UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

CAIO RODRIGO DE ALMEIDA VIEIRA

Applying Decoupled Instruction Offloading to Enhance Asymmetric Multi-cores

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Antonio Carlos Schneider
Beck

Porto Alegre
August 2022

CIP — CATALOGING-IN-PUBLICATION

Vieira, Caio Rodrigo de Almeida

Applying Decoupled Instruction Offloading to Enhance Asymmetric Multi-cores / Caio Rodrigo de Almeida Vieira. – Porto Alegre: PPGC da UFRGS, 2022.

92 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Advisor: Antonio Carlos Schneider Beck.

1. Asymmetric multi-core. 2. FU sharing. 3. Instruction offloading. 4. ISA. I. Beck, Antonio Carlos Schneider. II. Título.

“Se você quiser, se você se esforçar, se você treinar,
se você entrar de cabeça, se você se concentrar,
nada garante que você vai conseguir.”

— Craque Daniel

AGRADECIMENTOS

Agradeço primeiramente à minha avó, Lúcia, responsável por ter me criado desde o nascimento e aos meus pais, Renato e Lucinete, pelo apoio nas minhas escolhas e compreensão nos meus momentos de ausência. A Deus, pela minha vida e conquistas. Ao professor e orientador Antonio, pelo fundamental apoio e orientação que permitiram a realização deste trabalho.

ABSTRACT

Asymmetric multi-cores (AMC) are an alternative to provide performance and energy efficiency in the same chip. AMC designs have at least two types of processors, a high-performance but power-hungry core and an energy-efficient but low-performance core. AMC supports the same Instruction Set Architecture (ISA) and thus, can execute the same binaries. Besides asymmetry, modern processors present modular ISA extensions to enhance the CPU capabilities for niche applications. For example, ARM's NEON extension features floating-point (FP) and single instruction, multiple data (SIMD) capabilities for ARM processors. In ARM's AMC design big.LITTLE, the NEON extension is implemented with different hardware costs for both core types, the high-performance big and the energy-efficient little. The big core implements two high-performance NEON functional units (FU), whereas the little core implements a simpler NEON FU. Previous works estimate that the area cost to implement both NEON FUs in big is equivalent to four full little cores. However, the NEON extension may be underused in many applications, leading to wasted expensive resources. When considering low NEON usage applications, there is no need to use the high-performance NEON units in big. Instead, a better solution would be to allow the big core to use the energy-efficient NEON unit in little. A common strategy to deal with expensive resources used infrequently is to share them between multiple processors. However, typical FU sharing schemes share the FU at the execute stage of the pipeline. The drawback of this coupled approach is that a processor must send the instruction to the shared FU and wait until the result is ready. In this work, we propose the *decoupled offloader* to allow the big core to use the energy-efficient NEON FU in little without waiting for the instruction to complete. We power gate both NEON units in big and use the offloader to save energy. We also propose an arbiter to detect the current application phase and fall back to use big's NEON units when NEON is used intensively. Moreover, we use the decoupled offloader to propose *partial cores with full ISA*. We create partial cores from big cores by removing its NEON units. We maintain the full ISA capability by using the decoupled offloader to execute the NEON instructions in a little core. The partial cores have the same performance as the big cores for integer instructions requiring a smaller area at the cost of limited NEON performance. We discuss how both ideas can be implemented and their advantages and drawbacks.

Keywords: Asymmetric multi-core. FU sharing. instruction offloading. ISA.

Aplicando Despacho de Instruções Desacoplado para Melhorar Multi-cores Assimétricos

RESUMO

Multi-cores assimétricos (AMC) são uma alternativa para prover desempenho e eficiência energética no mesmo chip. AMCs têm pelo menos dois tipos de processadores: um de alto desempenho, mas baixa eficiência energética, e outro eficiente energeticamente, mas com baixo desempenho. AMCs suportam o mesmo ISA e, portanto, podem executar os mesmos binários. Além da assimetria, processadores modernos apresentam extensões modulares de ISA que permitem melhorar as capacidades de uma CPU em certos nichos de aplicações. A extensão NEON adiciona operações de FP e SIMD para processadores ARM. No caso do AMC big.LITTLE, da ARM, a extensão NEON é implementada com custos diferentes de hardware para ambos os tipos de núcleo, o *big* e o *little*. O núcleo big implementa duas FUs NEON de alto desempenho, enquanto o little implementa apenas uma FU mais simples. Trabalhos anteriores estimam que o custo de área para implementar ambas unidades NEON do big é equivalente a 4 núcleos little completos. Contudo, a extensão NEON pode ser pouco utilizada em muitas aplicações, levando à subutilização de recursos. Quando considerando aplicações que usam levemente o NEON, não há a necessidade de usar as unidades NEON de alto desempenho presentes no big. Ao invés disso, uma solução melhor seria possibilitar o núcleo big usar a unidade NEON do little, que é mais eficiente energeticamente. Uma estratégia comum para lidar com recursos caros que tendem a ser pouco utilizados é compartilhá-los entre múltiplos processadores. Contudo, abordagens tradicionais de compartilhamento de FUs tendem a ser feitas a partir do estágio de execução de um pipeline. A desvantagem dessa abordagem acoplada é que o processador deve enviar a instrução para FU compartilhada e esperar até que o resultado esteja pronto. Este trabalho propõe o *despacho desacoplado de instruções* para permitir que o núcleo big use a unidade NEON do little sem esperar que instrução complete. Dessa forma, é possível aplicar power gate em ambas unidades NEON do big e usar o despacho de instruções para economizar energia. Além disso, o trabalho também propõe um árbitro para detectar a atual fase da aplicação e desligar o despacho quando o NEON precisa ser usado intensamente.

O despacho desacoplado é usado para propor *núcleos parciais com ISA completo*. Os núcleos parciais são formados a partir de núcleos big removendo suas unidades NEON. Por meio do despacho desacoplado é possível manter o suporte completo ao ISA despachando instruções NEON para um núcleo little. Os núcleos parciais têm o mesmo desempenho dos núcleos big para aplicações de inteiro, mas necessitam de menos área e apresentam desempenho limitado com a extensão NEON. Ambas as ideias são discutidas acerca de sua implementação, vantagens e desvantagens.

Palavras-chave: multi-core assimétrico. compartilhamento de FU. despacho de instrução. ISA.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ABI	Application-binary Interface
AI	Artificial Intelligence
AMC	Asymmetric Multi-core
BTB	Branch Target Buffer
CAFIFO	Content-addressable FIFO
CCI	Cache Coherent Interconnect
CSD	Context-sensitive Decoding
DLP	Data-level Parallelism
DVFS	Dynamic Voltage Frequency Scaling
EDP	Energy-Delay Product
FP	Floating Point
FPSCR	Floating-Point Status Control Register
FU	Functional Unit
GPP	General Purpose Processor
GPR	General Purpose Register
GTS	Global Task Scheduler
HFI	Hardware Feedback Interface
ILP	Instruction-level Parallelism
InO	In Order
IPS	Instructions per Second
ISA	Instruction Set Architecture
LLC	Last-level Cache
MRaCW	Main Read after Coprocessor Write
MWaCR	Main Write after Coprocessor Read

MWaCW	Main Write after Coprocessor Write
OoO	Out of Order
OS	Operating System
PC	Program Counter
RD	Result Distribution
RS	Reservation Station
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SMT	Simultaneous Multithreading
TLB	Translation Lookaside Buffer
TLP	Threa-level Parallelism
TUNE	Tightly Coupled Instruction Offloader
VNNI	Vector Neural Network Instructions
vSMP	Variable Symmetric Multiprocessing

CONTENTS

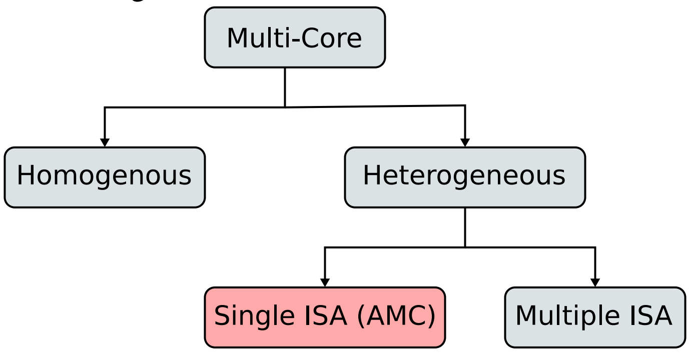
1 INTRODUCTION

The always present demand for performance has pushed computer systems evolution. Initially, the main optimization target was to increase sequential program execution, which was achieved by increasing the clock and novel microarchitectural solutions (PATTERSON, 2010). The simple in-order (InO) processors gave space to complex out-of-order (OoO) datapaths to exploit instruction-level parallelism (ILP). However, such complex designs combined with high clock rates lead to high power consumption. To solve the problem, the industry shifted the General Purpose Processor (GPP) design from single-core to multi-core (PARKHURST et al., 2006). This new paradigm improves performance by allowing multiple applications to run simultaneously, thus increasing the thread-level parallelism (TLP) of the system.

Multi-core systems can be divided into homogeneous and heterogeneous. Homogeneous multi-cores consist of the same core replicated multiple times, which leads to simpler designs since all cores are identical. On the other hand, heterogeneous multi-cores can be further classified into two categories: Single ISA and multiple ISA (KUMAR et al., 2003), as depicted in Figure 1.1. Single-ISA multi-cores are also known as asymmetric multi-cores (AMC) since all cores support the same binaries but have different capabilities. The key advantage of such designs is to allow more efficient application execution according to its requirements. It is also possible to design heterogeneous multi-cores with different ISA, further increasing the diversity in the chip.

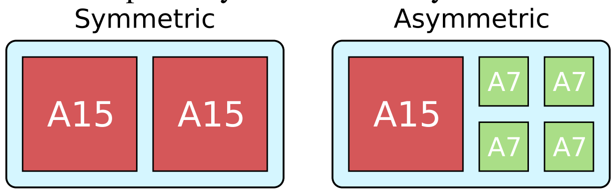
Commercial designs implement asymmetric multi-cores using at least two different cores: A high-performance big and an energy-efficient little core, as shown in Figure 1.2. When considering ARM big.LITTLE (bigLITTLE, 2011), the big core is an A15 OoO processor that targets performance. In contrast, the little core is an A7 processor

Figure 1.1: Multi-cores classification.



Source: The author.

Figure 1.2: Example of symmetric and asymmetric multi-cores.

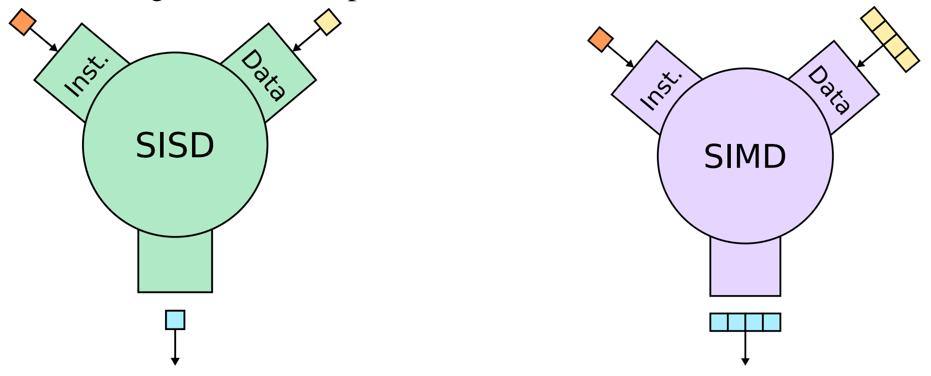


Source: The author.

which has a simple InO datapath to provide energy-efficient application execution. The combination of both cores on the same chip allows each application to run on the right core according to the current purpose of the system, i.e., performance or energy. More recent commercial designs that also follow this strategy are AppleM1 (Apple, 2020) and Intel Alder Lake (ROTEM et al., 2021). In such designs, the asymmetry lies only in processors microarchitecture. Thus, all cores can execute the same instructions since they maintain the same Instruction Set Architecture (ISA) support, allowing any application to run on any type of core without restrictions. This key insight, combined with the ISA compatibility to execute already existent binaries, permits the rapid adoption of asymmetric systems.

Besides multi-core asymmetry, which lies only at the microarchitectural level, many ISAs dispose of extensions targeting niche applications to improve performance. Common examples are the Floating Point (FP) and the Single Instruction, Multiple Data (SIMD) extensions. Unlike Single Instruction, Single Data (SISD), SIMD allows performing computation on multiple pieces of data using a single instruction. The main advantage of this approach over SISD is that it exposes the data-level parallelism (DLP) to the programmer, as depicted in Figure 1.3. Moreover, it provides reduced instruction and memory bandwidth. FP and SIMD instructions are frequently used in math-intensive applications, e.g., scientific and multimedia, and can greatly enhance application execution time. However, supporting such ISA extensions comes with hardware costs. When considering AMCs, the big and little cores usually implement ISA extensions differently. That is the case for the ARM big.LITTLE with its A7 (little) and A15 (big) processors. According to (SOUZA et al., 2020), both NEON units in the big core occupy more area than 4 little cores. This area difference comes not only from the OoO engine present in big, but also from the different Functional Units (FU) implemented. For example, the big core provides two FUs to support the NEON extension, which features FP and SIMD capabilities to the ARM ISA. Each big's NEON FU occupies the area of a fully capable little.

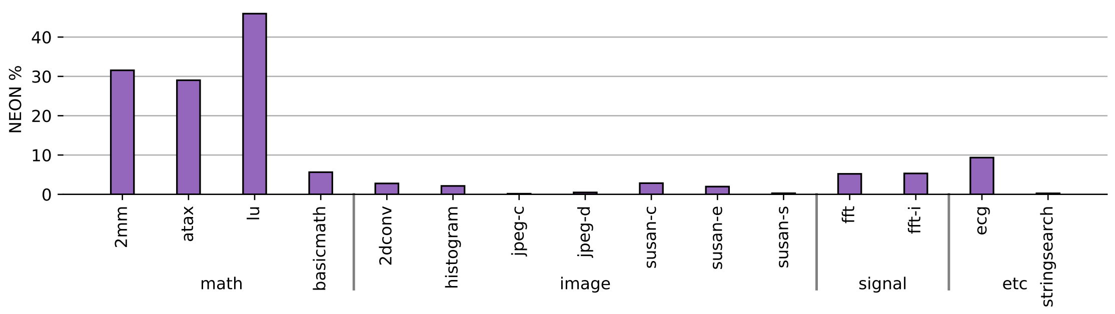
Figure 1.3: Example of SISD and SIMD instructions.



(a) Single Instruction, Single Data (SISD) (b) Single Instruction, Multiple Data (SIMD)

Source: The author.

Figure 1.4: Percentage of NEON instructions in different classes of applications.



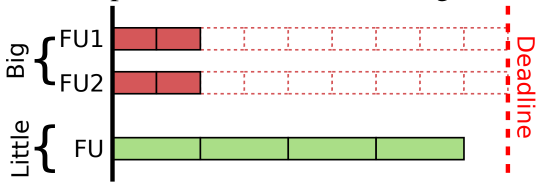
Source: The author.

However, such expensive hardware resources may not be used in all applications. We evaluate the percentage of NEON instructions in different application classes, as shown in Figure 1.4. Although some applications use NEON intensively, as in the linear algebra program *lu* (lower-upper decomposition), the majority of the evaluated programs have less than 10% of NEON instructions. Thus, this observation opens room for improvements considering the low NEON utilization. In the next sections, we present our ideas explored in this work to optimize the NEON usage in asymmetric systems with a single ISA considering an ARM big.LITTLE inspired system.

1.1 Decoupled Offloader

Since NEON instructions are expensive and not used in all applications, their execution units may be underutilized for a long time, wasting power and increasing energy consumption. In symmetric designs, a common solution to deal with such units is to share them between multiple processors. In the conventional approach, two processors

Figure 1.5: Example of FU utilization in big and little cores.



Source: The author.

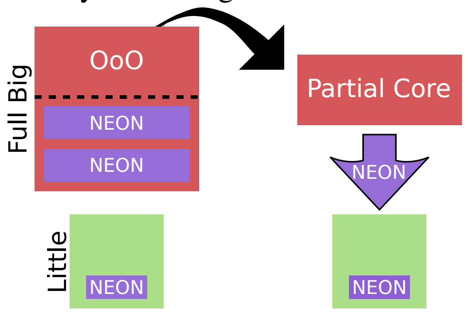
share a FU at the execution stage of the pipeline. Each core issues its instruction into the shared FU, and the result is returned to resume its passage through the following stages. The main problem arises when the shared execution unit takes too long to finish since it may hinder the processor's performance. This *coupled* strategy is used in designs as the Bulldozer (BUTLER et al., 2011) and the UltraSPARC T1 (Sun Microsystems, 2008).

