 x 2048 Bytes L1.5 I$ banks, 1-port	4-way
HIER_PRE	Private	1	≥3	19	512B I$ bank x 8 cores with prefetch	4-way
	Shared	≥1	-	17	2x 2048B I$ banks, single-port with Response buffer	4-way

Table 5.1: Instruction cache architecture configurations, including two-level cache.

to improve the performance and keep the energy efficiency.

In order to analyse the performance and power of all the caches, we run the same synthetic tests and real-life applications described in section 3.1.2 and 3.1.3 to summarize in detail the characteristic of the two-level instruction cache.

5.3 Evaluation

As shown in Table 5.1, we still use 4-way set-associate and same size for both L1 and L1.5 cache to reduce miss rate to ensure the performance for the two-level cache with prefetch (*HIER_PRE*). We use the same GF22FDX technology to implement the design and the same power model method to characterize the prefetch feature with previous work in detail.

5.3.1 Performance Results

5.3.1.1 Synthetic tests' performance

Fig. 5.4 summarizes the throughput of all synthetic tests measured on the different architectural configurations normalized to *PR*-0.375 KB. First, we can see that the performance drops of the *HIER* are significantly recovered by the prefetch feature. On average, the *HIER_PRE* has the same performance with shared caches. This means the prefetch fetches the next cache line in L1.5 in advance and in time. However, since the synthetic tests adapt to sequential prefetch, thus the *HIER_PRE* improves about 9% performance compared to the *HIER*. For real-life IoT applications with rich L2 instruction access patterns, the performance improvement of the prefetch feature may degrade.

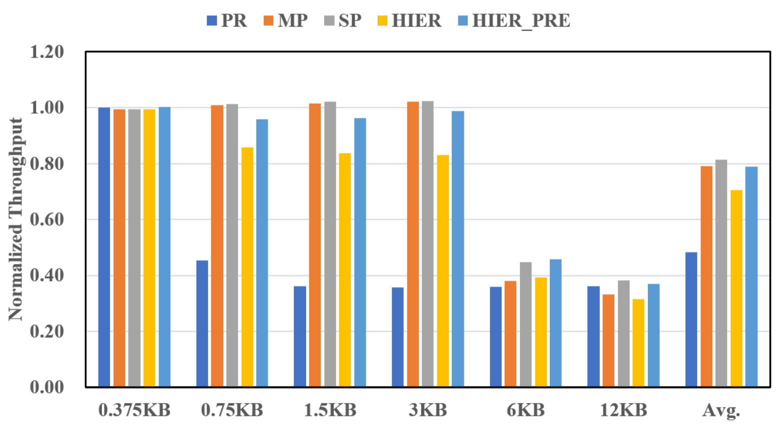
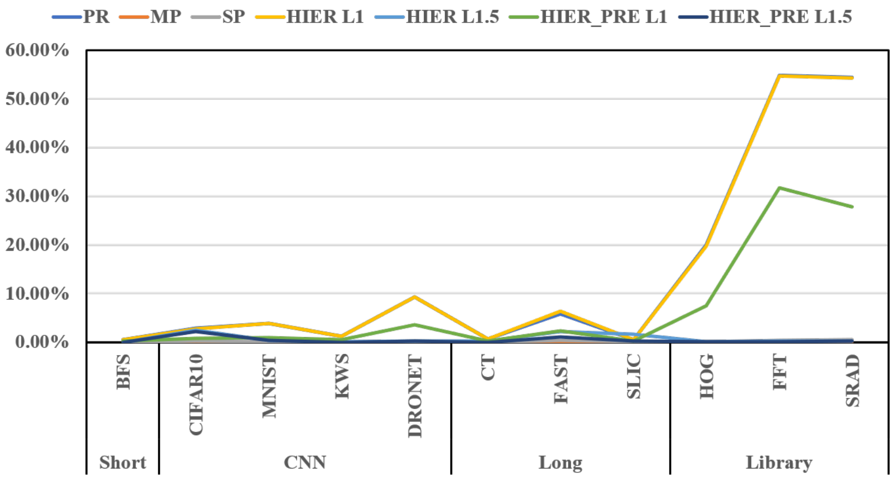


Figure 5.4: Throughput of the applications normalized to *PR* with 0.375 KB instruction size, 200MHz.

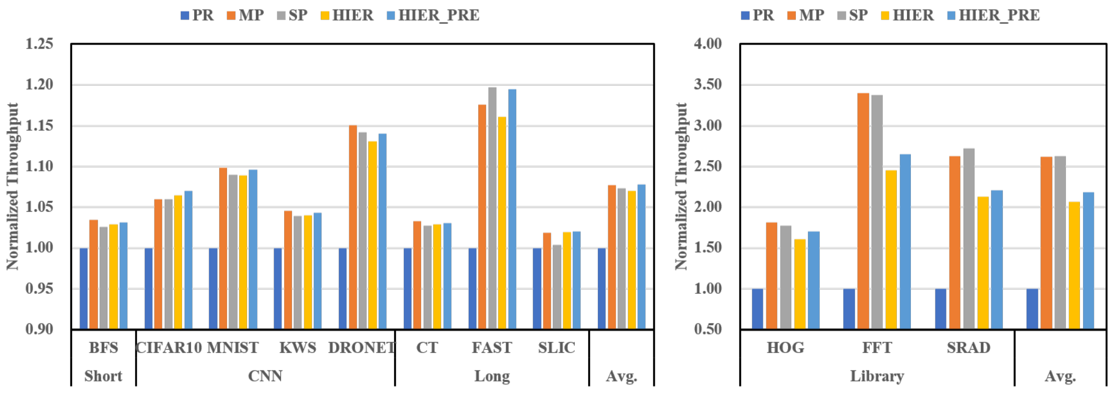
5.3.1.2 Real-life applications' performance

Fig. 5.5a shows each cache level's miss rate for each applications among all caches. It is clear that the L1 prefetch feature reduces the miss rate up by 50% compared with the two-level cache without prefetch. Thus, the *HIER_PRE* can always improve the performance. Fig. 5.5 shows the throughput of each application for all caches normalized to the *PR*. For the low miss rate application in the left, on average, the *HIER_PRE* has the best performance thanks to the prefetch feature. For the high miss rate applications, which are rarely in the IoT applications, the *HIER_PRE* improves the performance by 7% compared to the *HIER*. However, it still loses about 16% performance compared with shared caches due to the two-level structure with the limited L1 cache capacity. Besides, the sequential prefetch can not recover the performance when the frequent branches happen.

5.3.2 Physical Implementation Results



(a) Miss rate



(b) Throughput of the applications normalized to *PR*, 200MHz. the low miss rate applications in the left and the high miss rate applications in the right

Figure 5.5: Miss rate and throughput of the applications.

5.3.2.1 Area and Timing results

Fig. 5.6 illustrates the silicon area costs for all cache. We can see that the *HIER_PRE* has little area increase, about 2%, thanks to the small and efficient prefetch strategies. Since no critical path is introduced, the *HIER_PRE* keeps the same max frequency with the *HIER*. Thus, it keeps the balance between scalability and performance (Table 5.2).

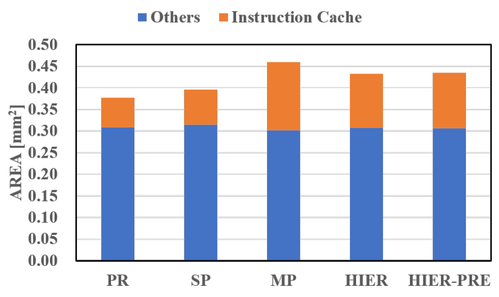


Figure 5.6: The 8-core cluster area breakdown for the different cache architectures and configurations. Instruction cache area contribution is placed on top.

Type	Cluster Maximum frequency [MHz]		Speed up compare with *PR* [%]	
	8-core	16-core	8-core	16-core
PR	378	363	0	0
SP	350	320	-7	-12
MP	357	306	-5	-16
HIER	372	354	-1	-2
HIER_PRE	372	354	-1	-2

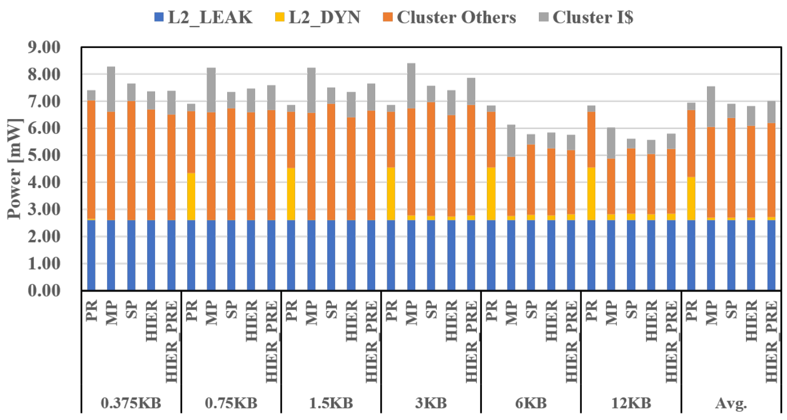
Table 5.2: Timing result

5.3.2.2 Synthetic tests' power and energy efficiency

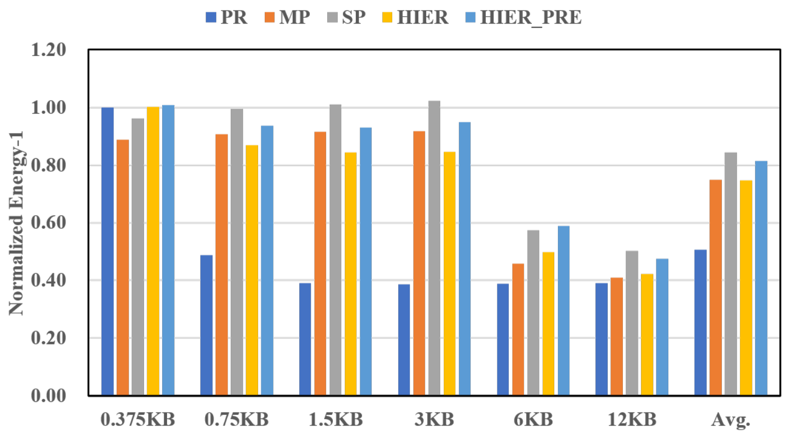
Fig. 5.7a shows the power of all synthetic tests measured on the different architectural configurations. As expected, the power of *HIER_PRE* increases a little about 3% compared with the *HIER* due to the extra area of the prefetch controller and the dual-read-port TAG memories.