In asymmetric multi-cores with the same ISA, all cores support the same instructions but with different microarchitectural costs. Since NEON tends to be underutilized in some applications (Figure 1.4), the best sharing strategy is to let the big core use the energy-efficient but slow unit present in the little core when the current application does not use NEON intensively. However, a coupled sharing may hinder big's performance since it has to wait for the instruction completion. In the ideal scenario, the big core would only schedule its instructions to execute in the shared FU and resume its computation without waiting for the instruction to complete. As an example, let us suppose a simple array addition program that uses the NEON extension to sum two arrays, A and B, to form a new array C. As shown in Figure 1.5, executing this sum on the big core results in low execution time since big dispose of two fast NEON FUs, whereas little has just a single and slow unit. In either case, both computations finish before the deadline, i.e., before the results are actually needed by the program. Thus, this case allows the big core to use the energy-efficient FU in little without waiting for the instruction to complete.

Therefore, in this work, we introduce the *decoupled offloader*, a solution to allow the big core only to schedule its NEON instructions to be executed in the energy-efficient little's FU and resume its computation without waiting for the offloaded instruction to complete. This strategy can be implemented with ISA extensions that behave as coprocessors, which is the case of NEON. In such extensions, the architectural state of the base ISA is extended with new registers that are used by the extension instructions.

The main state, defined by the base ISA, comprises the General Purpose Registers (GPR) and the control registers, e.g., the Program Counter (PC) and flags, and dictates the control flow. The coprocessor state adds new registers and instructions to access them. The communication between both states is done by specific move instructions or through

Figure 1.6: We form partial cores from big cores by removing their NEON units. The full-ISA support is maintained by offloading NEON instructions to little cores.



Source: The author.

memory. Therefore, the main state can dictate instructions to its coprocessor state as long as the data movements between both states are kept. We use this observation to enable the decoupled offloader, as further discussed in Section 3.1.

1.2 Partial Cores with Full ISA

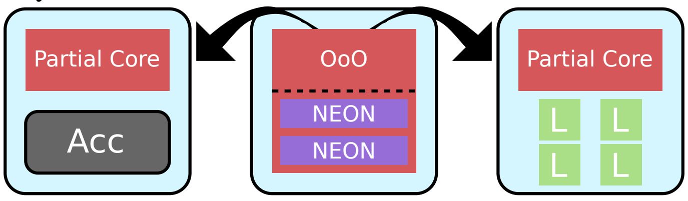
Since NEON instructions may be underused or not even used in many applications (as discussed in Section 1), we propose creating Partial Cores with Full ISA by completely removing the NEON units of big cores and equipping them with the decoupled offloader, as depicted in Figure 1.6. The name *partial core* stems from the fact that now the cores do not implement the necessary hardware structures to support the entire ISA and thus, should overcome this problem with the decoupled offloader. This idea adds a new level of heterogeneity to the system in a multi-core scenario: the big, the partial, and the little. The partial core provides the same performance as the big core for integer applications since both have an OoO engine at the cost of slower support for NEON instructions.

Since the NEON units in the big core are equivalent to 4 little cores, the freed space provided by partial cores can be used to improve the overall system's TLP by adding new little cores. Another option is to improve the system's performance by including accelerators. This idea is summarized in Figure 1.7. We discuss the trade-offs of partial cores with full-ISA in Section 3.3.

1.3 Structure of this Thesis

The structure of this work is organized as follows. Chapter 2 explains the necessary background and presents the related work. We discuss studies and commercial

Figure 1.7: The freed area due to partial cores can be used to incorporate accelerators or little cores into the system.



Source: The author.

designs in different areas: AMC, ISA extensions, resource sharing in multi-cores, and power gating. Chapter 3 introduces the proposed work. First, we explain the decoupled offloader and how it can be implemented considering coprocessors like ISAs such as the NEON extension. Subsequently, we explain how it can be used to enable partial cores with full ISA. Chapter 4 presents the simulation tools used to evaluate this work and the methodology. Chapter 5 presents simulated results of the decoupled offloader and partial cores with full ISA. Finally, chapter 6 compiles the conclusions of this work.

2 BACKGROUND AND RELATED WORK

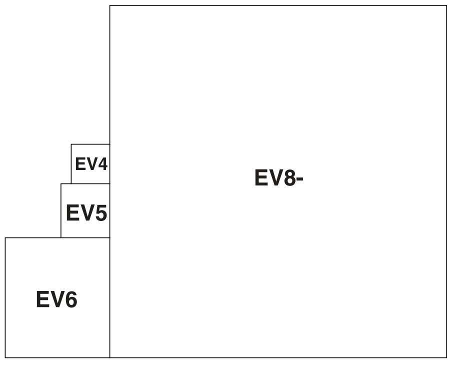
In this chapter, we present the necessary background and discuss the related work. In Section 2.1 we discuss asymmetric multi-cores and their importance in providing performance and energy efficiency on the same chip. Section 2.2 brings a discussion on ISA extensions and their advantages and disadvantages on moderns processors. Section 2.3 presents works related to power-gating and its capabilities to provide power and energy efficiency.

2.1 Heterogeneous Multi-core

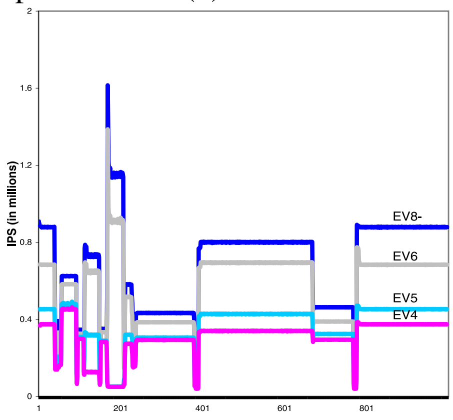
In (KUMAR et al., 2003), the authors introduce heterogeneous multi-core with a single ISA as an alternative to reduce power consumption in processor designs. The paper presents a multi-core architecture with the same ISA across all processors but with different microarchitectures. The authors evaluate 4 different types of Alpha cores: EV4 (Alpha 21064), EV5 (Alpha 21164), EV6 (Alpha 21264), and a single-threaded version of the EV8 (Alpha 21464). Figure 2.1a. shows the relative size of each core. The motivation behind this idea is that applications may have different amount of instruction-level parallelism (ILP), which should be exploited by the right core. When this is not the case, a low ILP application on a high ILP core leads to wasted power and minimal performance gains, whereas a high ILP app on low ILP core results in low performance. Figure 2.1b shows the instructions committed per second (IPS) of an application evaluated in the work. The EV8 core can execute much more instructions than EV4 in some applications phases, but in others, the difference between both cores is low. The work concludes that having at least two distinct cores in the system is enough to achieve most of the possible energy gains without dramatic performance losses.

Therefore, AMC systems depend on a clever utilization of each core to obtain gains. This can be achieved by using thread migration at the hardware or operating system (OS) scheduler levels. The latter approach may take a long time to happen due to the OS intervention granularity. Thus, leading to migrations at coarse-grained intervals. In (LUKEFAHR et al., 2012), the authors tackle this problem by proposing Composite Cores (Figure 2.2). The idea is to integrate OoO and InO pipelines into the same processor to share common structures such as L1 caches and translation lookaside buffers (TLB). This sharing amortizes the migration overhead because only the register state needs to be

Figure 2.1: Relative core size (a) and performance (b).



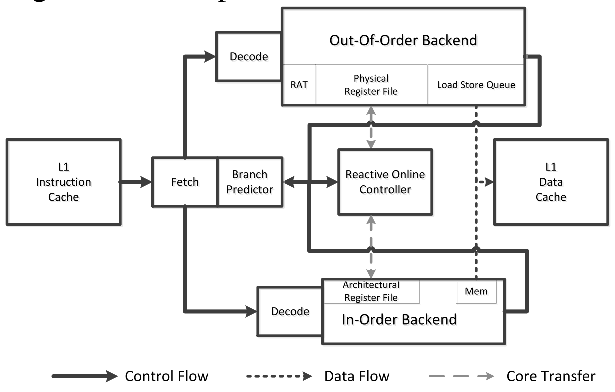
(a)



(b)

Source: (KUMAR et al., 2003)

Figure 2.2: Composite cores microarchitecture.



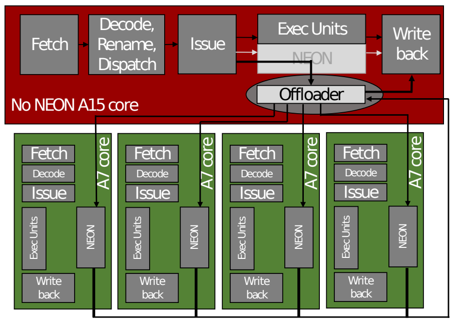
Source: (LUKEFAHR et al., 2012).

moved between the execution units. Furthermore, the process can be handled entirely at the microarchitectural level without the costly OS migrations. This way, it is possible to perform the core switching at fine granularity, enabling better utilization of the energy-efficient core whenever possible.

In (PADMANABHA et al., 2015), the authors use the observation that an InO core can achieve similar OoO performance using less energy if provided with an OoO instruction schedule. To leverage this, the authors propose the DynaMOS architecture. The idea is to integrate a big (OoO) core tightly with an equally provisioned little (InO) core. The OoO engine produces instructions schedules that are saved in a schedule-trace cache so that the InO engine can use it. (PADMANABHA et al., 2017) present Mirage Cores, which extends that concept to a multicore scenario with multiple InO cores and one OoO core. The goal is to use the OoO engine as a shared resource to produce traces to the multiple InO cores. Therefore, the InO cores have performance similar to the OoO.

AMC systems can provide great improvements in parallel applications, which

Figure 2.3: TUNE offloader. The NEON units of the A7 cores are shared with the A15 core.



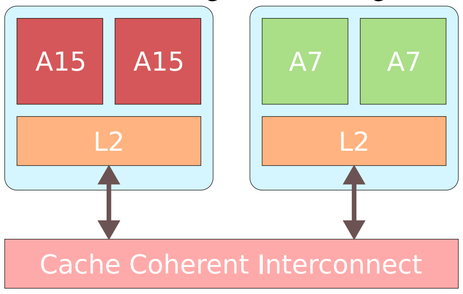
Source: (SOUZA et al., 2020).

have serial and parallel regions. The big cores are well suited for serial regions since they can execute them in a burst. In contrast, the little cores can execute the parallel regions, which benefit from high TLP, more energy efficient. In (SOUZA et al., 2020), the authors observe that serial regions contain few NEON instructions, whereas parallel regions tend to use NEON frequently. Therefore, the authors propose to remove the NEON units from the big A15 core to open space for little cores. This approach uses the big core in the serial regions and the little cores in the parallel regions. With the same area budget, the authors can significantly improve the performance and energy efficiency of parallel applications. To cope with the absence of NEON units in the big core, the authors propose the **t**ightly co**u**pled i**n**struction offloa**de**r (TUNE), depicted in Figure 2.3. This mechanism allows the big core to offload any NEON instruction found when executing the serial region to a little core. The instruction is offloaded at the execute stage of the big core, which must wait until the result is ready before moving on. Thus, the big core still maintains its full-ISA compatibility.

When considering heterogeneous multi-cores with **heterogeneous ISAs**, Venkat and Tullsen (VENKAT; TULLSEN, 2014) investigate how different instruction sets can improve application execution. The work evaluates three ISAs: ARM's Thumb, x86, and Alpha. Each ISA is better appropriated to a specific task. For example, Thumb provides high code density since its instructions are 16-bit, but it does not feature FP/SIMD instructions. On the other hand, x86 provides FP/SIMD instructions with a good code density. Alpha defines a large architectural state, Thus, having a high number of registers available to the programmer. The authors conclude that having multiple ISAs in a multi-core design can significantly increase the performance and energy efficiency since applications tend to have multiple phases, and each one can be better suited to a given ISA.

The previous work is extended in Composite-ISA cores (VENKAT; BASAVARAJ;

Figure 2.4: ARM big.LITTLE organization.



Source: The author.

TULLSEN, 2019). As in the previous work, the goal is to enable multi-ISA heterogeneity. However, this work considers a single ISA instead of the previous approach, which considers three commercial ISAs. The authors propose an ISA based on the x86 that also has features present in Thumb and Alpha. The proposed ISA can be derived in multiple cores, each implementing an instruction set feature. Thus, it is possible to isolate the specific ISA features that improve the single thread performance. Another advantage over the previous work is that different proprietary ISAs are no longer necessary, reducing licensing, legal, and verification costs. Moreover, the process migration between cores is simplified since they share the same application-binary interface (ABI).

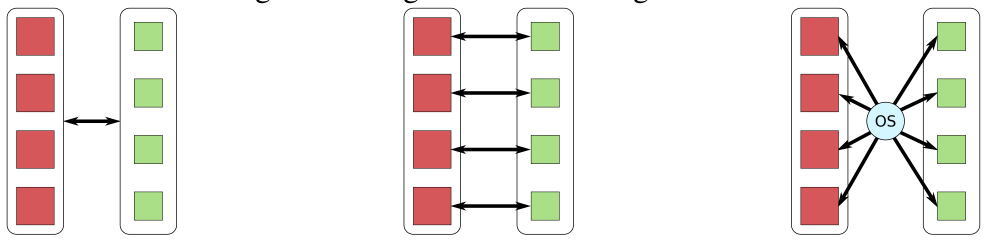
Commercial designs

ARM big.LITTLE (bigLITTLE, 2011) is an industry implementation of the single-ISA heterogeneous multi-core. The system is composed of *big* (OoO, larger, high performance) and *LITTLE* (InO, smaller, energy-efficient) cores. Figure 2.4 shows an example of a big.LITTLE system with two clusters. Each cluster contains two cores of each type, big and little, and an L2 cache. Both clusters are connected by a Cache Coherent Interconnect (CCI). This interconnection allows cache coherence between both clusters' L2 caches and reduces the thread migration impact since the data does not need to be moved through memory.

The big.LITTLE allows three working modes divided into two execution models: Migration and Global Task Scheduler (GTS). Figure 2.5 shows all modes available.

- **Migration:** The migration model comprises two types, *cluster migration* and *CPU migration*, and is a natural extension of power-performance management techniques such as Dynamic Voltage Frequency Scaling (DVFS).
  - **Cluster Migration:** Only one cluster must be active at a time. If high per-

Figure 2.5: big.LITTLE working modes.



(a) Cluster Migration (b) CPU Migration (c) Global Task Scheduling

Source: The author.

  formance is required, then the big cluster is used. When this is not the case, the big cluster is turned off, and the little cluster is used. This model does not cope well with unbalanced software workloads since a heavily loaded core will require a cluster migration to the big cluster even if the other cores are lightly loaded, resulting in big cluster underutilization.

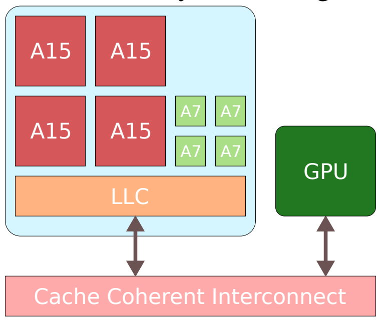
  - **CPU Migration:** In this model, each big core is paired with a little core, but only one type of processor can be active at a time. The working core is chosen according to the DVFS. Thus, the little core is used if a lightly loaded application is being executed. If the application load increases, the system switches the application to the big core and turns off the little core. This model requires the same number of cores on each cluster.

- **Global Task Scheduler:** The main difference of this mode is that the OS is aware of the microarchitectural difference between the cores, and it is responsible for the job assignment. Moreover, all cores can be active simultaneously, and the system can have a different number of big and little cores.

ARM's DynamIQ (DynamIQ, 2017) extends the big.LITTLE features by joining different cores into the same cluster, as depicted in Figure 2.6. In DynamIQ, big and little cores share a Last-Level Cache (LLC). This allows for faster thread migrations between the cores. Thus, improving the support for GTS from the previous big.LITTLE. The system also features a CCI to keep data coherency between the cluster and other SoC components such as the GPU and accelerators.

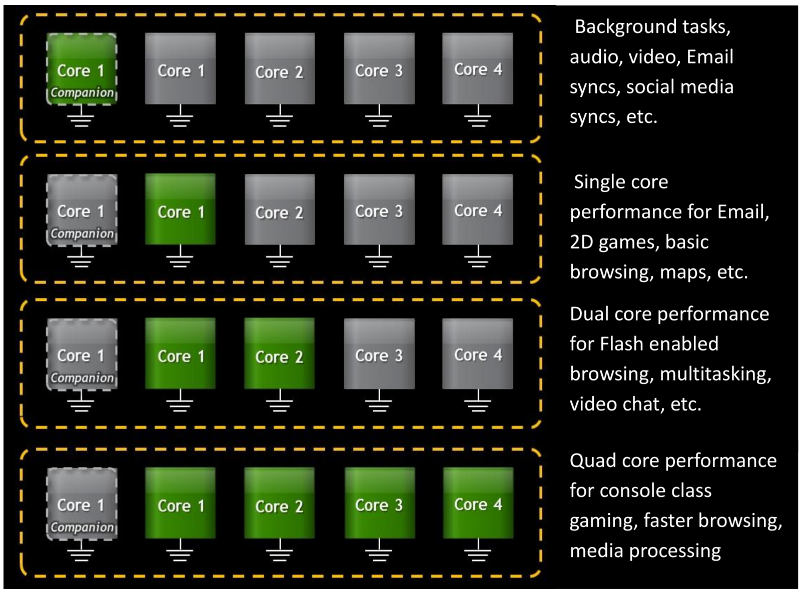
NVidia's project Kal-El (NVIDIA, 2011) is an asymmetric design that implements a Variable Symmetric Multiprocessing (vSMP) technology. The chip contains 5 ARM's A9 CPUs in a 4+1 configuration, i.e., 4 cores for high performance and 1 core for energy efficiency. The energy-efficient core is named *companion core* and is built using a low-power silicon process to reduce leakage consumption. Moreover, it runs on a limited

Figure 2.6: ARM DynamIQ organization.