As we talked about in section 5.2, if the improvement of the performance is larger than the increase of power for the prefetch feature, then we can treat it as a good prefetch scheme. In Fig. 5.7b, on average the *HIER_PRE* has a 6% energy efficiency improvement brought by the prefetch feature. Thus, we can say that our proposal is a good prefetch scheme that always improves performance while achieving higher energy efficiency. However, for the tests of 0.375 KB instruction size, the improvement of the prefetch is not apparent since the miss rate is close to 0. Besides, it still has 3% less energy efficiency on average compared with the *SP* due to its two-level structure.

By exploiting the same power model method, Fig. 5.8 updated the LUT with the power of *HIER_PRE* running with synthetic tests. We can see that the *HIER_PRE* has higher power than the *HIER* while they have the same power when the miss rate is close to 0.



(a) Power



(b) Normalized Energy efficiency

Figure 5.7: Power, energy efficiency of the synthetic tests normalized to *PR* with 0.375 KB cache size, 200MHz.

5.3.2.3 Real-life applications' power and energy efficiency

Fig. 5.9 shows the energy efficiency for each cache normalized to PR. In the left, for the low miss rate applications, the *HIER_PRE* keeps the energy efficiency with the *HIER* since the instruction fetch always hit. In this situation, the performance improvement is less than the extra power increase for prefetch. The software can choose to disable the prefetch feature to avoid extra power. However, the energy efficiency of the *HIER_PRE* still surpasses both the private and the shared caches.

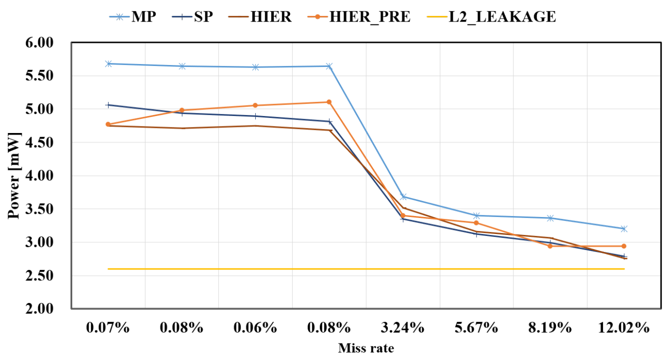


Figure 5.8: $Power_{Average} \propto (1/MissRate)$ inverse linear regression, 200MHz, updated with the *HIER_PRE*.

For the high miss rate applications, the energy efficiency improvement of the prefetch is 4% on average compared to the *HIER*. The improvement decreases compared with synthetic tests is due to the rich access patterns and frequent branches of the applications. As a result, flat shared caches always keeps the best energy efficiency with a specific frequency.

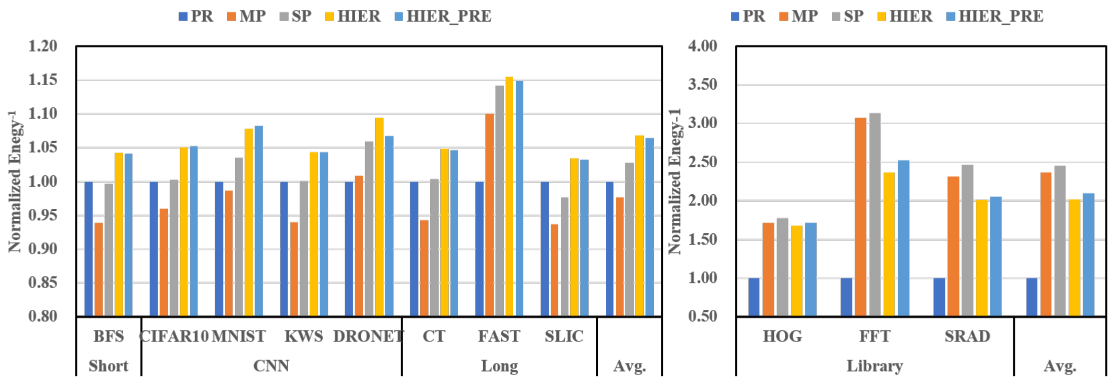


Figure 5.9: Energy efficiency of the applications normalized to *PR*, 200MHz. The low miss rate applications in the left and the high miss rate applications in the right.

5.4 Conclusion

This work proposed a sequential prefetch in L1 based on a two-level instruction cache to reduce the performance drop compared with a shared cache and keep energy efficiency. An effective prefetch scheme is adopted with the limited extra area, including an extra prefetch controller sharing the bandwidth to L1.5 with the fetch controller through an out-of-order interconnect. We explored various instruction cache architectures in an energy-efficient and cost-effective tightly coupled cluster with several signal processing and CNN applications that feature diverse instruction memory access patterns. Results show that the prefetch feature constantly improves the performance up to 7% while keeping 4% more energy efficiency in the two-level cache. Finally, the two-level instruction cache with software-enabled prefetch adapts to real-life IoT applications to achieve the highest performance and balanced energy efficiency.

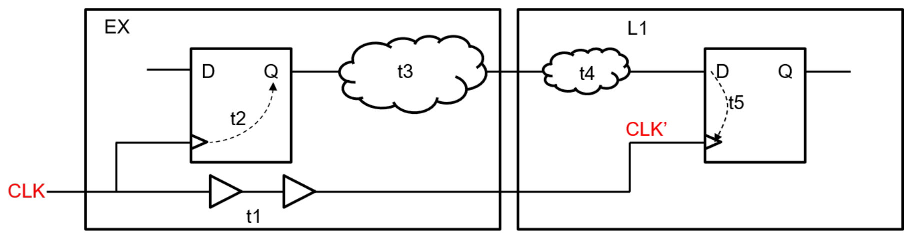
Chapter 6

Core instruction fetch timing optimization

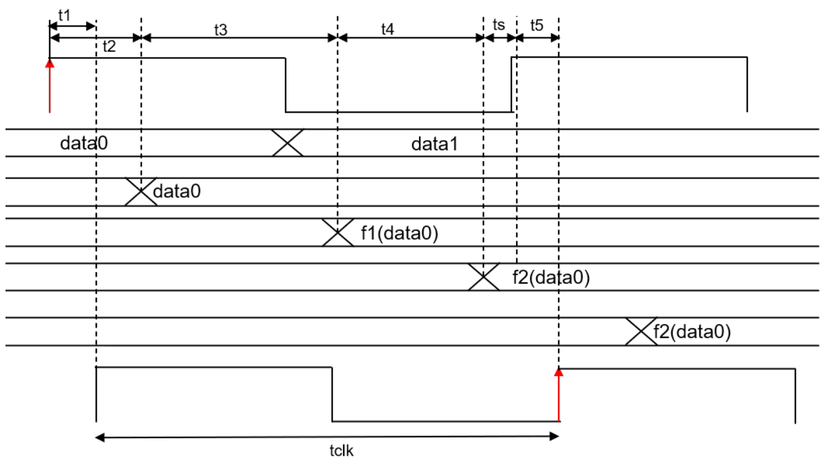
6.1 Overview

As mentioned in section 1.1.3.2 and shown in Fig. 2.15c, 2.16b and 2.17c, the caches with the legacy 128-bit instruction fetch stage (IF) have limited frequency due to some long combinational paths through the instruction fetch stage. To analyze the critical path in detail, we need to use the Static timing analysis (STA) method, a simulation method of computing the expected timing of a synchronous digital circuit without requiring a simulation of the full circuit. Fig. 6.1 shows the setup time violation from one register in the execute stage of the core to the one register in the instruction cache. Both of the two registers are clocked by the same clock CLK. In Fig. 6.1, we have the equation of $(tclk + t1) \geq (t2 + t3 + t4 + t5)$. The positive timing slack (ts) is the requirement that we must meet with certain design corners. We usually analyze the worst case for setup time to ensure the system's maximum frequency.

The first STA normally happens after the synthesis. The designers analyze the critical paths shown by the report with a given target frequency. For example, Fig. 6.2 reports one critical path with a given frequency (500MHz) in a cluster with the private cache. This critical path starts from the ID stage in the core, going to the private cache with the $fetch\_req$, then the cache responds to the core with



(a) Notion of critical path (register-to-register setup violation).

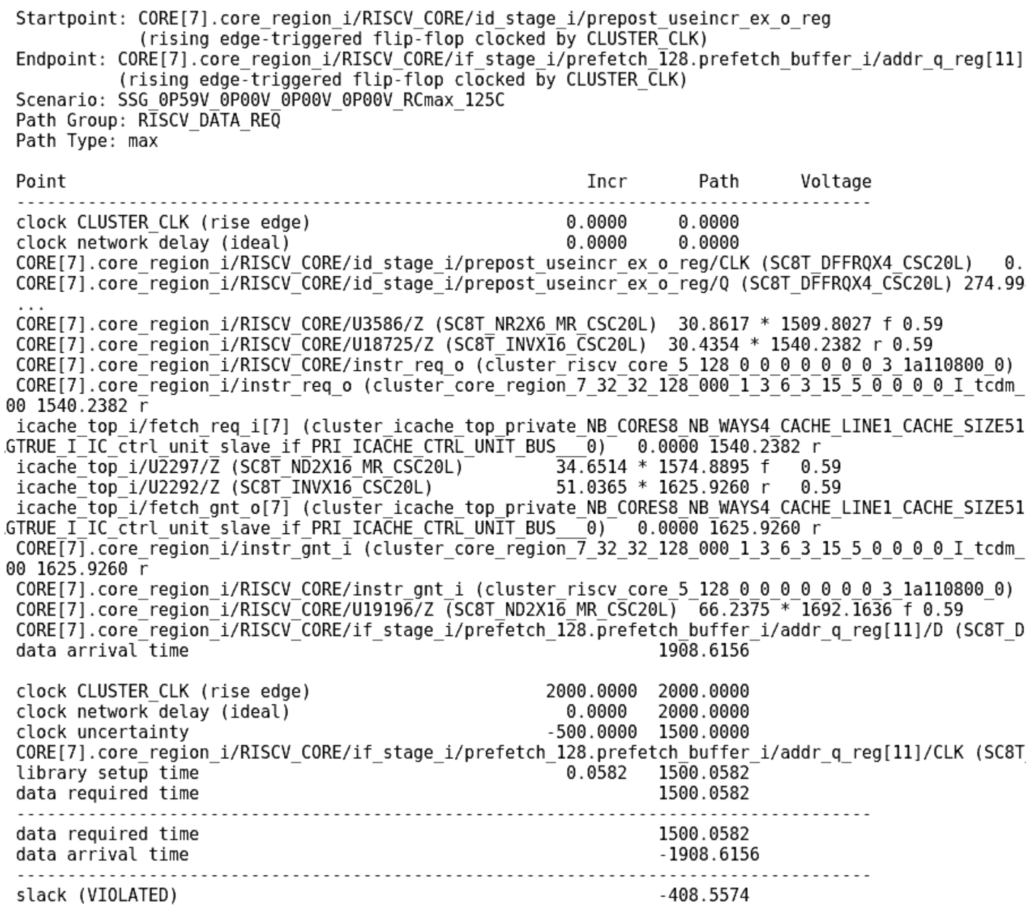


(b) Timing diagram.

Figure 6.1: Notion and timing diagram of the critical path between the cores and the instruction cache. The tclk is the clock period; the t1 is the clock skew and jitter, can be negative; the t2 is the clock to data transition time; the t3 and t4 are the time for combinational logics; the t5 is the data setup time (minimum stable time) before next clock rising edge; the ts the timing slack.

$fetch\_gnt$. Finally, the timing slack is negative about 408ps timing violation with 500ps clock uncertainty. The timing analysis is the most important task in the front-end and back-end circuit design. To close the timing issue, designers need to check through the timing reports and to fix the issues one by one. Since cache memory is always in the critical timing path, one solution is to pipeline the primary cache to reduce the memory access time. However, this will largely decrease the fetch efficiency, especially for the cache hit, which may take (1 + pipeline) times the cycles in a ULP cluster. Thus, it is necessary to distinguish the critical path and find appropriate methods to cut the critical path between the cores and the caches to speed up the system frequency.

The rest of the chapter is organized as follows: in section 6.2, we propose the methods for each type of critical path in practice. Next, the comparison results with synthetic tests and applications are shown in section 6.3. Finally, section 6.4 concludes the chapter.



```
 Startpoint: CORE[7].core_region_i/RISCV_CORE/id_stage_i/prepost_useincr_ex_o_reg
             (rising edge-triggered flip-flop clocked by CLUSTER_CLK)
 Endpoint: CORE[7].core_region_i/RISCV_CORE/if_stage_i/prefetch_128.prefetch_buffer_i/addr_q_reg[11]
           (rising edge-triggered flip-flop clocked by CLUSTER_CLK)
 Scenario: SSG_0P59V_0P00V_0P00V_0P00V_RCmax_125C
 Path Group: RISCV_DATA_REQ
 Path Type: max

 Point                                                   Incr       Path      Voltage
 ------------------------------------------------------------------------------------
 clock CLUSTER_CLK (rise edge)                         0.0000     0.0000
 clock network delay (ideal)                           0.0000     0.0000
 CORE[7].core_region_i/RISCV_CORE/id_stage_i/prepost_useincr_ex_o_reg/CLK (SC8T_DFFRQX4_CSC20L)   0.
 CORE[7].core_region_i/RISCV_CORE/id_stage_i/prepost_useincr_ex_o_reg/Q (SC8T_DFFRQX4_CSC20L) 274.99
 ...
 CORE[7].core_region_i/RISCV_CORE/U3586/Z (SC8T_NR2X6_MR_CSC20L)  30.8617 * 1509.8027 f 0.59
 CORE[7].core_region_i/RISCV_CORE/U18725/Z (SC8T_INVX16_CSC20L)  30.4354 * 1540.2382 r 0.59
 CORE[7].core_region_i/RISCV_CORE/instr_req_o (cluster_riscv_core_5_128_0_0_0_0_0_0_0_3_1a110800_0)
 CORE[7].core_region_i/instr_req_o (cluster_core_region_7_32_32_128_000_1_3_6_3_15_5_0_0_0_0_I_tcdm_
00 1540.2382 r
 icache_top_i/fetch_req_i[7] (cluster_icache_top_private_NB_CORES8_NB_WAYS4_CACHE_LINE1_CACHE_SIZE51
GTRUE_I_IC_ctrl_unit_slave_if_PRI_ICACHE_CTRL_UNIT_BUS___0)   0.0000 1540.2382 r
 icache_top_i/U2297/Z (SC8T_ND2X16_MR_CSC20L)         34.6514 * 1574.8895 f   0.59
 icache_top_i/U2292/Z (SC8T_INVX16_CSC20L)            51.0365 * 1625.9260 r   0.59
 icache_top_i/fetch_gnt_o[7] (cluster_icache_top_private_NB_CORES8_NB_WAYS4_CACHE_LINE1_CACHE_SIZE51
GTRUE_I_IC_ctrl_unit_slave_if_PRI_ICACHE_CTRL_UNIT_BUS___0)   0.0000 1625.9260 r
 CORE[7].core_region_i/instr_gnt_i (cluster_core_region_7_32_32_128_000_1_3_6_3_15_5_0_0_0_0_I_tcdm_
00 1625.9260 r
 CORE[7].core_region_i/RISCV_CORE/instr_gnt_i (cluster_riscv_core_5_128_0_0_0_0_0_0_0_3_1a110800_0)
 CORE[7].core_region_i/RISCV_CORE/U19196/Z (SC8T_ND2X16_MR_CSC20L)  66.2375 * 1692.1636 f 0.59
 CORE[7].core_region_i/RISCV_CORE/if_stage_i/prefetch_128.prefetch_buffer_i/addr_q_reg[11]/D (SC8T_D
 data arrival time                                              1908.6156

 clock CLUSTER_CLK (rise edge)                       2000.0000  2000.0000
 clock network delay (ideal)                           0.0000   2000.0000
 clock uncertainty                                  -500.0000  1500.0000
 CORE[7].core_region_i/RISCV_CORE/if_stage_i/prefetch_128.prefetch_buffer_i/addr_q_reg[11]/CLK (SC8T
 library setup time                                    0.0582   1500.0582
 data required time                                             1500.0582
 ------------------------------------------------------------------------------------
 data required time                                             1500.0582
 data arrival time                                              -1908.6156
 ------------------------------------------------------------------------------------
 slack (VIOLATED)                                               -408.5574
```

Figure 6.2: One critical path report generated by Synopsis Design Compiler after synthesis with target frequency 500 Mhz.

6.2 Architecture

As mentioned in section 1.1.3.2 and shown in Fig. 2.15c, 2.16b and 2.17c, the caches with the legacy 128-bit instruction fetch stage have limited frequency due to some long combinational paths through the instruction fetch stage (IF). There are two types of paths shown in Fig. 5.1: The path from $fetch\_valid$ or $fetch\_data$ to $fetch\_req$ and back with $fetch\_gnt$ to the core. It exists because the IF prefetch depends on the previous fetched (unaligned or non-unaligned, compressed or noncompressed) instruction to fetch the next instruction as soon as possible. Besides, the $fetch\_req$ is acknowledged by $fetch\_gnt$ back to the core. These port-to-port paths cause another critical path for the L1 cache, from the $fetch\_valid$ of the L1 cache's TAG or DATA arrays to $fetch\_req$, then back to it when TAG LOOKUP. This path worsens when logarithmic interconnect is used, such as in SP cache. 2) The path from the instruction execution stage (EX) to the IF's $fetch\_req$ until the cache has conditional branches. As shown in Fig. 6.3, we can see that instruction 0xA0 is an unconditional jump taken directly from the ID stage to instruction 0xC0. It is not on the critical path. Then, instruction 0xC0 is a conditional branch taken directly from EX stage to instruction 0xD0, instruction 0xA2, 0xC2 and 0xC4 are dropped. This critical path from EX to $fetch\_req$ improves fetching efficiency and saves one cycle. However, it limits the frequency.

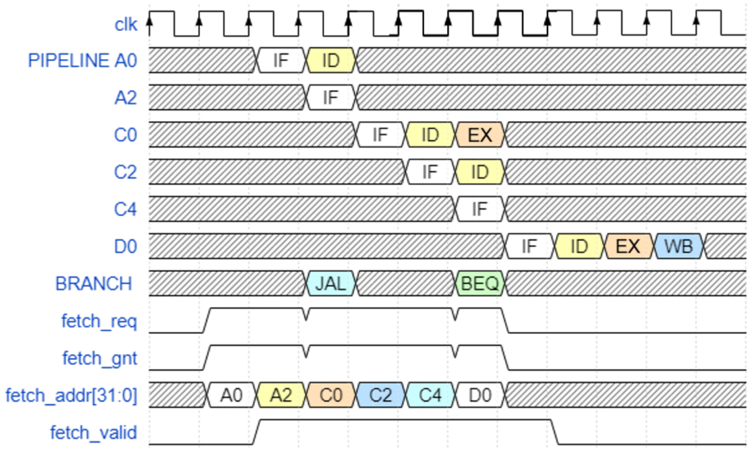
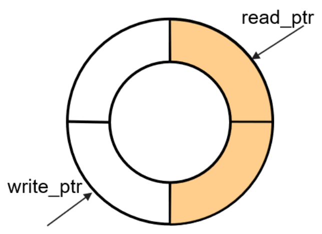


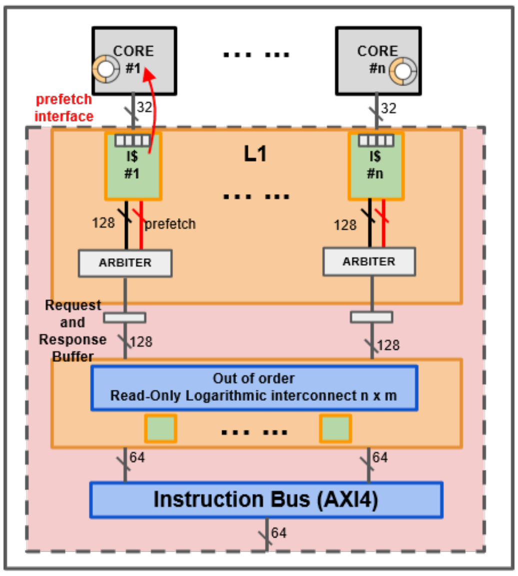
Figure 6.3: Instruction fetch and branch in RI5CY core with 4-stage pipeline.Instruction 0xA0 is a unconditional jump to 0xC0, and 0xC0 is conditional branch to 0xD0. The $fetch\_req$ depends on the $fetch\_valid$ and $fetch\_data$ because of the core prefetch with RISC-V compressed instruction set.

To cut the first type of critical path, we integrate a 4 × 32-bit ring FIFO buffer to simplify the cores' instruction fetch (Fig. 6.4a). The small ring FIFO buffer has the following features: 1) the read pointer points to the current useful instruction and the write pointer points to the next available writing space; 2) the FIFO is full when the number of useful instruction is equal or greater than the FIFO depth minus one; 3) the core can send non-blocking to fetch requests when the FIFO is not full; 4) the ring FIFO helps act as a primary cache when the short branches hit in the ring FIFO. To cut the second type of critical path from EX for the conditional branch, we must insert a pipeline or implement a branch predictor in the IF/ID stage. However, the branch predictors with branch addresses registering for diverse branches take huge extra area and power. Besides, branch predictors' index searching is also in the critical path to the $fetch\_req$. In the end, we choose to delay the conditional branch one cycle to increase the frequency.