Source: The author.

Figure 2.7: The CPU management is based on the workload. Either the companion core or the high-performance core can be active at a time.

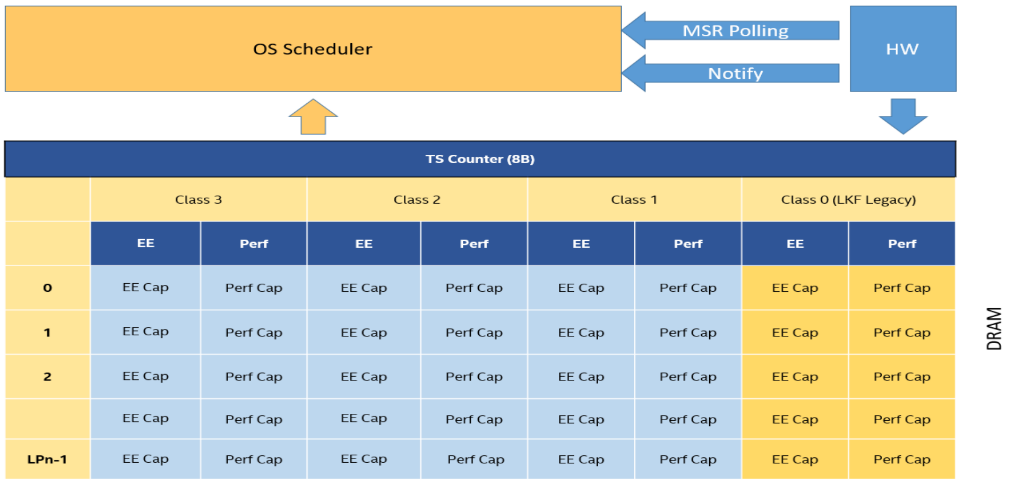

Core 1 Companion
Core 1
Core 2
Core 3
Core 4
Background tasks, audio, video, Email syncs, social media syncs, etc.
Single core performance for Email, 2D games, basic browsing, maps, etc.
Dual core performance for Flash enabled browsing, multitasking, video chat, etc.
Quad core performance for console class gaming, faster browsing, media processing

Source: (NVIDIA, 2011).

frequency of 500 MHz. The companion core is OS transparent, and its management is performed by a hardware/software NVidia's technology. The management is based on the current workload, as shown in Figure 2.7. If lightly loaded, the companion core is used. Kal-El turns off the companion core at a sufficient workload level and switches the application to a fast core. The other cores are awakened proportionally to the current workload. Thus, either the companion core or the fast cores are active at a time. All cores share a common LLC that provides the same access time to all cores to enable fast application migration. However, the companion core accesses it in fewer cycles since it operates at lower frequencies.

Alder Lake (ROTEM et al., 2021) is the first Intel AMC design. The high-performance and the energy-efficient cores are named *P-Cores* and *E-Cores*, respectively. The microarchitecture allows different processor configurations, ranging from mobile to desktops. Thus, mobile configurations can have more E-cores, whereas desktop chips can

Figure 2.8: The hardware maintains a table in memory with hints to the OS scheduler for different classes of applications.



	Class 3		Class 2		Class 1		Class 0 (LKF Legacy)	
	EE	Perf	EE	Perf	EE	Perf	EE	Perf
0	EE Cap	Perf Cap	EE Cap	Perf Cap	EE Cap	Perf Cap	EE Cap	Perf Cap
1	EE Cap	Perf Cap	EE Cap	Perf Cap	EE Cap	Perf Cap	EE Cap	Perf Cap
2	EE Cap	Perf Cap	EE Cap	Perf Cap	EE Cap	Perf Cap	EE Cap	Perf Cap
	EE Cap	Perf Cap	EE Cap	Perf Cap	EE Cap	Perf Cap	EE Cap	Perf Cap
LPn-1	EE Cap	Perf Cap	EE Cap	Perf Cap	EE Cap	Perf Cap	EE Cap	Perf Cap

Source: (RUKMABHATLA et al., 2021).

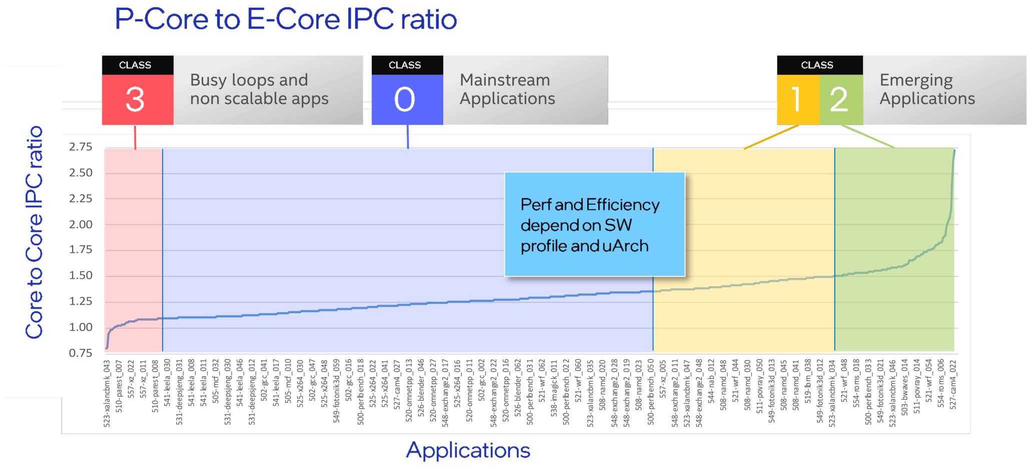
feature a high number of P-cores. The thread migration model is managed by the OS scheduler, similar to the GTS execution model in big.LITTLE. Since scheduling applications is difficult, Intel provides the Intel Thread Director technology. This technology is a hardware solution that monitors the instructions in applications running in all cores and provides hints to the OS scheduler, helping it make the best decision in thread allocation.

Intel's Hardware Feedback Interface (HFI) (INTEL, 2021b) enables the hints behind Thread Director. The interface exposes a table (Figure 2.8) in memory, which is continuously updated by the processor. Each table entry represents a Logical Processor (LP) due to Simultaneous Multithreading (SMT). Each entry is divided into four application class fields, numbered from 0 to 3, and each class exposes the Energy Efficiency (EE), and the Performance (Perf) capabilities of the LP, i.e., how capable is the LP of providing performance or energy efficiency the application class. The application classes are divided into four groups:

- **Class 0:** Applications that perform similarly on P- and E- cores.
- **Class 1:** Applications that use ISA extensions such as AVX2 or FP32, which are more performant on P-cores than on E-cores.
- **Class 2:** Indicates emerging applications (e.g., AI), which are better suited on P-cores.
- **Class 3:** Represent applications that does not scale on high performance cores, e.g., applications that depend heavily on busy loops or I/O operations.

Figure 2.9 shows the P- to E- core IPC ratio, i.e., the IPC increment of the P-core

Figure 2.9: IPC advantage of P-core to E-core for different application classes.



Source: (ROTEM et al., 2021).

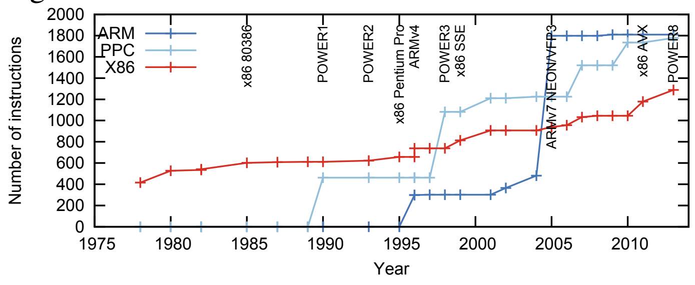
over the E-core, for each application class. Class 3 applications present minor speedup on P-cores, better suited to E-cores. Class 0 programs perform similarly on both cores but present some performance increment on P-cores. Classes 2 and 3 are classified as emerging applications and tend to use ISA extensions with greater performance gains on P-cores.

When the OS scheduler needs to pick an LP between multiple free LPs to assign a thread of a given class X, it reads the entries of free LPs. If, for example, the OS wants to schedule for performance, it looks for the highest Perf capability value for class X among the LPs available. Similarly, if the OS needs to schedule for EE, it searches for the highest EE-capable LP for the given class X. The values in the table are in the range of 0-255, with 0 and 255 meaning "not recommended" and "highly recommended", respectively. These values may be updated at run time, e.g., due to changes in core frequency, voltage, or power budget, causing the hardware to notify the operating system of the new hints.

2.2 ISA extensions and resource sharing

The ISA is the main interface between the software and the underlying hardware. In GPPs, the ISAs are continuously incremented to support new application demands. As an example, Intel AVX-512 (INTEL, 2013) extension, which comprises SIMD instructions, now includes Vector Neural Network Instructions (VNNI) (INTEL, 2021a) to improve machine learning applications. Besides machine learning, ISA extensions enhance GPP features in different ways. Compact instructions extensions allow the processor to operate on reduced size instructions, allowing to reduce the memory footprint of the bi-

Figure 2.10: Number of instructions in different ISAs.



Source: (LOPES et al., 2015).

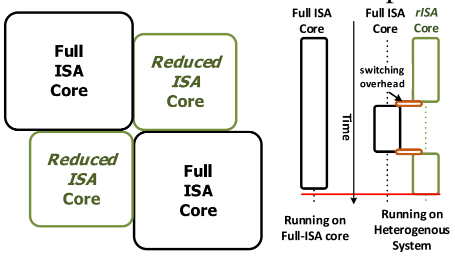
naries. Such extensions are represented by RISC-V "C" extension (WATERMAN et al., 2016) and ARM THUMB (ARM, 2014), which reduce the instruction size to 16 bits, instead of the traditional 32 bits from RISC ISAs.

Other examples of frequently used extensions are FP and SIMD. FP instructions are helpful in scientific computation since it allows the processor to compute real numbers. On the other hand, SIMD instructions can explore data parallelism in an application by performing the same operation on a wide data word. Both types of extensions are present in several ISAs. The x86 has the x87 extension for FP operations and several SIMD extensions starting with the MMX, which was continuously incremented over the years.

Although specialized extensions can improve the performance of specific applications, their implementation can be expensive to the microarchitecture. This problem is even more important when considering backward compatibility. Thus, many works investigate solutions o **amortize ISA extension costs**. When considering the x86 ISA, several extensions were added to the ISA over the decades and all of them are still supported. Figure 2.10 shows the ISA growth in different architectures. In (LOPES et al., 2015), the authors investigate the **ISA aging problem**, which is the trade-off between adding new instructions and supporting old ones. They notice that old still supported instructions, which occupy shorter instruction encodings, are rarely used, whereas new instructions, with larger encoding, can be often used. The work presents SHRINK, a method to recycle unused instructions and reassign their encodings to more frequently used instructions while the recycled instructions are still supported by emulation. This approach allows to reduce the decoder area, critical path, and power consumption of the processor.

When considering AMC, the authors in (LEE et al., 2017) use the observation that some ISA extensions require great microarchitectural costs to propose reduced ISA cores. In the work, the authors consider the ARM-v7 (ARM, 2014) ISA and its extensions. They analyze the costs and performance of a reduced A15 core without some extensions, such

Figure 2.11: Full- and reduced-ISA processors.



Source: (LEE et al., 2017).

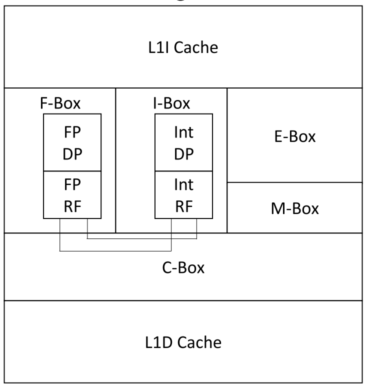
as NEON, load and store multiple, predicated instructions, and DSP-like instructions. The idea is depicted in Figure 2.11. The authors propose prioritizing executing applications on reduced cores to save power and energy. If the reduced core finds an unimplemented instruction, it emulates its behavior. However, the emulation must be done with caution since it drastically reduces the performance. If the application uses the removed instructions intensively, it is swapped to a full core and is swapped back after an interval without issuing those instructions.

In (BECKER; SOUZA; BECK, 2020), the authors observe that both cores in an AMC system support the same instructions but with different implementation costs. The authors show that an OoO RISC-V core can dedicate 37% of its area to support FP instruction. Thus, they propose to create **partial-ISA** OoO cores by removing the FP support. The binary compatibility is preserved in the MPSoC since there are still full cores capable of executing FP instructions. The partial cores open room in the MPSoC for the inclusion of accelerators and energy-efficient little cores. In the work, the authors show how the freed area can be better utilized considering different workloads.

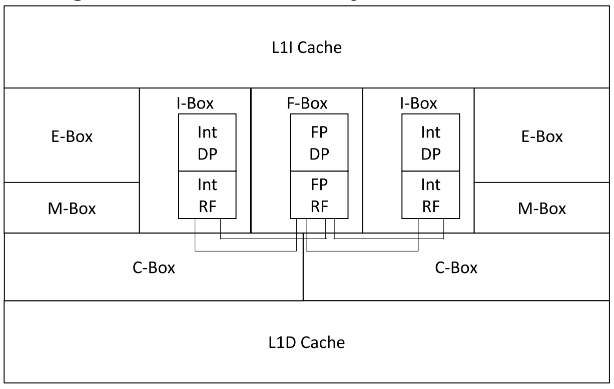
Other works research the possibility of **sharing resources** to implement complex designs. Kumar et al. (KUMAR; JOUPPI; TULLSEN, 2004) investigate different resource sharing strategies in multi-core designs. The work proposes conjoined cores, i.e., a pair of cores that share common resources. By pairing an original core and a mirrored one side-by-side, it is possible to minimize the distance between each core and the shared resources. Figure 2.12a shows an example of a full single-core, and Figure 2.12b shows an example of a conjoined core with shared resources. The authors evaluate the feasibility of sharing the instruction and data caches, the FP execution units, and a crossbar interconnection under different scenarios. They find that by sharing the crossbar ports and the FP FUs, it is possible to reduce the core area by 23% with at the cost of 2% performance impact.

Borodin et al (BORODIN; SIAUW; COTOFANA, 2011) consider FU sharing in

Figure 2.12: Floorplan of the original core and the conjoined core.



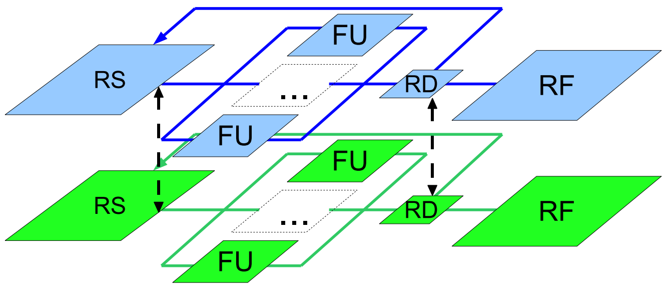
(a) Original core



(b) Conjoined core

Source: (KUMAR; JOUPPI; TULLSEN, 2004).

Figure 2.13: 3D System with FU sharing.

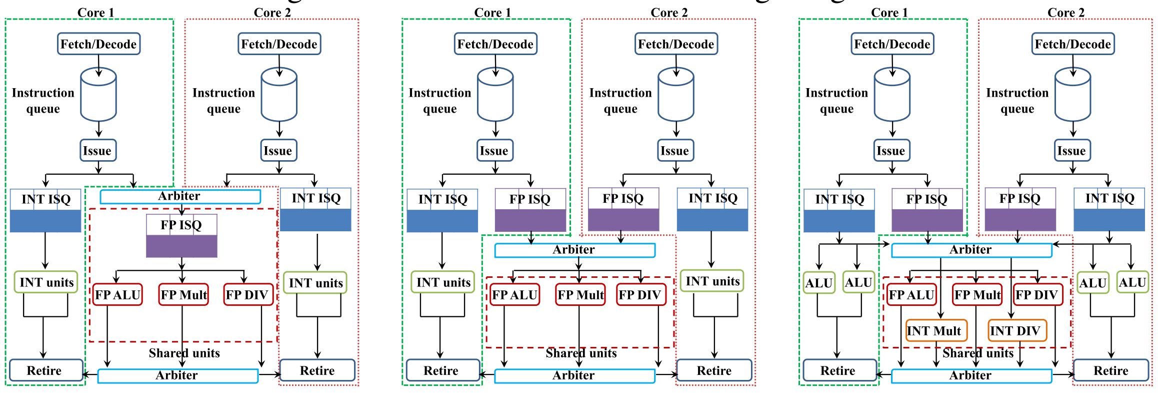


Source: (BORODIN; SIAUW; COTOFANA, 2011).