Fig. 6.4 shows the two-level cache after IF optimization, and we can see that a small 4×32-bit ring FIFO buffer is used to issue the $fetch\_req$ without dependence on other signals. Besides, a 128-bit L0 buffer is still used to avoid heavy request traffic to the L1 cache control unit. As a result, it brings more power than the legacy 128-bit IF with the extra ring FIFO buffer. In the end, the final remaining path (red arrow) starts from the L1 caches' DATA array to cores' inner instruction decompression logic, which is small.



(a) A 4×32 ring FIFO buffer.



(b) Throughput, 200MHz

Figure 6.4: Two-level cache with L1 prefetch after IF optimization with 4×32-bit ring FIFO buffer and additional conditional branch pipeline, the remain critical path shown in the red arrow.

In order to analyse the performance and power of all the caches, we run the same synthetic tests and real-life applications described in section 3.1.2 and 3.1.3 to summarize in detail the characteristic of the two-level instruction cache with the optimized IF.

Mnemonic	Type	Hit Cycles	L1.5 Penalty	L2 Penalty	Description	Set-associate
PR	Private	1	-	15	512 Bytes I$ bank x 8 cores	4-way
SP	Shared	≥1	-	17	8 x 512 Bytes I$ banks, 1-port	4-way
MP	Shared	1	-	19	2 x 2048 Bytes I$ banks, 8-port	4-way
HIER	Private	1	≥3	19	512 Bytes L1 I$ bank x 8 cores	4-way
	Shared	≥1	-	17	2 x 2048B I$ banks, single-port with Response buffer	4-way
HIER_PRE	Private	1	≥3	19	512B I$ bank x 8 cores with prefetch	4-way
	Shared	≥1	-	17	2x 2048B I$ banks, single-port with Response buffer	4-way
HIER_OPT	Private	1	≥3	19	512B I$ bank x 8 cores	4-way
	Shared	≥1	-	17	2x 2048B I$ banks, single-port with Response buffer	4-way
HIER_PRE_OPT	Private	1	≥3	19	512B I$ bank x 8 cores with prefetch	4-way
	Shared	≥1	-	17	2x 2048B I$ banks, single-port with Response buffer	4-way

Table 6.1: Instruction cache architecture configurations, including two-level cache.

6.3 Evaluation

As shown in Table 6.1, we use a 4-way set-associate for both L1 and L1.5 cache to reduce miss rate and ensure performance. We use the same GF22FDX technology to implement the design and the same power model method to compare our proposal with previous work in detail.

6.3.1 Performance Results

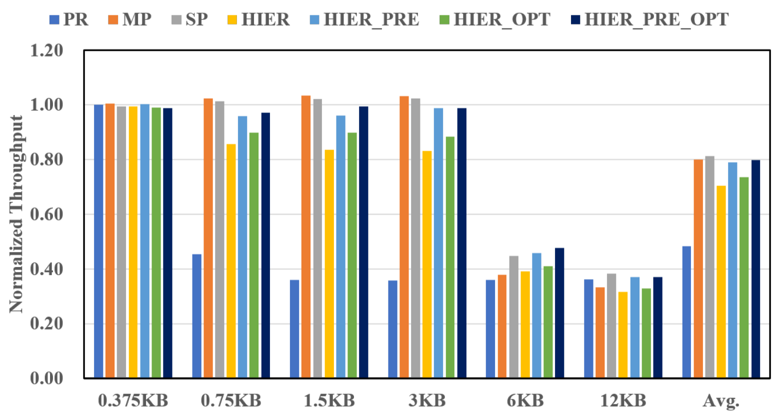
6.3.1.1 Synthetic tests' performance

Fig. 6.5a shows the normalized throughput of the synthetic tests running on the different architecture configurations, assuming that all the configurations are running at the same operating frequency, providing an insight into their functional performance. When the loop body is smaller than the size of L1 of the *HIER* caches (0.5 KB), the throughput of all the configurations is the same since they always hit in L1. When increasing the size of the loop body to 0.75 KB, the performance of the PR cache drops significantly (∼55%) since each cache miss is refilled from L2 featuring 15 cycles of latency. On the other hand, the *HIER* configurations feature only a slight drop in performance (∼15%) thanks to the low latency of the refills from the 4 KB L1.5, which is almost completely recovered activating the prefetcher. When the loop body is larger than 4 KB (size of shared configuration and L1.5 of the *HIER* configuration), the performance of these configurations also starts dropping due to capacity misses. Besides, We can see that the *HIER_OPT* has slightly better performance on average than the *HIER* is because the 4×32 ring FIFO buffer features the next instruction sequential prefetching when FIFO is not full.

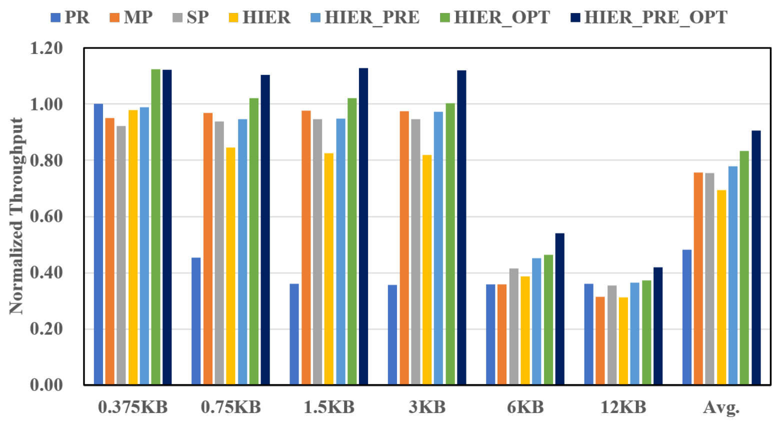
Fig. 6.5b shows the normalized throughput when each configuration runs at the maximum operating frequency. It is possible to note that *HIER_PRE_OPT* improves the performance by ∼16% for all the synthetic tests with respect to the shared caches thanks to the similar functional performance and much higher maximum operating frequency. In particular, most of the gain is thanks to the optimized instruction fetch stage of the RI5CY core described in section 6.2.

6.3.1.2 Real-life applications' performance

Fig. 6.6a shows the functional performance of the proposed cache architectures (i.e., when running at the same operating frequency) normalized to *PR*. In general, since shared caches and two-level cache can remove capacity miss with relatively larger cache size, they always have better performance than PR. Besides, *HIER* has two more cycles refill when L1 miss and L1.5 hit than *SP*, so it loses 2% performance compared with shared caches. When we compare the performance of low L1 miss rate applications, *HIER_PRE* can achieve the same performance with shared caches and always improve performance compared with *HIER*. *HIER_PRE_OPT* functional performance is on average smaller by 5% with respect to *HIER* and shared caches, due to the stall caused by the pipeline stage added to unconditional branches to improve the operating frequency of the design. For always hit applications such as BFS, CT, and SLIC, the functional performance drop is smaller, 3.5% on average compared to PR. Finally, for high L1 miss rate applications, even though *HIER_PRE* and *HIER_PRE_OPT* reduce the L1 capacity miss and improve the performance by 7% on average compared with *HIER* and *HIER_OPT*, their functional performance is about 17% smaller compared to shared caches mainly due to the high L1 miss



(a) Throughput, 200MHz



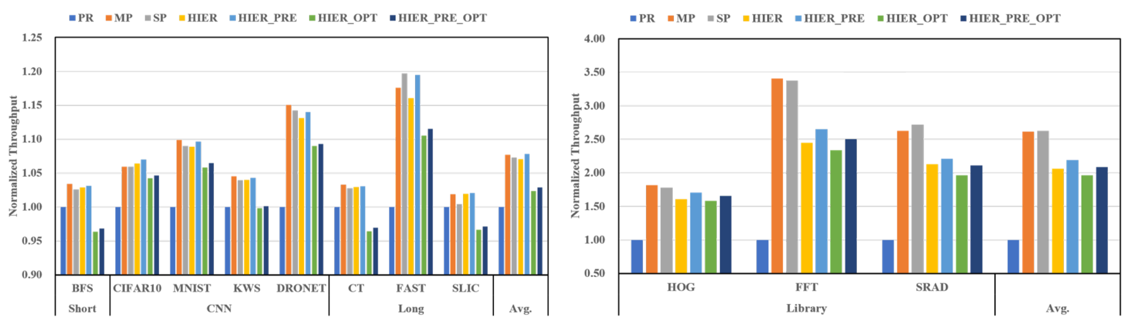
(b) Maximum throughput

Figure 6.5: Throughput of the applications normalized to *PR* with 0.375 KB instruction size.

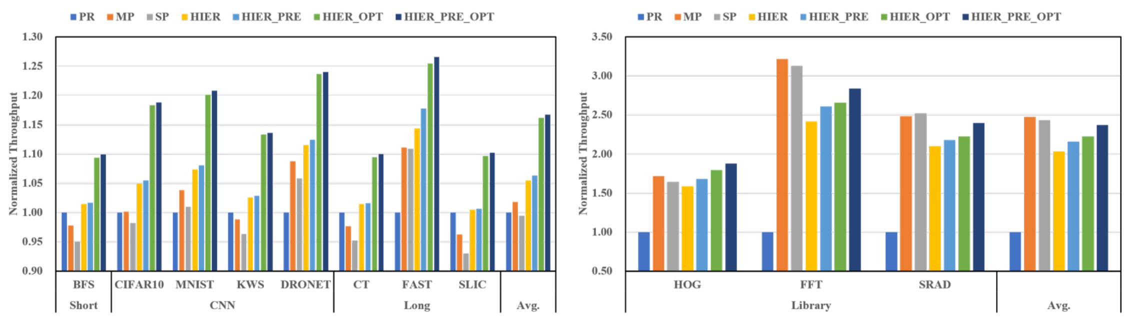
rate and additional latency required to refill from L2.

Fig. 6.6b shows the performance results of the different cache architectures when operating at the maximum operating frequency, highlighting the better scalability of the hierarchical solutions, especially the one with optimized instruction fetch unit. Results are normalized to the performance of *PR*. The *HIER_PRE_OPT* improves the performance by 17% compared with private cache and shared cache for low L1 miss rate applications. For high L1 miss rate applications (which are again not common in the IoT domain), *HIER_PRE_OPT* delivers 2.4x better performance than private caches and only about 5% smaller performance than the shared caches.

6.3.2 Physical Implementation Results



(a) Throughput, 200MHz



(b) Maximum throughput

Figure 6.6: Throughput of the applications normalized to *PR*, 200MHz. the low miss rate applications in the left and the high miss rate applications in the right.

6.3.2.1 Area and Timing results

Fig. 6.7 illustrates the silicon area costs for all cache. We can see that the *HIER_PRE_OPT* has almost no extra area increase because only a 4 $\times$ 32-bit ring FIFO is introduced in each of the cluster core.