3D stacked processor designs, as depicted in Figure 2.13. Instructions waiting in the Reservation Station (RS) can be issued into the local (current layer) or remote (distant layers) FUs. The FU processes the instruction, and the Result Distribution (RD) unit returns the result to the issuing processor. The authors consider two use cases in their evaluations. The first is to improve the system reliability by allowing the same instruction to execute locally and remotely and then comparing the results. The second case is to increase the overall system's performance by allowing the cores to access a bigger pool of FUs. Similarly, (HOMAYOUN et al., 2012) investigates the possibility of sharing poolable resources (such as instruction queues, reorder buffer, load-store queue) among cores in 3D stacked chips.

In (RODRIGUES; KOREN; KUNDU, 2015), the authors investigate three possibilities of FU sharing, as depicted in Figure 2.14. The first possibility is similar to the AMD's Bulldozer (BUTLER et al., 2011) implementation since it shares the FP units and the FP instruction queue between two cores. The second design differs by creating private instruction queues for both cores. The third design extends the second by sharing com-

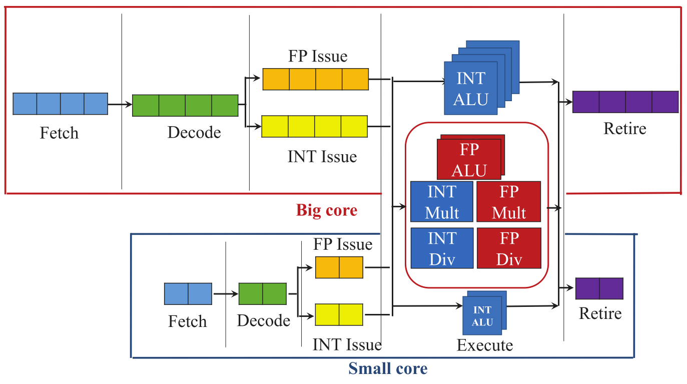
Figure 2.14: Evaluated resource sharing designs.



(a) Issue and FP FUs sharing (b) FP FUs sharing (c) FP and mul/div sharing

Source: (RODRIGUES; KOREN; KUNDU, 2015).

Figure 2.15: Pipeline view of the resource sharing. The pool of shared FUs contains complex operations such as integer multiplication, division, and FP.

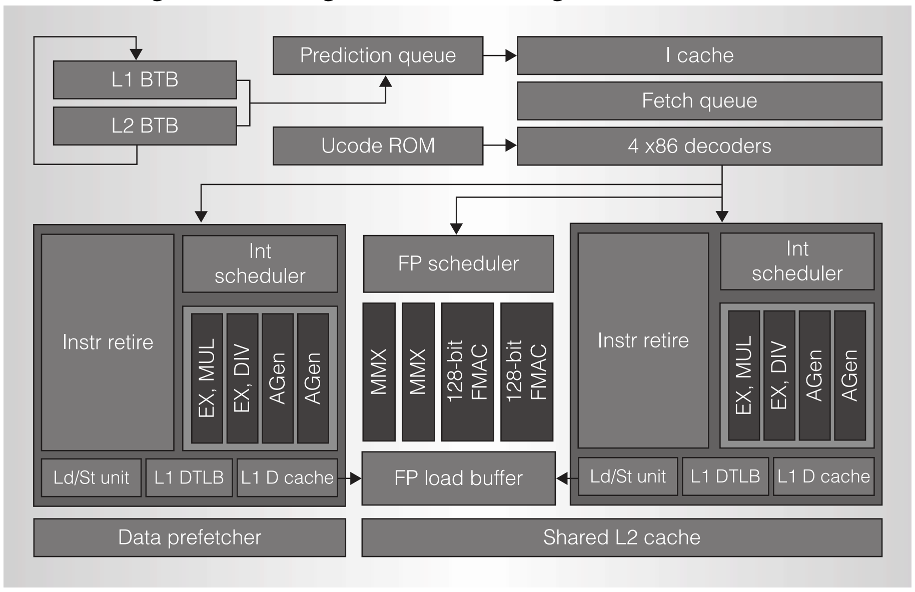


Source: (RODRIGUES et al., 2014).

plex integer operations such as multiply and divide. The work evaluates the designs in different scenarios. The authors found that resource contention and access latency to the shared resources significantly impact performance. While some applications have negligible performance loss due to FU sharing, others are significantly impacted since they use the shared FUs intensively. To deal with the last case, the authors propose to use a dynamic frequency boosting on the shared units, increasing their performance.

Besides FU sharing in homogeneous designs, Rodrigues et al. (RODRIGUES et al., 2014) investigate FU sharing in AMC processors. The idea is to share a pool of complex FUs, such as integer division and FP, in a cluster containing a big (OoO) and a little (InO) processors, as depicted in Figure 2.15. Therefore, the little core can improve its performance since now it has access to more FUs. They claim it is possible to maintain the area and power similar to the traditional cluster but increase the performance of the smaller core. Their results show that it is possible to increase system performance by 20% and improve performance/Watt by 12%.

Figure 2.16: High-level block diagram of Bulldozer.



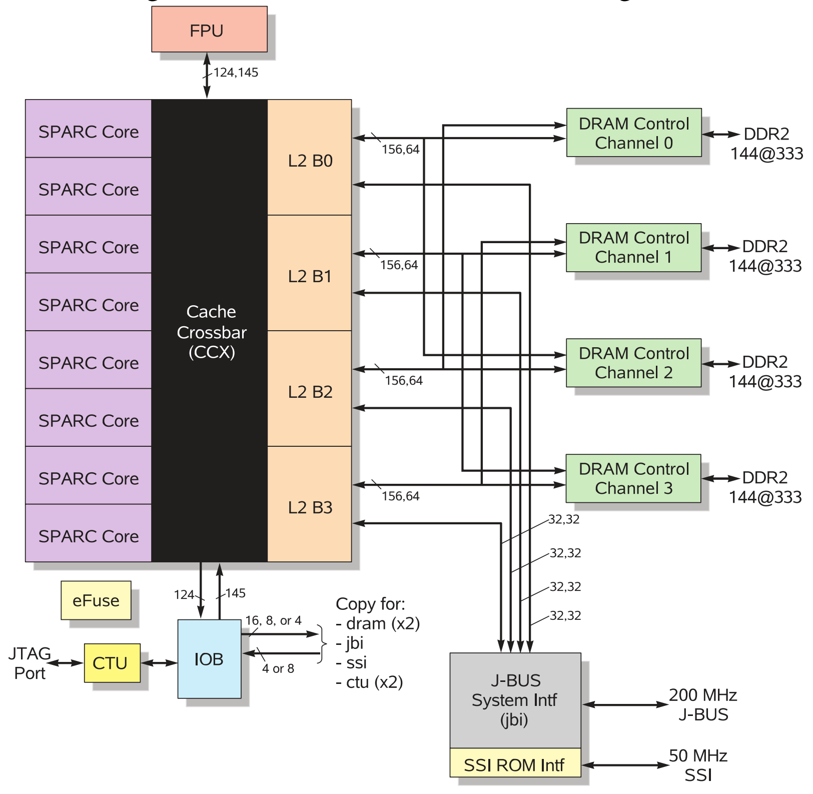
Source: (BUTLER et al., 2011).

Commercial designs

AMD's Bulldozer (BUTLER et al., 2011) is a commercial design that employs resource sharing. Each Bulldozer module combines two independent cores that share datapath resources. Figure 2.16 presents a high-level block diagram of the Bulldozer module. The front-end is shared by both cores (top half of the Figure 2.16), i.e., the branch predictor, represented by the branch target buffer (BTB), prediction queue, the instruction cache, the fetch queue, the decoders, and the microcode ROM. The front-end is shared vertically by both threads, i.e., only one thread can fetch at a time. The decoded integer instructions go into an integer core, which is full OoO. Thus, each integer core has its own scheduler and reorder buffer to retire instructions independently. SIMD/Floating-point instructions go into the shared FP pipeline, which has its own OoO engine. Besides the 64KB shared instruction cache, each core has its own private 16KB data cache but shares a common L2 cache.

In Sun's UltraSPARC T1 (Sun Microsystems, 2008), eight cores share a single FP unit (Figure 2.17). Since the processor targets server applications, each core supports 4 threads in a vertical multithreading fashion. When a FP instruction is found, the floating-point frontend unit decodes the instruction and reads the floating-point register file. Simple FP instructions (e.g., move, absolute value, and negate) are handled inside each core. On the other hand, complex FP instructions are dispatched to the shared FP unit through the crossbar and its operands. When a long latency instruction is executed,

Figure 2.17: UltraSPARC T1 block diagram.



Source: (Sun Microsystems, 2008).

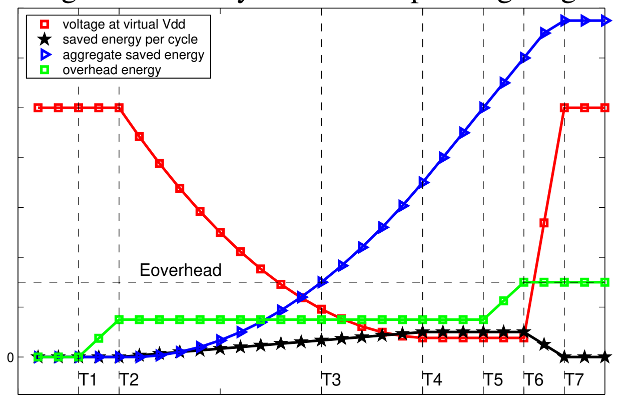
such as a FP, it immediately switches the running thread to avoid datapath stalling. Once the FPU finishes the computation, the result is returned to the original core to write back the instruction, and the halted thread restarts.

2.3 Power Gating

The advancements in technology nodes have shrunk the transistor size, making leakage power a concern in modern designs. Power gating (POWELL et al., 2000) addresses this problem by turning off the circuit's supply voltage to save energy. However, power gating has an intrinsic performance and energy overhead since the circuit is unused when power gated, and some energy is spent during the turning on and off process. Thus, a circuit must be power gated for an interval at least long enough to compensate for the technique's overhead.

Figure 2.18 depicts the key intervals in the power gating cycle. At T_1, the control circuit starts the power gating process, and the circuit power-gated circuit stops working. Between T_1 and T_2, the sleep signal is re-buffered, incurring energy overhead. After the sleep signal is delivered (T_2), the voltage at virtual Vdd starts going down, and the saved

Figure 2.18: Key intervals in power gating.



Source: (HU et al., 2004).

energy starts to increment. At T_4, the virtual Vdd is fully discharged, and the energy-saving per cycle achieves its maximal value. At T_5, the power-gated circuit needs to be used again, causing the sleep signal to be de-asserted and increasing the energy overhead. Starting at T_6, the virtual Vdd is charged up to the Vdd level. The circuit becomes fully operational at T_7. The break-even point occurs at T_3 since the aggregated saved energy is equal to the energy overhead of power gating.

In (HU et al., 2004), the authors investigate power gating capabilities on a FP FU. The goal is to turn off the FU at unused intervals to improve energy consumption. The work also proposes predictors to know when to turn off the FU according to the application phase. In (KUMAR et al., 2014), the authors propose a Hardware/Software codesigned environment to improve the time a SIMD FU can remain off. The authors observe that a FU can have small busy intervals during a long idle interval. If the FU is power gated, it will wake up for a small period of time and then be power gated again. The work tackles this problem by devectorizing SIMD instructions at runtime so that the FU does not need to be awakened. However, the devectorization must be performed with caution since a too aggressive strategy may worsen energy consumption.

The same problem is tackled in (TARAM; VENKAT; TULLSEN, 2018), but using a different approach. In this work, the authors propose a *context-sensitive decoding* (CSD) considering x86 processors. Nowadays, CISC CPUs have front-end and back-end operations, with the former represented by the instructions defined in the ISA and the second represented by *micro-ops*, i.e., smaller instructions that exist only in the microarchitecture. In general, complex CISC instructions are decomposed into several micro-ops at the decode stage of the pipeline. The work proposes a mechanism to change the generated micro-ops depending on the current CPU context, which can attend to different necessities. When considering the power gating of vector units, the authors propose to devectorize vector instructions at runtime to extend the time the vector units can be power gated. Thus, when a vector instruction is found, the decoding generates scalar micro-ops instead of using the power-gated vector FU.

2.4 Contributions of this Thesis

The present work has the following contributions:

- **Decoupled offloader:** An offloading mechanism to offload NEON instructions from the big core to execute on the energy-efficient NEON execution unit present in the little core. The offloader allows the big core to only schedule its NEON instructions to the little core without stalling, i.e., waiting for the instruction to complete its execution.
- **Partial cores with full ISA:** We create partial cores with full ISA by removing the NEON units of the big core and offloading all NEON instructions to a little core using the decoupled offloader.

Contributions on Asymmetric Multi-core with single ISA. Both our proposals keep the main advantage of AMC since all processors can execute the same binaries without requiring migration. Our decoupled offloader leverages the inherent microarchitecture asymmetry to allow a more energy-efficient execution of NEON instructions by the big core, and the partial cores reduce the implementation costs of the entire AMC.

Contributions on ISA extensions. Our decoupled offloader and partial cores with full ISA solutions help to reduce the hardware costs of supporting the entire ISA. The decoupled offloader allows infrequently used extensions to be power gated with minor performance impacts. The partial cores rely on the decoupled offloader to completely remove such extensions and open space for more useful hardware while keeping the full ISA support. This removes the need to move the application to a full core or emulate instructions, as performed in previous works (LEE et al., 2017; BECKER; SOUZA; BECK, 2020).

Contributions on power gating. A common power gating on FUs problem is how to increase its duration. Previous works attempt to predict the application phase (HU et al., 2004) and devectorize SIMD instructions at runtime (KUMAR et al., 2014; TARAM; VENKAT; TULLSEN, 2018). The former strategy must wake up the FU as soon as an instruction that requires it is found. The latter approach overcomes this problem with

SIMD instructions by decomposing them into scalar operations and executing them on scalar FUs. When considering an AMC system, the decoupled offloader can increase the power gating duration of a FP/SIMD unit of the big core by redirecting all these instructions to a more energy-efficient execution unit in a little core. Thus, it allows to apply power gating in intervals that were previously unable, i.e., intervals smaller than the necessary to achieve the energy break-even point. Moreover, our approach has the advantage of working not only on SIMD instructions but also on FP operations, which cannot be easily executed with unspecialized hardware.

3 PROPOSED WORK

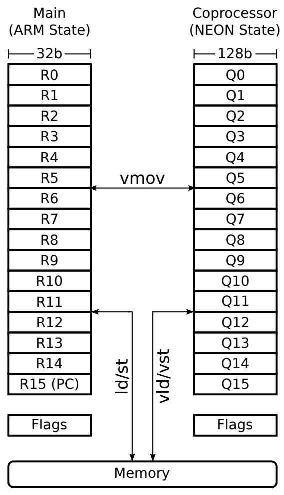
In this chapter, we discuss the decoupled offloader and the partial core with full ISA. We show how to implement both techniques, analyze their benefits and drawbacks, and propose solutions to problems that arise when using them.

3.1 Decoupled Offloader

ISAs are the main interface between hardware and software, defining how a programmer can use a processor. In our work, we evaluate our idea considering the ARM-v7 ISA (ARM, 2014), which is present in ARM big.LITTLE. The ISA defines 16 32-bit General Purpose Registers (GPR) and flags, as shown in Figure 3.1. This set of registers is named as *ARM state*. The ISA also features the NEON extension to allow FP/SIMD computing. The extension defines 16 128-bit registers and flags and forms the *NEON state*. The ARM state is the main state of the processor since it is mandatory and dictates the program flow. On the other hand, the NEON state is optional and is seen as a coprocessor by the main state. Figure 3.1 presents the register file view of both states and instructions to move data on them. The instructions *load* and *store* are used to move data between the ARM state and the memory. Since all NEON mnemonics are prefixed with the letter "v", the instructions *vld* and *vst* are similar to *ld* and *st* but affecting the NEON state. The *vmov* instruction is used to move data between both register files.

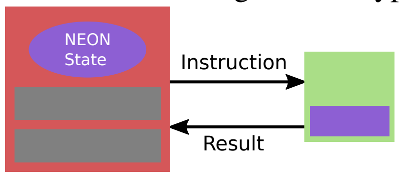
Processors usually work as follows: The processor fetches the instruction's operands

Figure 3.1: Overview of ARM and NEON states.

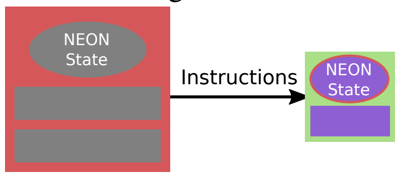


Source: The author.

Figure 3.2: Types of instructions offloading.



(a) Coupled offloader



(b) Decoupled offloading

Source: The author.

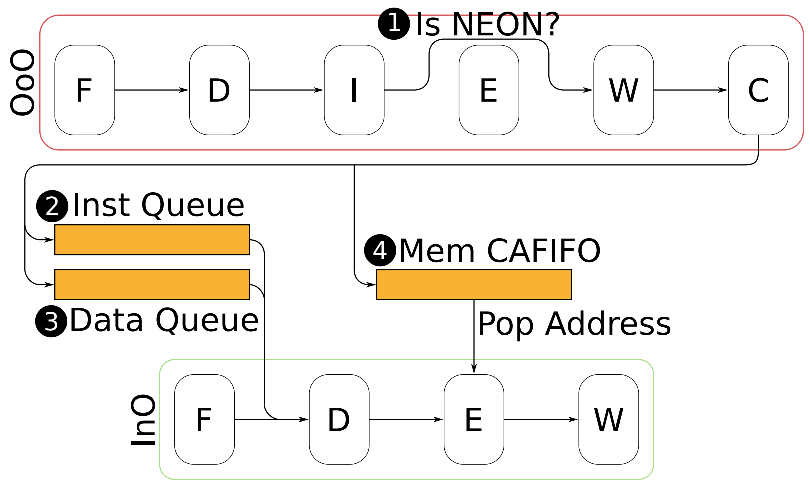
from the register file, performs some computation on them, and then save it back into the register file. The computation happens in the execute stage of the processor inside a FU. Some FUs are very costly in terms of hardware, as is the case of FP and SIMD. A common approach to alleviate these costs is to share expensive FUs between processors, as discussed in Section 2.2. Figure 3.2a presents a coupled FU sharing strategy in an ARM's big.LITTLE inspired design. This strategy allows the big core to turn off its NEON FUs and offload its NEON instructions to execute within the little core. However, this approach may hinder the big core since it has to wait for the offloaded instruction to return to save the result into its NEON state. A better solution is to allow the big core only to offload its NEON instructions and not wait for the results. To accomplish this task, it is necessary to eliminate the bottleneck of saving the result of processed instructions back into the NEON state present in the big core. This can be achieved by moving the NEON state from the big core to the little core, and thus, closer to where the instructions are executed, as shown in Figure 3.2b. We name this type of offloading as *decoupled offloading* since the big core can send instructions without waiting for the results to return.

However, the following problems must be addressed to enable the decoupled offloading:

1. Which stage of an OoO pipeline is the best candidate to offload instructions?
2. How to offload instructions without stalling the big core?
3. How to guarantee coherence and correct program execution?

Figure 3.3 presents the necessary hardware and modifications to implement the decoupled offloader considering big (OoO) and little cores (InO). An InO core issues, execute, and complete instructions in-order. The limitation of this design is that it cannot issue ahead instructions even if they are ready to execute. An OoO design overcome this problem by issuing future instructions speculatively and committing them InO to keep correct program execution. An OoO pipeline usually presents the following stages:

Figure 3.3: Big and little cores with the necessary modifications to support the decoupled offloader.



Source: The author.

- Fetch: Fetches instructions from memory and inserts them into the pipeline.
- Decode: Decodes instructions and retrieves their operands.
- Issue: Decoded instructions are inserted into the issue queue. These special hardware structures can pick instructions out of order and insert them into FUs to execute.
- Execute: Dispose of multiple FUs to perform instruction computation.
- Writeback: Once in this stage, the results produced by the execute stage are now free to be used by other instructions waiting at the issue stage.
- Commit: This stage performs in-order instruction retirement, i.e., after this point, the instruction modifies the processor state permanently.

To solve the first problem, we need to choose the best pipeline stage of an OoO core to offload NEON instructions to the little core. The Fetch stage cannot be used as offloading stage since NEON instructions can only be detected after their decoding. After the Decode stage, all instructions fly inside the datapath in an OoO fashion until they hit the Commit stage, which re-orders the instructions before retiring them and modifying the processor state permanently. Issue, Execute, and Writeback stages present the same problems because all instructions inside them are being executed speculatively, i.e., it is impossible to know whether they complete successfully until the commit stage. If we offload instructions to execute on the little core starting from one of these stages, the little core would have to send control signals to the big core to inform each NEON instruction completed, which creates an undesired coupling between the processors. Thus, the big core would have to wait for the offloaded instructions results.

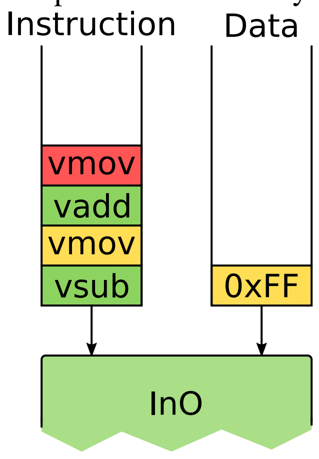
The only stage left is the Commit stage. Committed instructions have two guarantees. First, all older instructions were successfully completed. Second, committed instructions were executed successfully. Thus, we propose to offload instructions from the big core to execute inside little's InO datapath only after they complete the Commit stage. Since offloaded instructions leave the Commit stage in order and are also executed in order inside the little core, there are no ordering execution problems in the offloaded instruction stream. As shown in Figure 3.3, we bypass ❶ the execution of all NEON instructions and offload them after the commit stage. From the OoO core point of view, all offloaded instructions are successfully completed, and changed the architectural state. We name this as *fake commit* because the big core considers the instruction as committed even though it has not been executed yet. For the sake of the explanation, let us ignore interrupts and exceptions. We further explain this topic in Section 3.1.

The next problem is to offload instructions without stalling the big core. The key idea to enable the decoupled offloader is to create a copy of the big's coprocessor state inside the little core and allow the big core to bypass the execution of NEON instructions and commit them despite that. Thus, the instructions end their passage through the big core without changing the architectural state. Once the instructions are fake committed, they are offloaded to the little core. Since the big's coprocessor state is present inside the little core, it can execute the received offloaded instructions in-order and save the results back into the big's coprocessor state inside little. This eliminates the necessity of returning any produced result to the big core, allowing it only to schedule instructions to execute in little.

It is important to highlight that the offloader mechanism must support all NEON instructions, which can be divided into three groups (ARM, 2014): Data processing, data movement, and memory access. The data processing group represents instructions that perform computation using only values present in the coprocessor state, e.g., add two numbers. Data movement comprises instructions that move data between states, i.e., from main to coprocessor and vice versa. The last group, memory access, constitutes load and store instructions.

The simplest group to support is data processing due to its one-way communication. Simply offloading the instructions to execute inside little's datapath is enough since the data used in the computation is already present in the big's NEON state inside little. For that, we add an instruction queue ❷ to hold temporary instructions waiting for execution. This allows the big core to only send instructions to little and continue its

Figure 3.4: Example of instruction synchronization.



Source: The author.

computation. Whenever possible, the little core pops an entry from the instruction queue, and the instruction travels little's pipeline as if it were one of its instructions. The only difference is that offloaded instructions access the big's NEON state present in little.

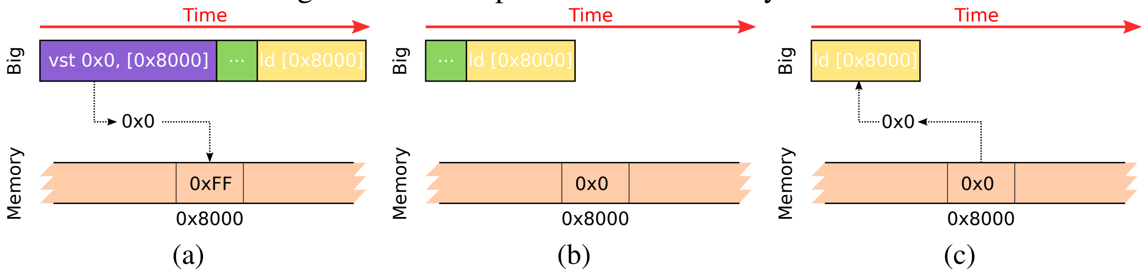
To support data movement (e.g., *vmov*[1]), it is necessary to handle communication both ways. Considering main to coprocessor communication instructions, we add a data queue ❸ to hold data consumed by offloaded instructions. This way, it is possible for the big core to offload data movement instructions without stalling until the instruction is executed. To support NEON to ARM state communication, we add a link at the end of little's datapath to the big core, allowing it to receive data from its NEON state in little.

Figure 3.4 depicts both examples of data movement instructions. The Figure presents the instruction and data queues connected to the InO datapath. The instruction queue contains 4 instructions: *vsub*, *vmov* (main to coprocessor), *vadd*, and *vmov* (coprocessor to main). The data queue contains only the value of the first *vmov* instruction. The last offloaded instruction is a NEON to ARM state data movement. This instruction requires datapath **synchronization**, i.e., the big core must wait until all offloaded instructions are complete before offloading new instructions. This occurs because the value transferred by the last *vmov* is necessary for the big core to resume processing.

The last group to support is memory access (e.g. *vld/vst*). Although there are explicit move instructions to transfer data between states, it is also possible to move data between them through memory. Thus, the offloader must track when this happens and continuously check for memory consistency problems, which usually arise when we have an ARM state memory instruction after a NEON memory instruction to the same address.

[1]The *vmov* instruction allows data movement not only between ARM and NEON states but also inside the NEON state. However, the last case is covered as a data processing instruction since it relies only on values already present in the NEON RF.

Figure 3.5: Example of correct memory access.



Source: The author.

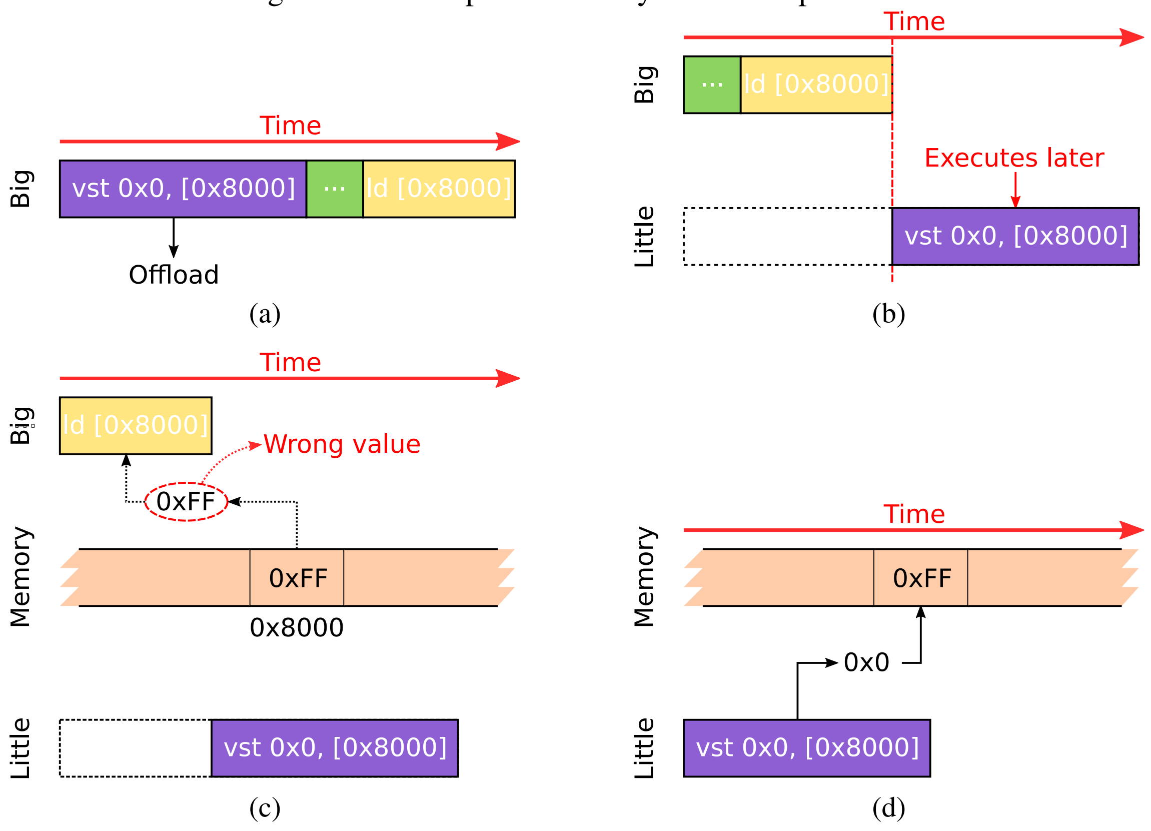
The problem is that the offloaded instruction might not have been executed when the main state memory instruction executes, creating a memory ordering problem.

As an example, let us suppose that the big core executes a simple sequence of instructions containing *vst* (NEON write) followed by a *ld* (ARM read) instruction, as depicted in Figure 3.5. Firstly (Figure 3.5a), the *vst* instruction writes the value 0x0 to the address 0x8000, overwriting the old value 0xFF. After some instructions (Figure 3.5b), the *ld* instruction loads the 0x0 value into the ARM state (Figure 3.5c).

However, the correct program order might not be preserved due the offloader, as exemplified in Figure 3.6. As shown in Figure 3.6a, the *vst* instruction is offloaded, which means that its execution will happen later (Figure 3.6b. In the example, the *vst* instruction is executed by the little core after the *ld* instruction. When the big core executes the *ld* instruction, it reads the wrong 0xFF value (Figure 3.6c) from memory since the *vst* instruction only updates it later (Figure 3.6d). The case described is named as Main Read after Coprocessor Write (MRaCW). Coherence problems also arises in Main Write after Coprocessor Write (MWaCW) and Main Write after Coprocessor Read (MWaCR).

We use a content-addressable FIFO (CAFIFO) ❹ to hold the addresses of outstanding offloaded memory instructions. Every time a coprocessor memory instruction is offloaded, the target memory address is pushed into the CAFIFO. The address remains in the CAFIFO until the offloaded instruction completes its passage through the little core. If the main state needs to perform a read or write, it will check if the target address is present at the CAFIFO. If there is a memory conflict, then the big core waits until all conflicting offloaded instructions finish (i.e., a synchronization) to guarantee coherence and correct program execution.

Figure 3.6: Example of memory coherence problem.



Source: The author.

Interrupts and exceptions

Interrupts and exceptions are unexpected events that change the normal execution flow of a processor. Interrupts are asynchronous events that come from outside the processor, whereas exceptions are synchronous and result from execution errors in applications or environment (OS) calls. Both types of events change the execution flow of the processor, i.e., the CPU jumps to a handler routine to attend the event. A common example of an interrupt is a keyboard input, which causes the processor to read the key pressed. On the other hand, trying to execute an illegal instruction is a common exception found in many ISAs.

Since interrupts are dissociated from current program execution, they can happen anytime. When interrupted, an OoO pipeline flushes all in-flight instructions, saves the address of the last instruction committed, and jumps to the interrupt handler. In our decoupled offloader model, the OoO core fake commits NEON instructions and trusts that the InO core will execute them successfully. This approach has no problem with interrupts since it is similar to a simple mispredicted branch, which also causes datapath flushes.

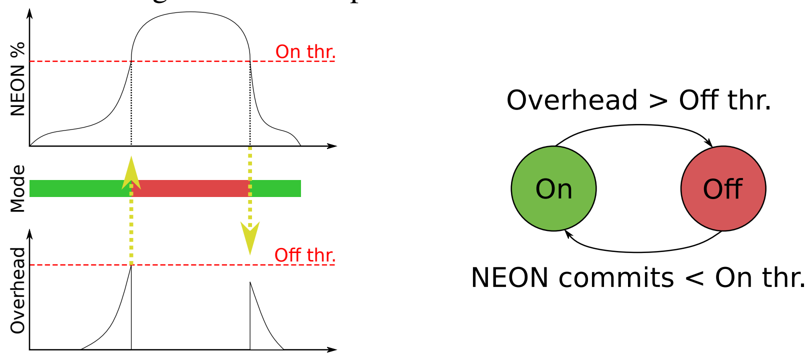
However, the decoupled offloader model can not deal with offloaded coprocessor instructions that raise exceptions. This occurs because the OoO core trusts that all offloaded instructions are successfully executed, which will not be the case if an exception happens. NEON provides instructions for integer SIMD, FP, and FP-SIMD. Fortunately, only FP/FP-SIMD raise exceptions, but the NEON ISA allows the programmer to choose whether data processing instructions should raise them. This behavior is controlled by the Floating-Point Status Control Register (FPSCR). This special register provides FP flags and allows fine control over the FP unit. Moreover, it enables/disables FP exceptions. A programmer can disable FP exceptions to avoid unexpected changes in control flow and query the FPSCR register to check whether an exception behavior occurred. In other words, the NEON ISA allows the programmer to check if an exception behavior happened without triggering an exception. We assume that applications query the FPSCR register when they need to check for exceptional behavior.

Furthermore, memory instructions also raise exceptions. This occurs when it is not possible to translate the accessed virtual address, i.e., it is not present in the TLB, or when the address accessed is protected. In both cases, the execution is transferred to the OS, which handles the problem. The decoupled offloader can support both situations considering NEON's memory access instructions because the address is generated by the OoO datapath and offloaded to the InO core. Thus, the big core can detect exceptions before offloading instructions, ensuring the support for this type of instruction. Moreover, the big core can offload the physical address accessed by memory instructions. The little core performs the memory access using the physical address, and the coherence between both cores after the instruction execution can be preserved by existing coherence mechanisms, such as the CCI or the common LLC, as discussed in Section 2.1.