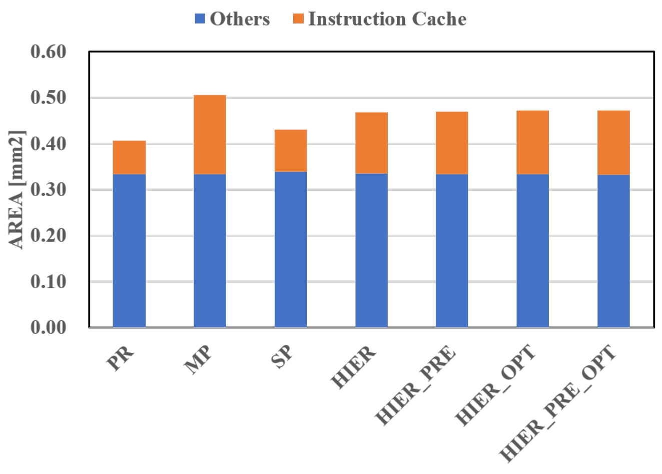


Figure 6.7: The 8-core cluster area breakdown for the different cache architectures and configurations. Instruction cache area contribution is placed on top.

Table 6.2 shows the results of the static timing analysis for all the cache architectures, implemented in clusters of 8 cores or 16 cores to highlight the better scalability of the proposed cache. For a system with 8 cores, the caches with simple L1 private cache have the best timing, keeping about 6% timing improvement compared to shared caches. After optimization of the IF, the *HIER_OPT* can have 14% maximum frequency improvement compared to the flat private cache. When we increase the system core number to 16, we observe that *MP* and *SP* have about 19% and 15% maximum frequency drop respectively compared to *HIER_OPT*. For the *MP*, 16-port memory banks cause serious wire congestion, leading to worse timing results. It is the same issue for *SP*, with more channels and more levels logarithmic interconnection, the critical paths become worse with interactions of *request* and *response* channels. Thus the two-level cache with n optimized core fetch interface has better timing and scalability by eliminating the long paths in the *request* and *response* channels.

Type	Cluster Maximum frequency [MHz]		Speed up compare with *PR* [%]	
	8-core	16-core	8-core	16-core
PR	378	363	0	0
SP	350	320	-7	-12
MP	357	306	-5	-16
HIER	372	354	-1	-2
HIER_OPT	429	399	+14	+10
HIER_PRE_OPT	429	399	+14	+10

Table 6.2: Timing result

6.3.2.2 Synthetic tests' power and energy efficiency

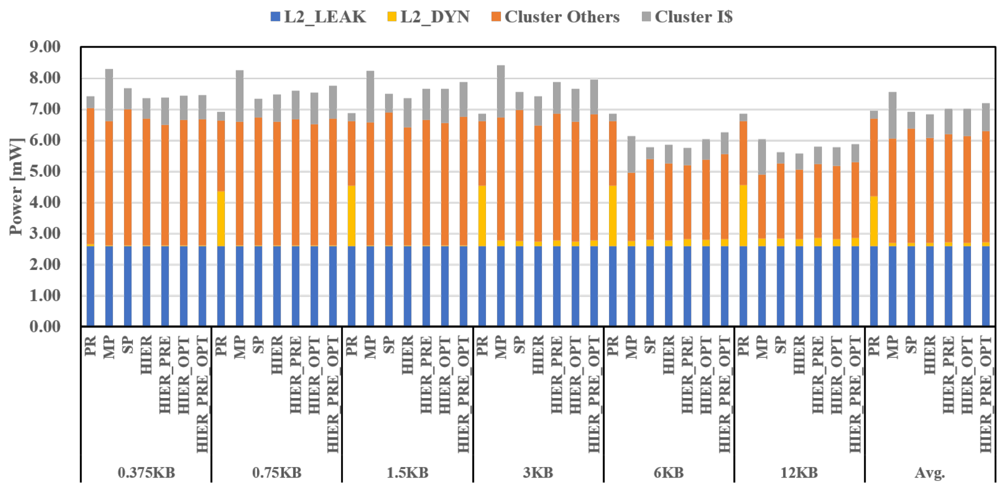
Fig. 6.8a shows the power of all synthetic tests measured on different architecture configurations. The *HIER_OPT* and the *HIER_PRE_OPT* have one more ring buffer FIFO described in section 6.2 than the *HIER*, so they bring 2% and 3% more power than the *HIER* on average, respectively.

Fig. 6.8b shows the energy efficiency of all synthetic tests measured on the different architecture configurations. In general, the *SP* is the cache providing the best trade-off between performance and power, delivering better energy efficiency on these synthetic benchmarks. Besides, two-level caches are penalized due to the higher power consumption caused by their two-level nature. The versions without prefetcher are further penalized by the worse functional performance, which is recovered by enabling the prefetcher to bring 10% to 20% better performance with only ~3% increase of power.

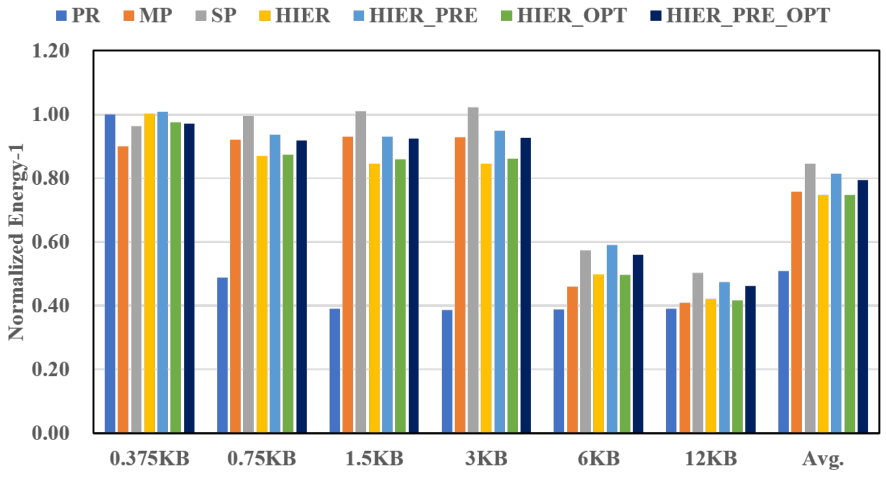
By exploiting the same power model method, Fig. 6.9 updated the LUT with the power of the *HIER_OPT* and the *HIER_PRE_OPT* running with synthetic tests. We can see that the two-level caches with the optimized IF have more power than the previous two-level caches with an extra 4 $\times$ 32-bit ring FIFO buffer. We use it to do power evaluation for the proposed caches.

6.3.2.3 Real-life applications' power and energy efficiency

Fig. 6.10 shows the results of energy efficiency, which are normalized to *PR*. As well as before, for low L1 miss rate applications, the improvement of efficiency of *HIER* over *PR* is limited since the L1 cache suffers less capacity miss. Moreover, the additional power brought by the prefetcher in *HIER_PRE* is more or less equivalent to the improvement of performance, which keeps the same energy efficiency compared with *HIER*. The same applies when we compare *HIER_PRE_OPT* with *HIER_OPT*. *HIER_OPT* and *HIER_PRE_OPT* consume more power than *HIER* because of the additional 4x32-bit ring FIFO buffer to cut the critical path. Since most of the real-life IoT applications have a low L1 miss rate, *HIER*'s small L1 cache takes advantage of lower power and relatively high energy efficiency without losing much performance compared with shared caches. As a result, *HIER* and *HIER_PRE* feature, on average, 7% better energy efficiency than all other caches. Besides, the decision of using the prefetch feature in a two-level cache should be considered by software for low L1 miss applications. For high L1 miss rate applications, *HIER_PRE* has the same energy efficiency as *HIER*, which means the additional power gain is equal to the performance gain. Finally, *HIER_PRE* brings a 7% improvement in performance while keeping the same energy efficiency compared with *HIER*. Nevertheless, shared caches remove capacity miss with large L1, so they have



(a) Power



(b) Normalized Energy efficiency

Figure 6.8: Power, energy efficiency of the synthetic tests normalized to *PR* with 0.375 KB cache size, 200MHz.

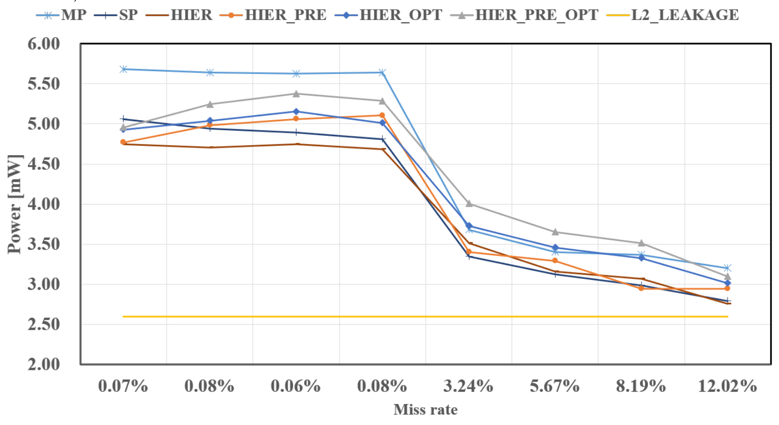


Figure 6.9: $Power_{Average} \propto (1/MissRate)$ inverse linear regression, 200MHz, updated with the *HIER_OPT* and the *HIER_PRE_OPT*.

about 20% gain in energy efficiency compared with two-level caches. However, since *HIER_OPT* isolates the critical path from the cores to instruction caches, it brings better scalability for multi-core systems regarding the number of cores per cluster and maximum operating frequency.

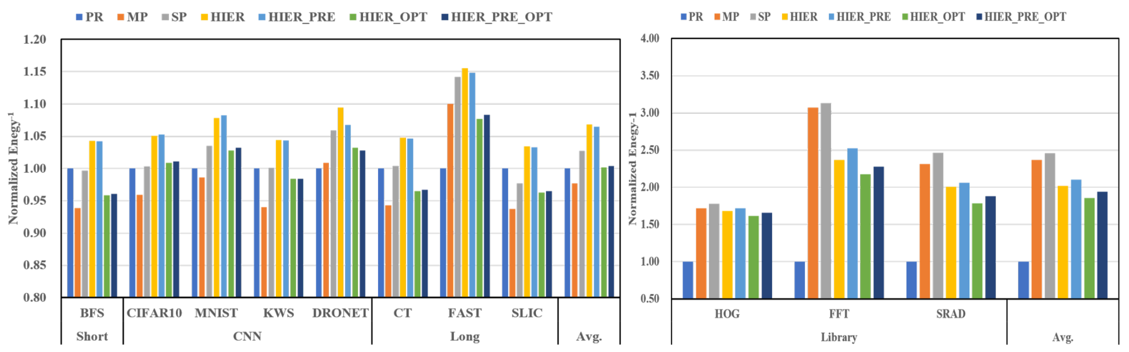


Figure 6.10: Energy efficiency of the applications normalized to *PR*, 200MHz. The low miss rate applications in the left and the high miss rate applications in the right.

6.3.3 Discussion

Type	Area	Maximum Frequency		Low L1 miss rate			High L1 miss rate		
		8-core	16-core	MxP	P	1/E	MxP	P	1/E
PR	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
MP	1.25	0.95	0.84	1.02	1.10	0.98	2.47	1.10	2.38
SP	1.06	0.93	0.88	0.99	1.04	1.03	2.43	1.06	2.48
HIER	1.17	0.99	0.98	1.06	1.00	1.07	2.03	1.01	1.99
HIER_PRE	1.17	0.99	0.98	1.06	1.01	1.07	2.16	1.04	2.11
HIER_OPT	1.18	1.14	1.10	1.16	1.01	1.00	2.22	1.05	1.87
HIER_PRE_OPT	1.18	1.14	1.10	1.17	1.02	1.00	2.37	1.07	1.95

Table 6.3: Summary of Maximum frequency and Maximum Performance (MxP), Power (P), Area and Energy Efficiency (1/E) of the proposed caches.

To put the experimental results in perspective, we collect them in Table 6.3. We separate the results into two groups, low and high L1 miss rate applications. For high L1 miss rate applications, not common in the IoT domain, both shared and two-level caches feature 2× better performance than private cache thanks to the large cache capacity. Single-port shared cache features the best energy efficiency, and multi-port shared cache has the maximum performance. Still, the two-level cache performance and energy efficiency are not so far from that of shared caches, and the prefetcher can mitigate the performance drop reducing it to 5% at the cost of some more power.

For the low L1 miss rate applications, we note that the two-level cache with optimized instruction fetch subsystem delivers significant maximum performance, up to 17% larger with respect to private and shared caches. The baseline two-level cache has the best energy efficiency, 7% and 4% better than private cache and single-port shared cache, respectively. It is interesting to note the trade-off between the optimized fetch unit and the legacy one, the one performing better efficiency thanks to the larger 128-bit interface requiring less control overhead for refills, and the other one significantly relaxing the critical path through the core by means of a simpler straight forward implementation leading to higher operating frequency for the cluster, particularly when scaling up the number of computing cores. Finally, when we review the target architecture in Table 1.1, our proposed two-level cache with optimized timing realizes the goal of large cache capacity, high performance, no congestion, small area, and good timing.

6.4 Conclusion

This work proposed a timing optimization of the core instruction fetch stage in a two-level instruction cache to maximize performance and scalability. We explored various instruction cache architectures in an energy-efficient and cost-effective tightly coupled cluster with several signal processing and CNN applications that feature diverse instruction memory access patterns. Results show that the proposed two-level cache improves the maximum performance up to 17% compared with private and shared caches. Finally, the timing improvement enables the two-level cache to adapt to real-life IoT applications to achieve the highest performance and balanced energy efficiency.

Chapter 7

Conclusions

This thesis presents a hierarchy instruction cache based on Stand Cell Memory with prefetching in L1 cache, combining both L1 private iCache with L1.5 shared iCache with an ultra-low latency out-of-order logarithmic interconnect to increase the scalability while balancing the performance and energy in a general-purpose ultra-low-power multi-core cluster. This architecture meets the requirement of real-life applications, including signal-processing and convolutional neural network applications to enable the multi-core cluster to adapt to high performance and low power usage cases. First, three iCache based on SCM are optimized and adapted to the PULP platform and explored in detail to select the best iCache configuration in terms of cache size and associativity to achieve high performance and low power. To solve the private iCahce's small capacity issue, two shared caches with N *times* cache capacity are proposed to increase the performance and energy efficiency. At the same time, shared caches suffer from timing issues with limited operating frequency and low scalability when the number of cores increases. Next, the two-level iCache is present to combine both of the benefits of private and shared iCache to balance the performance and energy efficiency. Physical implementations are done for each of the clusters featuring different iCache for concrete comparison; area results are obtained with the P & R netlists. Clusters' power characteristics are extracted with the help of standalone synthetic tests and are applied to real-life applications. Based on that, energy efficiency is calculated with the cycle-accurate simulation results. The result shows that the two-level iCache improves the energy efficiency by 7% and 4% respectively compared to private and shared iCache for the low L1 miss rate applications. However, the L1 cache's small capacity for some library-based applications with frequent branches and long jumps can influence the performance achieved by such two-level architecture. Thus, we add sequential next cache line prefetching to improve the performance with little area overhead. Results show that the prefetch feature constantly improves the performance up to 7% while keeping the same energy efficiency in the two-level cache. After timing optimization of the core instruction fetch stage, the two-level cache improves the maximum performance up to 17% compared with private and shared iCaches. Finally, the two-level instruction cache with software-enabled prefetch and up to 20% timing improvement adapts to real-life IoT applications to achieve the highest performance and balanced energy efficiency.

Publications

2020

C. Jie, I. Loi, L. Benini and D. Rossi, ”Energy-Efficient Two-level Instruction Cache Design for an Ultra-Low-Power Multi-core Cluster,” 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2020, pp. 1734-1739, doi: 10.23919/DATE48585.2020.9116212.

2021

D. Rossi et al., ”4.4 A 1.3TOPS/W @ 32GOPS Fully Integrated 10-Core SoC for IoT End-Nodes with 1.7µW Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode,” 2021 IEEE International Solid-State Circuits Conference (ISSCC), 2021, pp. 60-62, doi: 10.1109/ISSCC42613.2021.9365939.

Bibliography

[1] Agarwal A., Li H., Roy K., 2002, in Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324). *DRG-cache: a data retention gated-ground cache for low power*. pp 473–478, doi:10.1109/DAC.2002.1012671

[2] Albonesi D., 1999, in MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture. *Selective cache ways: on-demand cache resource allocation*. pp 248–259, doi:10.1109/MICRO.1999.809463

[3] Andreas Meinerzhagen P., Sherazi S. M. Y., Burg A., Rodrigues J., 2011, *Benchmarking of Standard-Cell Based Memories in the Sub-V_{rmT} Domain in 65-nm CMOS Technology*, Emerging and Selected Topics in Circuits and Systems, IEEE Journal on, 1, 173

[4] Ansari A., Lotfi-Kamran P., Sarbazi-Azad H., 2020, in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). *Divide and Conquer Frontend Bottleneck*. pp 65–78, doi:10.1109/ISCA45697.2020.00017

[5] Ansari A., Golshan F., Lotfi-Kamran P., Sarbazi-Azad H., 2021, *MANA: Microarchitecting an Instruction Prefetcher*, doi:10.48550/ARXIV.2102.01764, `https://arxiv.org/abs/2102.01764`

[6] Baer J.-L., Wang W.-H., 1988, *On the Inclusion Properties for Multi-Level Cache Hierarchies*, SIGARCH Comput. Archit. News, 16, 73–80

[7] Banerjee U., Juvekar C., Wright A., Arvind Chandrakasan A. P., 2018, in 2018 IEEE International Solid - State Circuits Conference - (ISSCC). *An energy-efficient reconfigurable DTLS cryptographic engine for End-to-End security in iot applications*. pp 42–44, doi:10.1109/ISSCC.2018.8310174

[8] Benini L., Macii A., Macii E., Poncino M., 2000, *Increasing Energy Efficiency of Embedded Systems by Application-Specific Memory Hierarchy Generation*, IEEE Des. Test, 17, 74–85

[9] Benini L., Flamand E., Fuin D., Melpignano D., 2012a, in 2012 Design, Automation Test in Europe Conference Exhibition (DATE). *P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator*. pp 983–987, doi:10.1109/DATE.2012.6176639

[10] Benini L., Flamand E., Fuin D., Melpignano D., 2012b, in 2012 Design, Automation Test in Europe Conference Exhibition (DATE). *P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator*. pp 983–987, doi:10.1109/DATE.2012.6176639

[11] Berg S. G., 2002. *Cache Prefetching*

[12] Burrello A., Garofalo A., Bruschi N., Tagliavini G., Rossi D., Conti F., 2021, *DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs*, IEEE Transactions on Computers, pp 1–1

[13] Calhoun B., Chandrakasan A., 2006, in 2006 IEEE International Solid State Circuits Conference - Digest of Technical Papers. *A 256kb Sub-threshold SRAM in 65nm CMOS*. pp 2592–2601, doi:10.1109/ISSCC.2006.1696325

[14] Canziani A., Paszke A., Culurciello E., 2016, *An Analysis of Deep Neural Network Models for Practical Applications*, CoRR, abs/1605.07678

[15] Canziani A., Paszke A., Culurciello E., 2017, *An Analysis of Deep Neural Network Models for Practical Applications* (`arXiv:1605.07678`)

[16] Chang L., et al., 2005, in Digest of Technical Papers. 2005 Symposium on VLSI Technology, 2005.. *Stable SRAM cell design for the 32 nm node and beyond*. pp 128–129, doi:10.1109/.2005.1469239

[17] Chang I. J., Kim J.-J., Park S. P., Roy K., 2008, in 2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers. *A 32kb 10T Sub-*

threshold SRAM Array with Bit-Interleaving and Differential Read Scheme in 90nm CMOS. pp 388–622, doi:10.1109/ISSCC.2008.4523220

[18] Chung S. W., Skadron K., 2008, *On-Demand Solution to Minimize I-Cache Leakage Energy with Maintaining Performance*, IEEE Transactions on Computers, 57, 7

[19] Conti F., Marongiu A., Benini L., 2013, in Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. *Synthesis-friendly Techniques for Tightly-coupled Integration of Hardware Accelerators into Shared-memory Multi-core Clusters*, CODES+ISSS '13. IEEE Press, Piscataway, NJ, USA, pp 5:1–5:10, http://dl.acm.org/citation.cfm?id=2555692.2555697

[20] Conti F., Marongiu A., Pilkington C., Benini L., 2016, *He-P2012: Performance and Energy Exploration of Architecturally Heterogeneous Many-Cores*, J. Signal Process. Syst., 85, 325–340

[21] Conti F., et al., 2017, *An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics*, IEEE Transactions on Circuits and Systems I: Regular Papers, 64, 2481

[22] Conti F., Schiavone P. D., Benini L., 2018, *XNOR Neural Engine: A Hardware Accelerator IP for 21.6-fJ/op Binary Neural Network Inference*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

[23] Dang X., Wang X., Tong D., Xie Z., Li L., Wang K., 2013, in 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC). *An adaptive filtering mechanism for energy efficient data prefetching*. pp 332–337, doi:10.1109/ASPDAC.2013.6509617