3.2 Arbiter

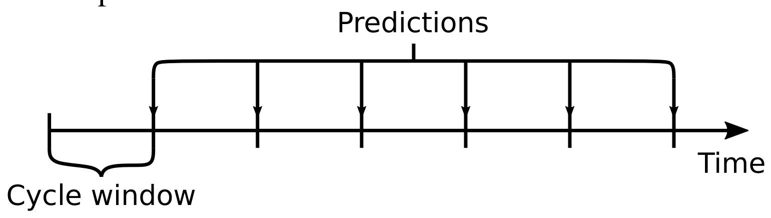
The offloader is designed to explore low NEON usage phases of applications with minor impacts on performance. However, in high usage phases, the offloader fails to provide benefits since the big core offloads instructions to the low performance NEON FU inside the little core. Thus, it is important to detect at runtime when the offloader is providing gains. We propose a simple 2-state hardware arbiter that can be attached to the OoO datapath to achieve this. The states are based on the current offloader status. When the offloader is active (ON), the arbiter monitors the cycles overhead of the offloader, i.e.,

Figure 3.7: Example of the arbiter in action.



Source: The author.

Figure 3.8: Arbiter predictions happen only at the end of a cycle window interval to facilitate hardware implementation.



Source: The author.

how much offloading harms big's performance. The performance harm is measured by a counter that increments every time the big core cannot complete an instruction due to the offloader. For example, if it is waiting for synchronization to complete or cannot offload an instruction due to a full queue. Whenever the overhead is above a given threshold, then the current program phase is better suited to run on the big core. Thus, the arbiter turns off the offloader. Once in the OFF state, the arbiter compares the rate of NEON instructions to a turn-on threshold. If the rate of committed NEON instructions is below the turn-on threshold, the offloader is activated.

Figure 3.7 shows an example of the arbiter in action. Starting at the ON state, the arbiter monitors the overhead and triggers a mode change once it is above the defined threshold. When at the off state, it monitors the rate of committed NEON instructions and switches back to the ON state when the rate drops below the off threshold. To facilitate hardware implementation, the arbiter makes a prediction only after a period of cycles. We name this interval as *cycle window*. Thus, at the end of a cycle window, the arbiter evaluates the current application behavior and predicts that it will repeat in the next window, as illustrated in Figure 3.8. Thus, the arbiter can be implemented with a counter that ticks every cycle. When the counter reaches the value of the cycle window, the arbiter makes a prediction and resets the counter.

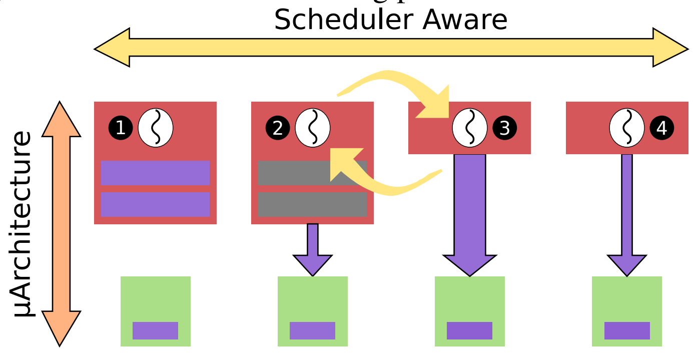
It is important to notice that high percentages of NEON instructions do not always lead to high overheads. For example, if the total number of instructions committed in an interval is low (e.g., due to frequent cache misses), the NEON instructions can have a representative rate in the interval. On the other hand, low NEON rates can result in high overheads if, for example, the instructions executed incur frequent datapath synchronization.

We allow the system designer to tune the threshold parameters for each application, but it can be challenging to find the best parameters for some programs. Even when configured properly, applications may have problematic phases that make the *basic* arbiter described so far trigger excessively mode changes, harming execution time and energy efficiency. We handle these cases by proposing another arbiter that extends the previous arbiter's behavior described with a performance-oriented approach. The performance-oriented arbiter counts the number of mode changes over the last predictions. If the arbiter made more state switches than a given threshold in the last predictions, then the arbiter is turned off for a long time. The reasoning is to favor performance by bypassing problematic program phases using the NEON FUs in big. We refer henceforth to the basic arbiter as *Basic* and the performance-oriented arbiter as *Performance*. We evaluate both arbiters in Section 5.1.

3.3 Partial Core with full ISA

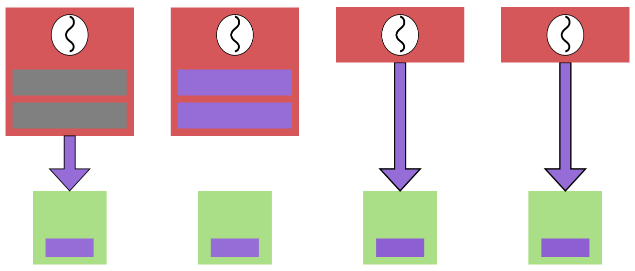
The decoupled offloader allows to completely remove all NEON hardware from a big core to form a partial core while still maintaining full-ISA support. The partial cores are equivalent to big cores in integer processing but have limited NEON capabilities, bringing a new level of heterogeneity to the system. Figure 3.9 shows a multi-core system containing partial cores with the same ISA in action. The first two cores are full big processors equipped with the offloader to save power. The offloader is managed by the microarchitecture using the proposed arbiter at runtime without the intervention of the programmer or the OS. A scheduler, aware of the big cores heterogeneity, handles thread mapping on big and partial cores taking NEON usage into account. In the example depicted, the core ❶ uses its internal NEON units since it is in an intensive NEON phase, making the arbiter turn off the offloader. On the other hand, core ❷ is in a low utilization phase, which allows the arbiter to turn off its NEON units and use the offloader to save power. Core ❸ heavily uses the offloader, whereas core ❹ has a well-suited application for partial cores due to the low NEON usage.

Figure 3.9: Multi-core containing partial cores with full ISA.



Source: The author.

Figure 3.10: The scheduler swaps applications at cores 2 and 3 to improve overall's system performance.



Source: The author.

The scheduler decides to swap the threads running on cores ❷ and ❸, so each one runs on the most appropriate processor. In Figure 3.10, the arbiter detects the change on the application behavior of the first core, which is not using as much NEON as previously, and then power gate the core's NEON units and activates the offloader.

OS support for partial cores

In our model of partial cores, every core supports the full ISA. Thus, the OS does not need to be adapted to support heterogeneous ISAs. However, it is necessary to enhance current OS schedulers to take advantage of the heterogeneity of big cores. We propose a heuristic to help schedulers decide which type of big core (full or partial) is better suited for a given thread. The heuristic works similarly to the arbiter discussed in Section 3.2. We count the rate of committed NEON instructions over a time window of OS interventions. During a scheduler intervention of a partial core, the scheduler compares the rate of NEON instructions for each thread running in a full core. Threads are swapped if the partial core thread has a NEON rate bigger than a thread running on a full core.

However, this over-simplistic approach might lead to unnecessary thread swaps, which may occur when two threads have close NEON rate values. Nevertheless, it is necessary to swap threads only if the thread running on a partial core has lower performance due to the offloader. Therefore, a partial core thread is moved to a full core only if it has a bigger NEON rate and lower performance due to the offloader. This way, it is possible to provide a fair utilization time of the full cores by all threads mapped to big cores. This scheduling heuristic can be embedded in current AMC schedulers since it does not decide whether an application should run on a big or a little core but only picks the best big core type for a given task.

4 SIMULATION TOOLCHAIN

In this chapter, we describe the simulation environment used to evaluate our proposal, the in-house simulator developed to simulate the offloader, the energy modeling, the simulation parameters, and the benchmarks used.

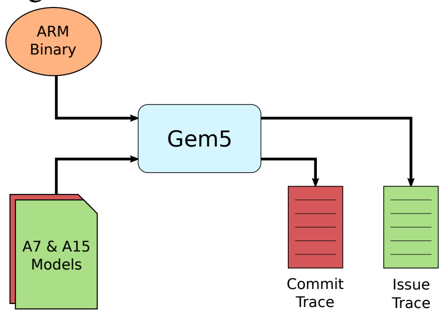
4.1 Gem5

We use gem5 (BINKERT et al., 2011) to produce the execution trace of simulated programs. As depicted in Figure 4.1, we feed gem5 with the CPU models of ARM A7 and A15 processors and the program binary. We modify the simulator to produce an execution trace for each core model. For the big core (A15), we produce the **commit trace** containing information about the instructions committed in the simulation. We produce the **issue trace** for the little core (A7), which contains information about all instructions issued into little's FUs. The commit trace contains the time an instruction is committed, its type (data processing, data movement, or memory access), and the memory address accessed if it is a memory instruction. The issue trace contains the time the issue happened, the instruction type, and the FU used by the instruction. We detail both types of traces and the gem5 modifications in Appendix B.

4.2 In-house simulator

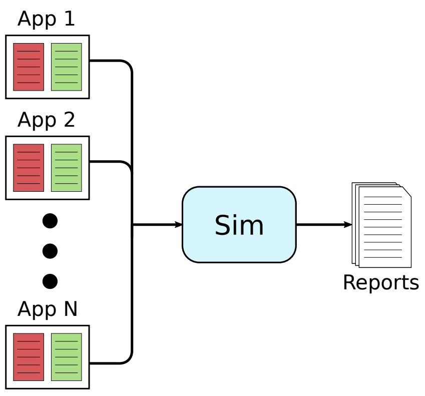
We implement a simulator written in Python to evaluate our idea. Figure 4.2 shows a high-level view of the inputs and outputs of the simulator. The simulator receives one or more applications, represented by the commit and issue traces produced by gem5. Each

Figure 4.1: Gem5 simulation flow.



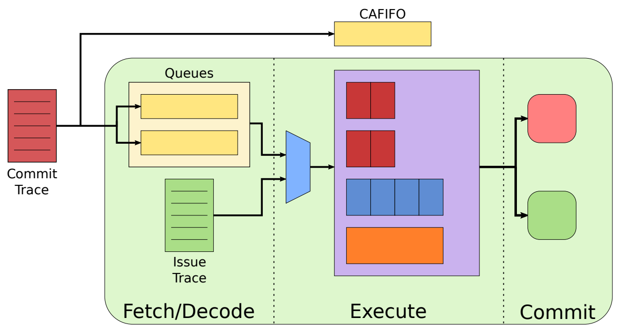
Source: The author.

Figure 4.2: Execution flow with the in-house simulator.



Source: The author.

Figure 4.3: Offloader simulation. We model little datapath with the offloader structures in our simulator.

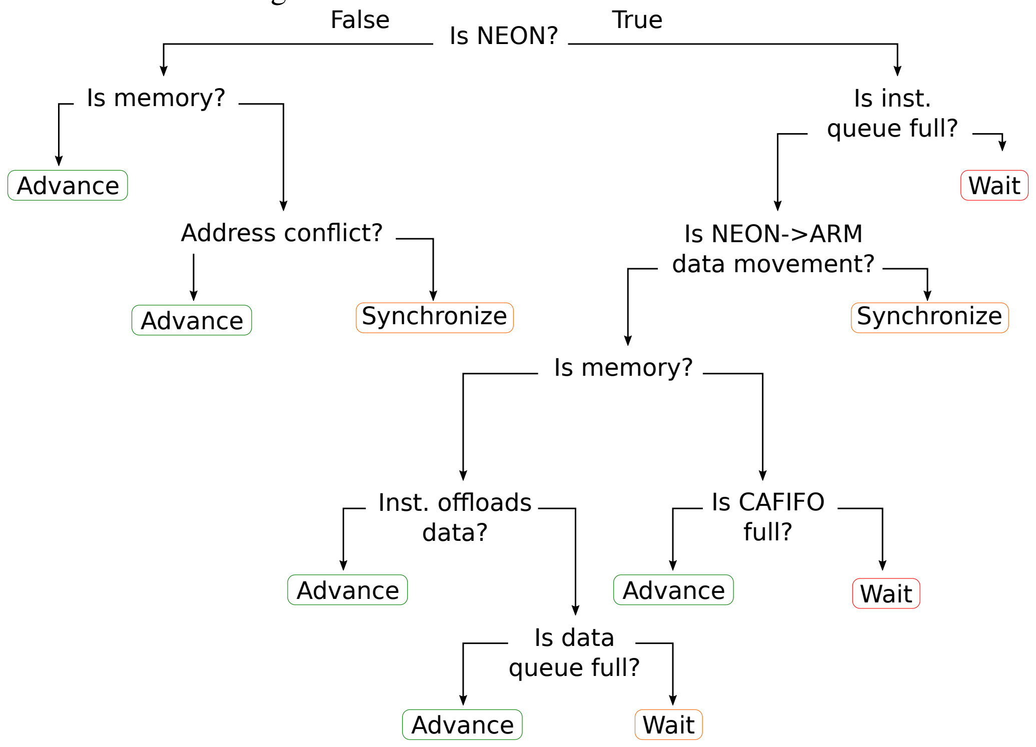


Source: The author.

pair of traces forms an application. The goal of the simulator is to assess the impact of the offloader on the system. At the end of the simulation, the program produces reports for each application execution showing the offloader impact.

The core of the simulator is the datapath model, as depicted in Figure 4.3. We model A7's datapath with the necessary hardware structures to support the offloader. With both traces (commit and issue) and the A7 datapath model, it is possible to reconstruct the gem5's simulation. By modeling the offloader structures, it is possible to measure how the offloader affects program execution. The simulator has three main components: the commit trace, the issue trace, and the datapath, which resembles the A7 core model available in gem5. At the beginning of the simulation, we place a trace pointer at the start of the commit trace. The pointer represents the big core state at the current simulation time and points to the next instruction to be committed. The simulation loop occurs as follows: at every cycle, it checks if there is an instruction commit at that time present in the commit trace. If it is a NEON instruction, it is inserted into the datapath and goes directly to the Queues stage, located at the Fetch/Decode stage of the little core. We model the Execute stage with different execution units for each instruction type, which requires

Figure 4.4: Decision tree for commit trace.



Source: The author.

different latencies to complete its passage through the stage. We also model pipelined and non-pipelined FUs. A pipelined FU can receive one instruction every cycle, whereas a non-pipelined unit handles just one instruction at once. After the Execute stage, the instruction goes to the Commit stage and is delayed for one cycle before being removed from the datapath.

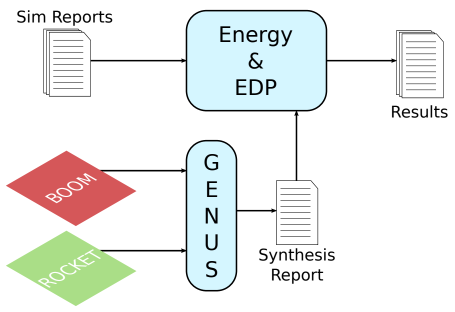
Ideally, the overhead of the offloader is as close as possible to zero. For each simulation, we keep an overhead counter that indicates the overhead of the decoupled offloader. The simulation starts with the counter at 0 and increments every time the big core cannot complete instruction in its commit trace due to the offloader, e.g., a full FIFO or synchronization. Figure 4.4 shows the decision tree the simulator does for every entry in the commit trace with "False" and "True" represented by left and right paths, respectively. The tree has three types of endpoints: Advance, Wait, and Synchronize. *Advance* means that the instruction can be successfully committed at this cycle, and the trace pointer advances one entry, whereas *Wait* increments the overhead counter, and the simulator tries to commit the current instruction again in the next cycle. *Synchronize* is a mix of *Advance* and *Wait* since it waits (incrementing the overhead) until the synchronization condition is satisfied to advance the trace pointer.

The decision tree can be implemented as *ifs* and *elses* in a loop. When reading an entry from the commit trace, the simulator checks whether it is a NEON instruction. If true, then it needs to offload. Since offloading is only possible when an entry in the instruction queue is available, it sees if there is room in the queue. If there is no room, then it waits (increment overhead) and tries again in the next cycle. However, if the queue is not full, then it performs the following test to know whether the instruction is a NEON to ARM state data movement. If that is the case, then a synchronization must be performed, causing the big core to wait for all offloaded instructions to complete before advancing the commit trace pointer. If false, the simulator queries whether the instruction performs memory access. If true, there must be room in CAFIFO to push the accessed memory address. Otherwise, the big core must wait and try again in the next cycle. If the instruction is not a memory instruction, the simulator checks if the instruction offloads data. If false, then the current instruction is a data processing type. If true, then it is a data movement type that moves data from ARM to NEON state. In this case, the simulator performs the final query to know whether the data queue is full. If false, then the data movement instruction can be offloaded.

Returning to the top of the flowchart, if the current evaluated instruction is not a NEON instruction, then it does not need to be offloaded. However, the offloader must track ARM state memory instructions to ensure memory coherence. The left path of the decision tree first checks whether the instruction is a memory instruction. If it is not, then the simulator can safely advance. If it is, then the simulator checks for address conflicts. In this case, the simulator compares the memory address accessed by the instruction to all memory addresses accessed by offloaded memory instructions. If there is a conflict, then the simulator waits until the little core executes the conflicting instruction.

The issue trace works similarly to the commit trace since it also has a trace pointer to represent the little core state by indicating the next instruction to be issued. At each cycle, the simulator checks for instructions to be issued. If there is any, the simulator inserts the instruction into its corresponding FU. We also model the issue interference between the offloaded stream and little's instruction stream inside the simulated datapath. If the little core issues into a FU, then this FU becomes unavailable to the offloaded instruction at this cycle. If both streams need to issue into the same FU at the same cycle, then the little core has the priority. If the little core needs to issue an instruction into a non-pipelined FU and it is already busy, then it waits and increments the little core overhead counter. This strategy prioritizes the little core application over the offloaded instructions. The reasoning is that it is better that the big core fall back to use its own NEON FUs than

Figure 4.5: We synthesize BOOM and Rocket processors to obtain power values used in our energy and EDP models.



Source: The author.

using the slow units in little at the cost of harming little's performance.

4.3 Energy modeling

The in-house simulator generates reports that are used to model energy and Energy-Delay Product (EDP), as shown in Figure 4.5. However, it is necessary to acquire the power values of both cores to use in the energy/EDP models. McPAT (LI et al., 2009) is a well-known tool to model power and energy of processors in computer architecture research. However, it may lead to inaccurate power models (XI et al., 2015). To overcome this problem, we choose to model our cluster using synthesis data from two RISC-V (WATERMAN et al., 2016) processors: BOOM (ZHAO et al., 2020) (OoO) and Rocket (ASANOVIć et al., 2016) (InO). We synthesize both cores using Cadence Genus to obtain power consumption values to represent the big, and little cores using a 15nm standard cell library (MARTINS et al., 2015). Since BOOM and Rocket are provided as generators, it is possible to change their parameters, e.g., the number o FUs. We tune both cores to be similar to A7 and A15 processors. We model the partial core (a big core without NEON) as a BOOM CPU without the FP support since NEON features FP and SIMD instructions, but RISC-V does not have SIMD instructions.

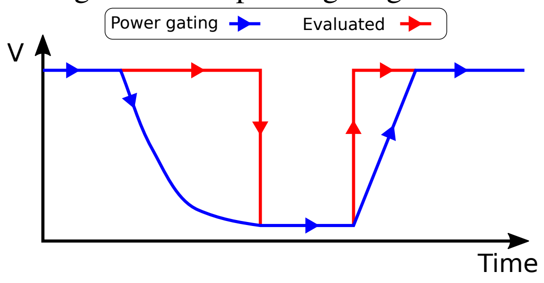
The following Equation defines the energy[1] consumption of the simulated system:

$$E = C_O * P_O + (C_N + C_M) * P_N$$

Where C and P stand for the number of cycles and power consumption, respectively. The M, N, and O represent the states *migrating*, *normal*, and *offloading*, respectively. Since the coprocessor state is migrated to the little core when in offload mode and

[1]The cycle time can be safely omitted since it is fixed for all evaluated designs.

Figure 4.6: Voltage curves in power gating and in our evaluation.



Source: The author.

is brought back to the big core when in normal mode, it is necessary to model the migration cost, which is defined as the number of cycles to move registers' data and power gate the FUs. Although power-gating has a voltage curve until the circuit is completely shut off (Section 2.3), we model the power consumption as a step function along the entire migration process. That is, we consider the system consumes its full power until the migration process is finished, as shown in Figure 4.6. This is a very conservative approach and represents the worst-case scenario for our proposed technique.

The EDP considers the energy consumed to perform a computation and the time used. Therefore, a good EDP result indicates that energy is saved and that the computation does not take too long. EDP is generally defined as $EDP = E * t$. In our case, the EDP[2] model is defined by the equation:

$$EDP = E * (C_M * C_N * C_O)$$

4.4 Simulation Parameters

Offloader (Section 3.1). We set instruction and data FIFOs to have 48 entries and the memory CAFIFO to 32 entries. This size is enough to handle small bursts of NEON instructions, e.g., in a procedure call.

Arbiter (Section 3.2). We set the Basic arbiter parameters as 1000, 20%, and 200 for the cycle window, minimal rate to enter in offload mode, and maximal overhead to deactivate it. These parameters are configurable and can be tuned for specific scenarios to achieve better results. Therefore, a prediction is made every 1000 cycles. If the big core is in normal mode and the current NEON commit rate is below 20%, then it changes to the offload mode. If in offload mode and the overhead is above 200 cycles, the big

[2]The cycle time can be safely omitted since it is fixed for all evaluated designs.

core changes back to normal mode. We chose these values by experimenting with the workloads used in this work. The Performance arbiter extends Basic by monitoring the last 10 predictions. If there are 3 mode changes in the last 10 predictions, then the arbiter turns off the offloader for 100K cycles.

Scheduler (Section 3.3). We set an OS intervention every 160K cycles (CONSTANTINOU et al., 2005). Thus, a thread enters into OS mode, and the OS evaluates whether it is necessary to move a thread from a partial to a full core. Since the time to populate the L1 cache is predominant in thread migrations (LI et al., 2007), we consider an overhead of 12K cycles (SOUZA et al., 2021) in migrations. Thus, we always consider the cache as cold in every migration, which represents the worst case. To assess if the offloader is providing gains, we set the maximal tolerable offloading overhead as 10% of the OS intervention window. Therefore, if 10% of the 160K cycles were wasted waiting for the offloader, we consider it is not providing gains and allow the thread on the partial core to move to a full core.

Power (Section 4.3). We synthesize BOOM and Rocket cores to obtain the power values. We consider a cluster formed by BOOM and Rocket processors. The FPU inside BOOM represents circa 23% of BOOM's power consumption and 20% of the entire cluster. Since we consider the FPU as the NEON unit in an ARM core, we assume that the cluster consumes 80% of its power when BOOM's FPU is inactive. Therefore, the cluster consumes 80% of its power when offloading. This also sets the upper bound of the technique since the maximal energy improvement, 20%, occurs when it is possible to execute an entire application using the offloader. A previous work (ENDO; COUROUSSé; CHARLES, 2015) models the A15 processor on McPAT. Both NEON units in the A15 core represent 31.2% of the energy consumption of the entire execution stage and circa 14.6% when compared to the entire core on average when considering the parsec (BIENIA et al., 2008) benchmarks. Thus, our synthesis data is close to the values available in the literature.

4.5 Benchmarks

To evaluate the proposed work, we use applications from two benchmarks sources: Polybench (POUCHET, 2015), and mibench (GUTHAUS et al., 2001). Polybench consists of 27 math applications, which have high NEON utilization. This allows us to evaluate the offloading under high NEON usage. Mibench is designed to represent common

Table 4.1: Polybench applications, the percentage of NEON instructions in each benchmark, and the percentage of instruction synchronization (IS) in NEON instructions.

Benchmark	NEON (%)	IS in NEON (%)
2mm	31.55	0.03
3mm	28.90	0.02
adi	46.40	0.01
atax	29.02	0.08
bicg	34.98	0.07
cholesky	57.06	0.01
correlation	39.38	0.10
covariance	38.44	0.02
doitgen	52.02	0.03
durbin	16.96	0.30
fdtd-2d	43.40	0.00
gemm	45.82	0.02
gemver	30.14	0.05
gesummv	31.27	0.10
gramschmidt	43.31	0.05
heat-3d	92.60	0.00
jacobi-1d	38.64	0.17
jacobi-2d	48.72	0.00
lu	45.96	0.00
ludcmp	44.09	0.00
mvt	25.25	0.09
seidel-2d	77.05	0.00
symm	43.43	0.03
syr2k	54.95	0.01
syrk	50.21	0.02
trisolv	24.42	0.23
trmm	37.67	0.6

Table 4.2: Mibench applications, the percentage of NEON instructions in each benchmark, and the percentage of instruction synchronization (IS) in NEON instructions.

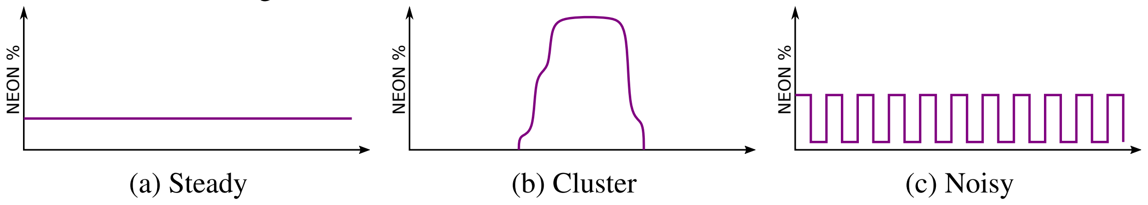
Benchmark	NEON (%)	IS in NEON (%)
aes-d	0.52	0.00
aes-e	0.52	0.00
basicmath	5.63	9.60
bitcount	0.00	15.44
dijkstra	0.00	0.00
fft	5.21	6.77
fft-i	5.31	6.59
jpeg-c	0.16	0.00
jpeg-d	0.48	0.00
patricia	0.55	17.05
qsort	2.01	0.00
stringsearch	0.26	0.00
susan-c	2.84	15.22
susan-e	1.97	16.00
susan-s	0.27	10.75

embedded system applications, being a representative scenario for ARM big.LITTLE. Tables 4.1 and 4.2 shows the percentage of NEON instructions executed in all evaluated polybench and mibench applications, respectively. Polybench apps tend to have high NEON usage, with almost all benchmarks (except *durbin*) containing more than 20% of NEON instructions executed. Some applications are almost entirely executed using NEON instructions, as is the case of *seidel-2d* (77.05%) and *heat-3d* (92.60%). On the other hand, mibench presents low NEON usage. The application that uses NEON the most in mibench is *basicmath* (5.63%), followed by *fft-i* (5.31%) and *fft* (5.21%). Thus, we expect our technique to obtain minor gains in polybench but to provide high gains in mibench. We expect minor performance impacts in both cases, especially in polybench, since the arbiter can bypass application phases that use NEON intensively.

Besides the number of NEON instructions executed, another important metric is the number of synchronizations. As discussed in Section 3.1, frequent datapath synchronizations can negate offloader gains by requiring the big core to stall and wait for the offloaded instructions. Thus, the worst case for the offloader is represented by applications that have high NEON usage and a high number of synchronizations. Tables 4.1 and 4.2 also present the percentage of instruction synchronizations (IS) in NEON instructions in each benchmark. All polybench applications demand few ISs. The benchmark *durbin* has the most representative number of IS compared to the total NEON instructions. This indicates that although polybench applications use NEON intensively, the instructions do not require frequent synchronizations. On the other hand, mibench features low NEON usage benchmarks but moderate IS. However, this is not a problem in many applications. For example, *bitcount* and *patricia* have only 0.00% and 0.55% of NEON instructions, but 15.44% and 17.05% of ISs, respectively. Although the number of ISs is high, the quantity of NEON instructions is close to 0%. Thus, these cases do not represent a problem to the offloader. On the other hand, applications such as *basicmath*, *fft*, and *fft-i* use NEON more intensively and have a higher number of ISs compared to other benchmarks.

In addition to the NEON usage, it is important to consider how these instructions are distributed throughout program execution. Figure 4.7 presents three possible NEON instructions distributions in programs. Applications with *steady* distributions (Figure 4.7a) have a constant number of NEON instructions along execution. This type of distribution makes the arbiter's job easier since it does not need to trigger frequent migrations between cores. Thus, if the rate of NEON instructions is low, the application will execute entirely with the offloader to save energy. On the other hand, a high rate of NEON instructions will execute entirely in the big core. Some applications tend to *cluster* its NEON usage in a specific phase (Figure 4.7b). In this case, the arbiter is able to detect the high NEON usage and deactivate the offloader to keep performance and energy. The last case is the *noisy* behavior (Figure 4.7c. Applications with this behavior alternate between high and low usage of NEON instructions, which may be a problem for the arbiter since it can mispredict the current application phase.

Figure 4.7: Possible NEON instructions distributions.



(a) Steady (b) Cluster (c) Noisy

Source: The author.

However, real applications may have several different phases, each one similar to the basic distributions presented in Figure 4.7. We present the distribution of NEON instruction in evaluated applications in Appendix A.

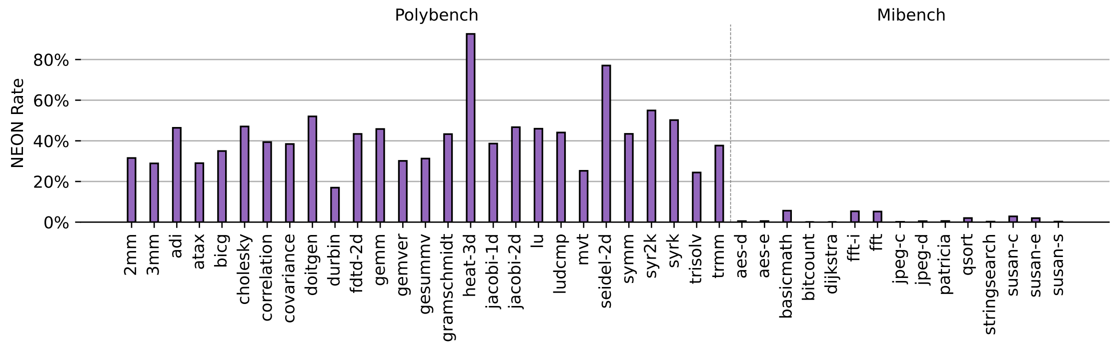
5 RESULTS

In this chapter, we present and discuss the results of the proposed work. First, we evaluate the decoupled offloader in a single-core scenario, i.e., with only the big core active and offloading instructions to an idle little core. In this first set of experiments, we focus on evaluating how the offloader impacts the big core. This allows us to assess the best case possible for the technique. After that, we use the offloader to evaluate partial cores with full-ISA in a multi-core scenario with multiple applications running simultaneously.

5.1 Decoupled Offloader

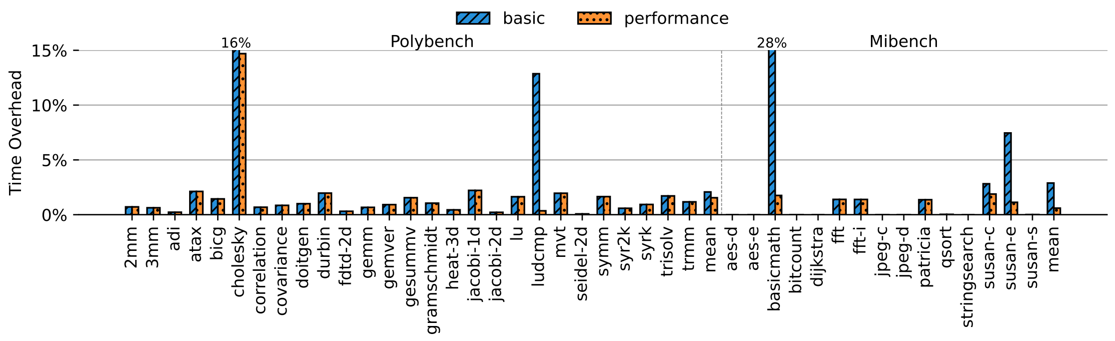
We evaluate how our decoupled offloader proposal can improve energy efficiency with low time overhead considering a big.LITTLE system as the baseline. In this first set of experiments, we consider that only the big core executes an application, and the paired little core, which receives offloaded instructions, is idle. This allows us to assess the maximal improvements provided by the offloader. In this experiment we use applications from polybench (POUCHET, 2015) and mibench (GUTHAUS et al., 2001), discussed in Section 4.5. Figure 5.1 shows the percentage of NEON instructions in all evaluated applications. Since the benchmarks range from high to low NEON usage, we can analyze the proposed system in different NEON utilization scenarios. Our goal is to obtain high energy gains with minimal time overhead for both scenarios. For polybench, we expect gains when possible since it uses NEON intensively, and we expect high gains in mibench. We consider both arbiter configurations (Basic and Performance) in all experiments.

Figure 5.1: Percentage of NEON instructions in each benchmark.



Source: The author.

Figure 5.2: Execution time overhead.



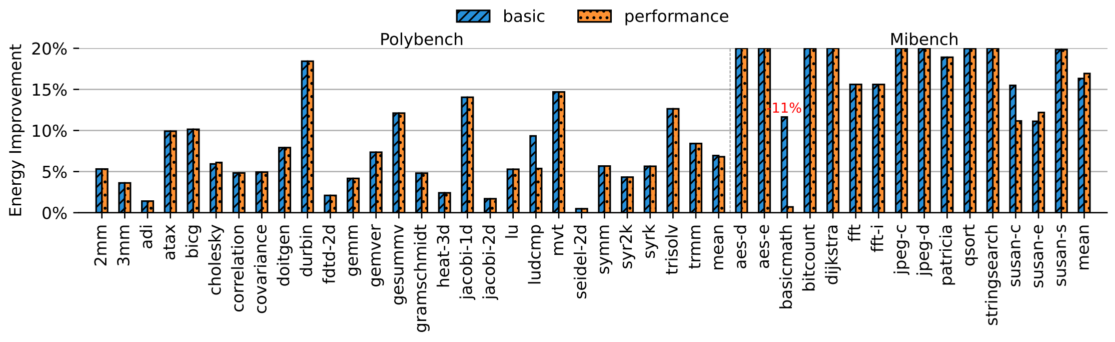
Source: The author.