[24] Davide Schiavone P., Conti F., Rossi D., Gautschi M., Pullini A., Flamand E., Benini L., 2017, in 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS). *Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications*. pp 1–8, doi:10.1109/PATMOS.2017.8106976

[25] Edmondson J. H., et al., 1995, *Internal Organization of the Alpha 21164, a 300-MHz 64-Bit Quad-Issue CMOS RISC Microprocessor*, Digital Tech. J., 7, 119–135

[26] Ferdman M., Wenisch T. F., Ailamaki A., Falsafi B., Moshovos A., 2008, in 2008 41st IEEE/ACM International Symposium on Microarchitecture. *Temporal instruction fetch streaming*. pp 1–10, doi:10.1109/MICRO.2008.4771774

[27] Ferdman M., Kaynak C., Falsafi B., 2011, in 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). *Proactive instruction fetch*. pp 152–162

[28] Flamand E., Rossi D., Conti F., Loi I., Pullini A., Rotenberg F., Benini L., 2018a, in 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP). *GAP-8: A RISC-V SoC for AI at the Edge of the IoT*. pp 1–4

[29] Flamand E., Rossi D., Conti F., Loi I., Pullini A., Rotenberg F., Benini L., 2018b, in 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP). *GAP-8: A RISC-V SoC for AI at the Edge of the IoT*. pp 1–4, doi:10.1109/ASAP.2018.8445101

[30] Flautner K., Kim N. S., Martin S., Blaauw D., Mudge T., 2002, in Proceedings 29th Annual International Symposium on Computer Architecture. *Drowsy caches: simple techniques for reducing leakage power*. pp 148–157, doi:10.1109/ISCA.2002.1003572

[31] Fu J. W. C., Patel J. H., Janssens B. L., 1992, in Proceedings of the 25th Annual International Symposium on Microarchitecture. *Stride Directed Prefetching in Scalar Processors*, MICRO 25. IEEE Computer Society Press, Washington, DC, USA, p. 102–110

[32] Garello K., et al., 2018, in 2018 IEEE Symposium on VLSI Circuits. *SOT-MRAM 300MM Integration for Low Power and Ultrafast Embedded Memories*. pp 81–82, doi:10.1109/VLSIC.2018.8502269

[33] Gautschi M., Traber A., Pullini A., Benini L., Scandale M., Di Federico A., Beretta M., Agosta G., 2015, in 2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC). *Tailoring instruction-set extensions for an ultra-low power tightly-coupled cluster of OpenRISC cores*. pp 25–30, doi:10.1109/VLSI-SoC.2015.7314386

[34] Gautschi M., et al., 2017, *Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 25, 2700

[35] Ge Z., Mitra T., Wong W.-F., 2009, in 2009 46th ACM/IEEE Design Automation Conference. *A DVS-based pipelined reconfigurable instruction memory*. pp 897–902

[36] Geer D., 2005, *Chip makers turn to multicore processors*, Computer, 38, 11

[37] GreenWaves Technologies 2018, *GAP8 Auto-tiler Manual*, `https://greenwaves-technologies.com`

[38] Hajimiri H., Rahmani K., Mishra P., 2012, *Compression-aware dynamic cache reconfiguration for embedded systems*, Sustain. Comput. Informatics Syst., 2, 71

[39] Harris D. M., Ho R., Wei G.-Y., Horowitz M., 1998. *The Fanout-of-4 Inverter Delay Metric*

[40] Hennessy J. L., Patterson D. A., 2002, *Computer Architecture: A Quantitative Approach*, 3rd edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

[41] IPC-1 2020, *The 1st instruction prefetching championship*, `https://research.ece.ncsu.edu/ipc/`

[42] ITRS 2011, *International Technology Roadmap for Semiconductors.*, `http://www.itrs.net/Links/2011ITRS/2011Chapters/2011ExecSum.pdf`

[43] Iacobovici S., Spracklen L., Kadambi S., Chou Y., Abraham S. G., 2004, in Proceedings of the 18th Annual International Conference on Supercomputing. *Effective Stream-Based and Execution-Based Data Prefetching*, ICS

’04. Association for Computing Machinery, New York, NY, USA, p. 1–11, doi:10.1145/1006209.1006211, `https://doi.org/10.1145/1006209.1006211`

[44] Ickes N., Sinangil Y., Pappalardo F., Guidetti E., Chandrakasan A. P., 2011, in 2011 Proceedings of the ESSCIRC (ESSCIRC). *A 10 pJ/cycle ultra-low-voltage 32-bit microprocessor system-on-chip*. pp 159–162, doi:10.1109/ESSCIRC.2011.6044889

[45] Ishii Y., Lee J., Nathella K., Sunwoo D., 2021, in 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). *Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective*. pp 172–182, doi:10.1109/ISPASS51385.2021.00034

[46] Jain S., et al., 2012, in 2012 IEEE International Solid-State Circuits Conference. *A 280mV-to-1.2V wide-operating-range IA-32 processor in 32nm CMOS*. pp 66–68, doi:10.1109/ISSCC.2012.6176932

[47] Jie C., Loi I., Benini L., Rossi D., 2020, in 2020 Design, Automation Test in Europe Conference Exhibition (DATE). *Energy-Efficient Two-level Instruction Cache Design for an Ultra-Low-Power Multi-core Cluster*. pp 1734–1739, doi:10.23919/DATE48585.2020.9116212

[48] Jimenez D., Lin C., 2001, in Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture. *Dynamic branch prediction with perceptrons*. pp 197–206, doi:10.1109/HPCA.2001.903263

[49] Jiménez V., Gioiosa R., Cazorla F. J., Buyuktosunoglu A., Bose P., O’Connell F. P., 2012, in 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT). *Making data prefetch smarter: Adaptive prefetching on POWER7*. pp 137–146

[50] Joel Hruska 2021, *How L1 and L2 CPU Caches Work, and Why They’re an Essential Part of Modern Chips*

[51] Joseph D., Grunwald D., 1997, *Prefetching Using Markov Predictors*, SIGARCH Comput. Archit. News, 25, 252–263

[52] Jouppi N. P., 1990, in Proceedings of the 17th Annual International Symposium on Computer Architecture. *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers*, ISCA '90. Association for Computing Machinery, New York, NY, USA, p. 364–373, doi:10.1145/325164.325162, https://doi.org/10.1145/325164.325162

[53] Kadayif I., Zorlubas A., Koyuncu S., Kabal O., Akcicek D., Sahin Y., Kandemir M. T., 2008, *Capturing and optimizing the interactions between prefetching and cache line turnoff*, Microprocess. Microsystems, 32, 394

[54] Kandiraju G., Sivasubramaniam A., 2002, in Proceedings 29th Annual International Symposium on Computer Architecture. *Going the distance for TLB prefetching: an application-driven study*. pp 195–206, doi:10.1109/ISCA.2002.1003578

[55] Kanter D., 2016, in RISC-V organization announcement. *RISC-V OFFERS SIMPLE, MODULAR ISA*. https://riscv.org/announcements/2016/04/risc-v-offers-simple-modular-isa/

[56] Keckler S. W., Olukotun K., Hofstee H. P., 2009, *Multicore Processors and Systems*, 1st edn. Springer Publishing Company, Incorporated

[57] Kim N. S., Flautner K., Blaauw D., Mudge T., 2004, in Proceedings of the 2004 International Symposium on Low Power Electronics and Design. *Single-v¡sub¿DD¡/sub¿ and Single-v¡sub¿T¡/sub¿ Super-Drowsy Techniques for Low-Leakage High-Performance Instruction Caches*, ISLPED '04. Association for Computing Machinery, New York, NY, USA, p. 54–57, doi:10.1145/1013235.1013254, https://doi.org/10.1145/1013235.1013254

[58] Kim T., Zhao D., Veidenbaum A. V., 2014, in Proceedings of the 11th ACM Conference on Computing Frontiers. *Multiple Stream Tracker: A New Hardware Stride Prefetcher*, CF '14. Association for Computing Machinery, New York, NY, USA, doi:10.1145/2597917.2597941, https://doi.org/10.1145/2597917.2597941

[59] Kin J., Gupta M., Mangione-Smith W., 1997, in Proceedings of 30th Annual International Symposium on Microarchitecture. *The filter cache: an energy efficient memory structure*. pp 184–193, doi:10.1109/MICRO.1997.645809

[60] Kolli A., Saidi A., Wenisch T. F., 2013, in 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). *RDIP: Return-address-stack Directed Instruction Prefetching*. pp 260–271

[61] Kuan K., Adegbija T., 2019, *HALLS: An Energy-Efficient Highly Adaptable Last Level STT-RAM Cache for Multicore Systems*, IEEE Transactions on Computers, 68, 1623

[62] Kumar M. A., Francis G. A., 2017, in 2017 4th International Conference on Electronics and Communication Systems (ICECS). *Survey on various advanced technique for cache optimization methods for risc based system architecture*. pp 195–200, doi:10.1109/ECS.2017.8067868

[63] Kwak J. W., Jeon Y. T., 2010, *Compressed tag architecture for low-power embedded cache systems*, Journal of Systems Architecture, 56, 419

[64] Lai A.-C., Fide C., Falsafi B., 2001, *Dead-block prediction & dead-block correlating prefetchers*, Proceedings 28th Annual International Symposium on Computer Architecture, pp 144–154

[65] Li L., Kadayif I., Tsai Y.-F., Vijaykrishnan N., Kandemir M., Irwin M., Sivasubramaniam A., 2002, in Proceedings.International Conference on Parallel Architectures and Compilation Techniques. *Leakage energy management in cache hierarchies*. pp 131–140, doi:10.1109/PACT.2002.1106012

[66] Loi I., Rossi D., Haugou G., Gautschi M., Benini L., 2015, in Proceedings of the 12th ACM International Conference on Computing Frontiers. *Exploring Multi-banked shared-L1 Program Cache on Ultra-low Power, Tightly Coupled Processor Clusters*, CF '15. ACM, New York, NY, USA, pp 64:1–64:8, doi:10.1145/2742854.2747288, `http://doi.acm.org/10.1145/2742854.2747288`

[67] Loi I., Capotondi A., Rossi D., Marongiu A., Benini L., 2018, *The Quest for Energy-Efficient I$ Design in Ultra-Low-Power Clustered Many-Cores*, IEEE Transactions on Multi-Scale Computing Systems, 4, 99

[68] Loquercio A., Maqueda A. I., del Blanco C. R., Scaramuzza D., 2018, *DroNet: Learning to Fly by Driving*, IEEE Robotics and Automation Letters, 3, 1088

[69] Marongiu A., Capotondi A., Tagliavini G., Benini L., 2015, *Simplifying Many-Core-Based Heterogeneous SoC Programming With Offload Directives*, IEEE Transactions on Industrial Informatics, 11, 957

[70] Meinerzhagen P., Sherazi S. M. Y., Burg A., Rodrigues J. N., 2011, *Benchmarking of Standard-Cell Based Memories in the Sub-V_TDomain in 65-nm CMOS Technology*, IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 1, 173

[71] Michaud P., 2016, in 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). *Best-offset hardware prefetching*. pp 469–480, doi:10.1109/HPCA.2016.7446087

[72] Mittal S., 2013, *A survey of techniques for improving energy efficiency in embedded computing systems*, International Journal of Computer Aided Engineering and Technology, 6

[73] Mittal S., 2014, *A survey of architectural techniques for improving cache power efficiency*, Sustainable Computing: Informatics and Systems, 4, 33

[74] Mittal S., 2016, *A Survey of Recent Prefetching Techniques for Processor Caches*, ACM Comput. Surv., 49

[75] Myers J., Savanth A., Gaddh R., Howard D., Prabhat P., Flynn D., 2016, *A Subthreshold ARM Cortex-M0+ Subsystem in 65 nm CMOS for WSN Applications with 14 Power Domains, 10T SRAM, and Integrated Voltage Regulator*, IEEE Journal of Solid-State Circuits, 51, 31

[76] Nesbit K., Smith J., 2004, in 10th International Symposium on High Performance Computer Architecture (HPCA'04). *Data Cache Prefetching Using a Global History Buffer*. pp 96–96, doi:10.1109/HPCA.2004.10030

[77] Oboril F., Bishnoi R., Ebrahimi M., Tahoori M. B., 2015, *Evaluation of Hybrid Memory Technologies Using SOT-MRAM for On-Chip Cache Hierarchy*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 34, 367

[78] OpenMP Architecture Review Board 2008, *OpenMP Application Program Interface Version 3.0*, http://www.openmp.org/mp-documents/spec30.pdf

[79] Patterson D. A., Hennessy J. L., 2014, in Computer Organization & Design The Hardware/Software Interface : Fifth Edition. *Large and Fast: Exploiting Memory Hierarchy*

[80] Petit S., Sahuquillo J., Such J. M., Kaeli D., 2005. *Exploiting Temporal Locality in Drowsy Cache Policies*, CF '05. Association for Computing Machinery, New York, NY, USA, p. 371–377, doi:10.1145/1062261.1062321, https://doi.org/10.1145/1062261.1062321

[81] Powell M., Yang S.-H., Falsafi B., Roy K., Vijaykumar T., 2000, in ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No.00TH8514). *Gated-V/sub dd/: a circuit technique to reduce leakage in deep-submicron cache memories*. pp 90–95, doi:10.1109/LPE.2000.155259

[82] Przybylski S., Horowitz M., Hennessy J., 1989a, in Proceedings of the 16th Annual International Symposium on Computer Architecture. *Characteristics of Performance-Optimal Multi-Level Cache Hierarchies*, ISCA '89. Association for Computing Machinery, New York, NY, USA, p. 114–121, doi:10.1145/74925.74939, https://doi.org/10.1145/74925.74939

[83] Przybylski S., Horowitz M., Hennessy J., 1989b, *Characteristics of Performance-Optimal Multi-Level Cache Hierarchies*, SIGARCH Comput. Archit. News, 17, 114–121

[84] Pugsley S. H., et al., 2014, in 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). *Sandbox Prefetching: Safe run-time evaluation of aggressive prefetchers*. pp 626–637, doi:10.1109/HPCA.2014.6835971

[85] Pullini A., Rossi D., Loi I., Tagliavini G., Benini L., 2019, *Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing*, IEEE Journal of Solid-State Circuits, 54, 1970

[86] Qureshi M. K., Patt Y. N., 2006, in 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06). *Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches*. pp 423–432, doi:10.1109/MICRO.2006.49

[87] Rahimi A., Loi I., Kakoee M. R., Benini L., 2011, in 2011 Design, Automation Test in Europe. *A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters*. pp 1–6, doi:10.1109/DATE.2011.5763085

[88] Reinman G., Calder B., Austin T., 1999, in MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture. *Fetch directed instruction prefetching*. pp 16–27, doi:10.1109/MICRO.1999.809439

[89] Ros A., Jimborean A., 2020, *The Entangling Instruction Prefetcher*, IEEE Computer Architecture Letters, 19, 84

[90] Ros A., Jimborean A., 2021, in 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). *A Cost-Effective Entangling Prefetcher for Instructions*. pp 99–111, doi:10.1109/ISCA52012.2021.00017

[91] Rossi D., Loi I., Haugou G., Benini L., 2014, in Proceedings of the 11th ACM Conference on Computing Frontiers. *Ultra-Low-Latency Lightweight DMA for Tightly Coupled Multi-Core Clusters*, CF '14. Association for Computing Machinery, New York, NY, USA, doi:10.1145/2597917.2597922, `https://doi.org/10.1145/2597917.2597922`

[92] Rossi D., et al., 2015a, in 2015 IEEE Hot Chips 27 Symposium (HCS). *PULP: A parallel ultra low power platform for next generation IoT applications*. pp 1–39, doi:10.1109/HOTCHIPS.2015.7477325

[93] Rossi D., Pullini A., Loi I., Gautschi M., K. Gürkaynak F., Bartolini A., Flatresse P., Benini L., 2015b, *A 60 GOPS/W, -1.8V to 0.9V body bias ULP cluster in 28nm UTBB FD-SOI technology*, Solid-State Electronics, 117

[94] Rossi D., Loi I., Pullini A., Müller C., Burg A., Conti F., Benini L., Flatresse P., 2017a, *A Self-Aware Architecture for PVT Compensation and Power Nap in Near Threshold Processors*, IEEE Design Test, 34, 46

[95] Rossi D., et al., 2017b, *Energy-Efficient Near-Threshold Parallel Computing: The PULPv2 Cluster*, IEEE Micro, 37, 20

[96] Rossi D., et al., 2021, in 2021 IEEE International Solid- State Circuits Conference (ISSCC). *4.4 A 1.3TOPS/W @ 32GOPS Fully Integrated 10-Core SoC for IoT End-Nodes with 1.7μW Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode*. pp 60–62, doi:10.1109/ISSCC42613.2021.9365939

[97] Sair S., Sherwood T., Calder B., 2002, in Proceedings Eighth International Symposium on High Performance Computer Architecture. *Quantifying load stream behavior*. pp 197–208, doi:10.1109/HPCA.2002.995710

[98] Sherwood T., Sair S., Calder B., 2000, in Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000. *Predictor-directed stream buffers*. pp 42–53, doi:10.1109/MICRO.2000.898057

[99] Shi W., Cao J., Zhang Q., Li Y., Xu L., 2016, *Edge Computing: Vision and Challenges*, IEEE Internet of Things Journal, 3, 637

[100] Smith A. J., 1982, *Cache Memories*, ACM Comput. Surv., 14, 473–530

[101] Smith J., Hsu W.-C., 1992, in Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing. *Prefetching in supercomputer instruction caches*. pp 588–597, doi:10.1109/SUPERC.1992.236645

[102] Srivastava N. K., Navalakha A. D., 2018, *Pointer-Chase Prefetcher for Linked Data Structures* (arXiv:1801.08088)

[103] Steinke S., Grunwald N., Wehmeyer L., Banakar R., Balakrishnan M., Marwedel P., 2002, in Proceedings of the 15th International Symposium on System Synthesis. *Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory*, ISSS '02. Association for Computing Machinery, New York, NY, USA, p. 213–218, doi:10.1145/581199.581247, https://doi.org/10.1145/581199.581247

[104] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya. 2020, *D-JOLT: Distant jolt prefetcher again*, The 1st Instruction Prefetching Championship (IPC-1)

[105] Tanaka K., Matsuda A., 2006, in TENCON 2006 - 2006 IEEE Region 10 Conference. *Static Energy Reduction in Cache Memories Using Data Compression.* pp 1–4, doi:10.1109/TENCON.2006.343807

[106] Teman A., Rossi D., Meinerzhagen P., Benini L., Burg A., 2015, in The 20th Asia and South Pacific Design Automation Conference. *Controlled placement of standard cell memory arrays for high density and low power in 28nm FD-SOI.* pp 81–86, doi:10.1109/ASPDAC.2015.7058985

[107] Teman A., Rossi D., Meinerzhagen P., Benini L., Burg A., 2016, *Power, Area, and Performance Optimization of Standard Cell Memory Arrays Through Controlled Placement*, ACM Trans. Des. Autom. Electron. Syst., 21, 59:1

[108] Tsai Y.-Y., Chen C.-H., 2011, *Energy-Efficient Trace Reuse Cache for Embedded Processors*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 19, 1681

[109] Vanderwiel S. P., Lilja D. J., 2000, *Data Prefetch Mechanisms*, ACM Comput. Surv., 32, 174–199

[110] Verma N., Chandrakasan A. P., 2007, in 2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. *A 65nm*

8T Sub-Vt SRAM Employing Sense-Amplifier Redundancy. pp 328–606, doi:10.1109/ISSCC.2007.373427

[111] Viswanathan V., , *Disclosure of H/W prefetcher control on some Intel processors*, `https://software.intel.com/en-us/articles/disclosure-%20of-hw-prefetcher-control-on-some-intel-processors`

[112] Wong H., Papadopoulou M.-M., Sadooghi-Alvandi M., Moshovos A., 2010, in 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS). *Demystifying GPU microarchitecture through microbenchmarking*. pp 235–246, doi:10.1109/ISPASS.2010.5452013

[113] Yang S., Powell M., Falsafi B., Roy K., Vijaykumar T., 2001, in Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture. *An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches*. pp 147–157, doi:10.1109/HPCA.2001.903259

[114] Yang S.-H., Powell M., Falsafi B., Vijaykumar T., 2002, in Proceedings Eighth International Symposium on High Performance Computer Architecture. *Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay*. pp 151–161, doi:10.1109/HPCA.2002.995706

[115] Yeh T.-Y., Patt Y. N., 1991, in Proceedings of the 24th Annual International Symposium on Microarchitecture. *Two-Level Adaptive Training Branch Prediction*, MICRO 24. Association for Computing Machinery, New York, NY, USA, p. 51–61, doi:10.1145/123465.123475, `https://doi.org/10.1145/123465.123475`

[116] Zhang C., Vahid F., Najjar W., 2003, in 30th Annual International Symposium on Computer Architecture, 2003. Proceedings.. *A highly configurable cache architecture for embedded systems*. pp 136–146, doi:10.1109/ISCA.2003.1206995

[117] Zhang C., Vahid F., Najjar W., 2005, *A Highly Configurable Cache for Low Energy Embedded Systems*, ACM Trans. Embedded Comput. Syst., 4, 363