To provide real gains, the offloader must not increase execution time while still providing energy gains. Therefore, we want to maintain the big core performance and improve its energy efficiency. Figure 5.2 shows the time overhead for each arbiter configuration: Basic and Performance. The time overhead occurs when the big core cannot complete instructions due to the offloader (e.g., a full FIFO or synchronization), as discussed in Section 3.1. For polybench benchmarks, the Basic arbiter has a 2.1% mean overhead while the Performance configuration achieved 1.5%. Except for *cholesky* and *ludcmp* apps, both arbiters perform equally. The Performance arbiter was slightly faster (14.7%) than the Basic (16.4%) for the *cholesky*. The difference between both arbiters is significant for *ludcmp*, with 12.9% overhead for Basic and just 0.4% for Performance. The mean overheads show that both arbiters do minor harm to application execution. However, the results for *ludcmp* show that the Performance arbiter can make a substantial difference in this case.

The advantage of Performance over the Basic arbiter is more evident for mibench since the mean overhead was 2.9% and 0.6% for Basic and Performance, respectively. The most significant advantage of Performance over Basic occurs in *basicmath*, with 28.8% for Basic and 1.8% for Performance. For *susan-c* and *susan-e*, the Performance achieved lower overhead (1.9% and 1.1%) when compared to Basic (2.8% and 7.5%). The results for polybench and mibench show that both arbiters tend to perform equally well for the majority of the applications. However, Performance presents an improvement in specific cases, as expected, since it prioritizes using the big core instead of the offloader.

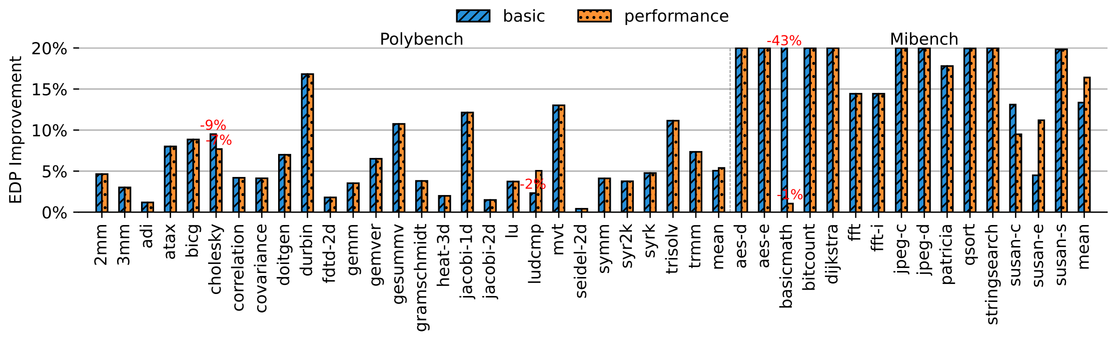
However, the low execution time must come with energy improvements. Figure 5.3 shows the energy gains for each scenario. The Basic provides a slightly higher mean energy efficiency for polybench (6.9%) than Performance (6.8%). That is expected since the Basic arbiter tends to use more the offloader than Performance. When analyzing

Figure 5.3: Energy improvement.



Source: The author.

Figure 5.4: EDP improvement.



Source: The author.

specific applications, Performance provides mild gains for *cholesky* (6.1%) over Basic (5.9%), whereas the Basic energy improvement for *ludcmp* (9.3%) is higher than Performance (5.4%). Since the Basic arbiter attempts to use the offloader as much as possible, it is expected to be more energy-efficient than Performance. For mibench, the Performance provides more energy gains (16.9%) than Basic (16.3%). The biggest difference occurs in *basicmath*, since the Basic worsens energy consumption by -11.6% (we discuss this result later in this Section), whereas Performance provides a mild improvement (0.7%). For *susan-c*, the Basic provides higher energy gains (15.4%) than Performance (11.2%), but this changes for *susan*-e with 11.1% and 12.2% for Basic and Performance, respectively. Both arbiters tend to provide similar energy improvements for all other applications. However, only Performance provides energy gains for all applications since it can execute *basicmath* with lower time overhead compared to Basic. The Performance's advantage in *basicmath* comes from its ability to turn off the offloader in application phases that are hard to predict. We discuss basicmath's characteristics later in this section.

To assess if the offloader provides energy gains with a low time overhead, we perform an EDP evaluation for each benchmark, presented in Figure 5.4. For polybench,

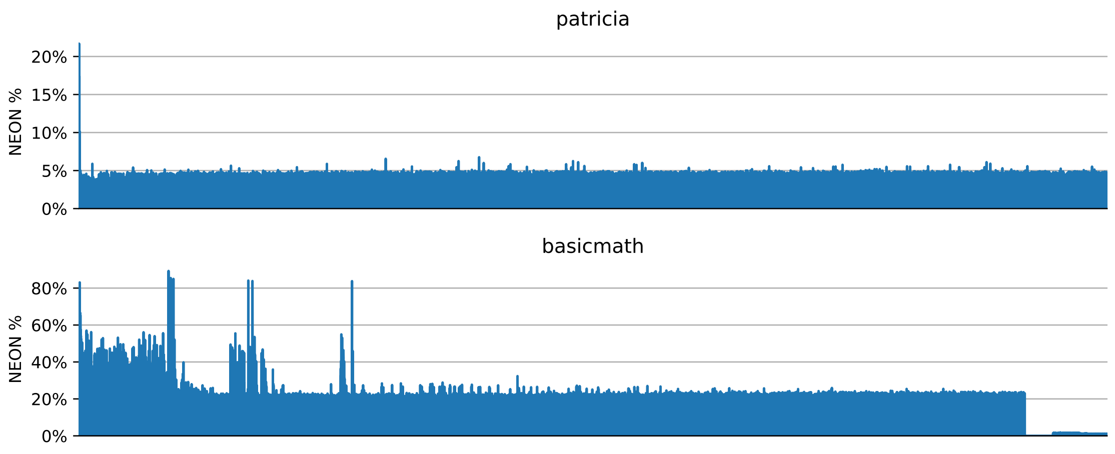
Table 5.1: Mean execution time overhead, energy, and EDP savings for each benchmark set and arbiter.

	Basic		Performance	
	polybench	mibench	polybench	mibench
Overhead	2.1%	2.9%	1.5%	0.6%
Energy	6.9%	16.3%	6.8%	16.9%
EDP	5.0%	13.4%	5.4%	16.4%

which is NEON intensive, the mean EDP improvement is 5.0% and 5.4% for Basic and Performance, respectively. The low EDP improvement is expected because polybench tends to use NEON intensively, and thus, the arbiters have less room to use the offloader. The EDP gains are higher for mibench, 13.3%, and 16.4% for Basic and Performance. For *cholesky*, the Perfomance arbiter is slightly better (-7.7%) than Basic (-9.5%) since it executes the benchmark with lower overhead and better energy efficiency. However, both arbiter configurations are worse than the baseline since they have negative EDP values. The difference between both arbiters is bigger for *ludcmp* since the Performance improves EDP by 5.0%, whereas the Basic worsens by -2.3%. The reason is that although Performance provides fewer energy gains than Basic, the considerable difference in overhead compensates. The difference between the arbiters is more significant for *basicmath* since Performance (-1.0%) presents EDP closer to the baseline than Basic (-43.9%). Except for *susan-c*, the Performance arbiter achieves better or the same EDP as Basic for all applications. Table 5.1 summarizes the mean values (overhead, energy, and EDP) obtained by the offloader for each benchmark set per arbiter configuration. Our technique can provide energy and EDP gains at low time overheads in high and low NEON usage scenarios, represented by polybench and mibench applications. As discussed in Section 4.5, polybench represents math applications that tend to use NEON intensively, allowing few opportunities to use the offloader. On the other hand, mibench represents different kinds of applications that use fewer NEON instructions and thus, allow the offloader to save energy.

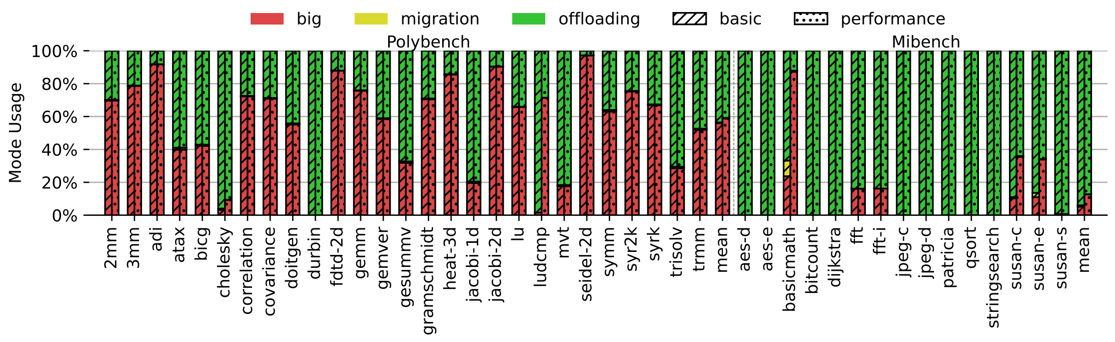
To understand why *basicmath* has the worst result using the Basic arbiter, it is necessary to evaluate the application behavior during the program execution. Figure 5.5 shows the percentage of NEON instructions committed in intervals of 1000 cycles (the same value as the arbiter window cycle) for *patricia* and *basicmath*, respectively. We pick those benchmarks because both present a noisy rate of NEON instructions distributed during the execution, which means that both applications alternate quickly between high and low usage periods of NEON, which can mislead the arbiter to make wrong predictions.

Figure 5.5: NEON commits during the benchmark execution.



Source: The author.

Figure 5.6: Percentage of benchmark execution in each mode.

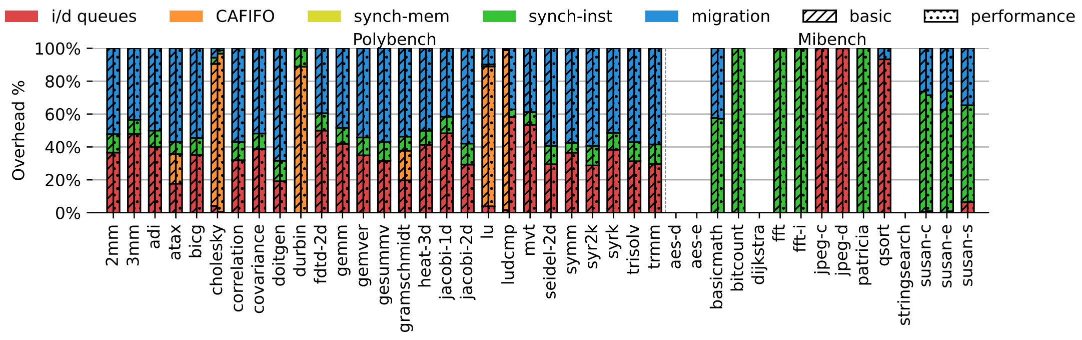


Source: The author.

The problem of this type of distribution arises when a high NEON instruction rate triggers the arbiter to change to normal mode (i.e., turn off the offloader) and execute NEON instructions in the big core high-performance FUs. However, the high peak may not last long, and as soon as the NEON rate goes below 20%, which is our predefined set parameter, the offloader is activated once again. The Performance arbiter solves this problem by detecting the high number of mode changes and turning off the offloader for 100K cycles. After this period, it wakes up and analyzes whether the problematic phase is gone. Although there is a noisy distribution in *patricia*, its peak value tends to be low and does not negatively influence the offloading system. Moreover, it represents the ideal application to our technique since its bursts of NEON usage can be handled by the offloader with low time overhead to improve energy efficiency.

The time and energy differences between both arbiters depend on the offloader utilization. Figure 5.6 shows the percentage of mode execution in program execution. The stacked bars represent, from bottom to up, the execution time spent in *big*, not offloading,

Figure 5.7: Overhead sources in each benchmark execution.

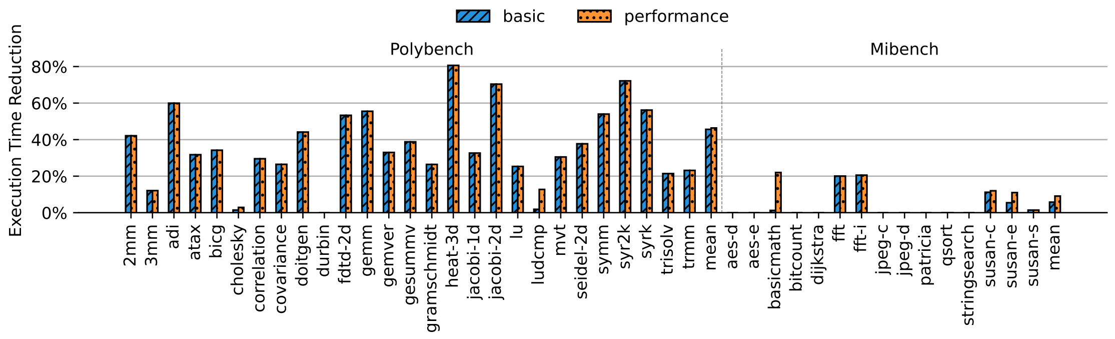


Source: The author.

migration, and *offloading*. Ideally, the migration time must not consume significant execution time since it is an overhead of the technique. Both arbiters provide similar mean results for polybench. Basic spends 56.0%, 0.5%, and 43.5% in big, migration, and offloading, respectively. On the other hand, Performance spends 58.8%, 0.5%, and 40.7% in each mode. This difference is slightly bigger in mibench, since Basic spends 5.2%, 0.9%, and 94.0% in each mode, whereas Performance spends 12.6%, 0.1%, and 87.3%. Overall, Performance provides lower time overhead than basic because it prioritizes the big core utilization, which makes a significant difference in *cholesky*, *ludcmp*, *basicmath*, *susan-c*, and *susan-e*. The Performance arbiter spends negligible time in migration for all benchmarks, whereas Basic has difficulty handling basicmath, which triggers frequent mode changes.

Next, we analyze offloader overheads sources and their impacts on application execution. Figure 5.7 shows the impact of each overhead source in each application execution. As discussed in Section 3.1, the offloader can be affected by different overhead sources. The first source is *i/d queues* and represents the number of cycles the big core cannot commit because it is waiting for an entry in the instruction or data queues. Similarly, the *CAFIFO* also accounts for the number of cycles waiting for a free entry in the structure. The *synch-mem* and *synch-inst* count the number of cycles spent in memory and instruction synchronizations. The last source, *migration*, counts the time spent in mode migrations, i.e., the time spent turning on and off the offloader. The offloader does not cause any overhead for four benchmarks: *aes-d*, *aes-e*, *dijkstra*, and *stringsearch*. The most common source of overhead in polybench is arbiter migration, followed by i/d queues. The CAFIFO represents a significant overhead source only for few benchmarks: *cholesky*, *durbin*, *lu*, and *ludcmp* using the Basic arbiter. When considering the synchronizations, the memory synchronization does not significantly increase the overhead,

Figure 5.8: Execution time reduction with an arbiter compared to not using an arbiter.



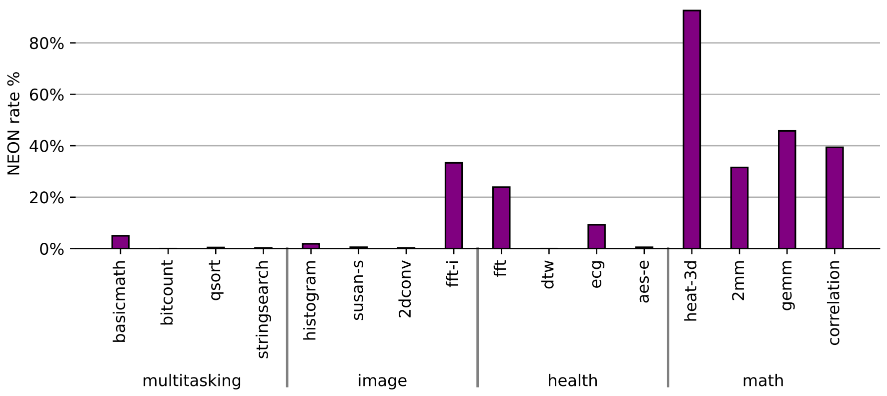
Source: The author.

whereas instruction synchronization has a meaningful impact on the program execution, especially in mibench benchmarks.

Finally, we also evaluate the performance of all applications in the absence of an arbiter. This experiment allows us to reason how effective an arbiter is for each application. Figure 5.8 shows the execution time reduction for each benchmark when using both arbiters compared to running the entire application in offload mode as the baseline, i.e., without the possibility to execute NEON instructions in big in high usage phases. Therefore, a positive result means that the arbiter was able to reduce the execution time. For polybench, Basic and Performance reduce mean execution time by 45.6% and 46.4%, respectively. That shows that having an arbiter is essential to bypass high NEON usage phases and keep time overhead low. For mibench, having an arbiter is less significant since the mean overhead reductions are 5.8% and 9.1% for Basic and Performance, respectively. Some benchmarks (e.g., *durbin* and *bitcount*) do not benefit from having an arbiter since they have almost 0% difference from the baseline.

Despite its similarities, Performance is worse than Basic only for specific applications. On the other hand, programs that are hard to predict present a big advantage when executed with the Performance arbiter because it can handle problematic application phases by compulsory turning off the offloader for a longer period. That enables the system to improve energy and EDP significantly at low time overheads in scenarios that Basic fails to. Thus, the Performance is the best option to compose a system with decoupled offloader.

Figure 5.9: Percentage of NEON instructions in each benchmark.



Source: The author.

5.2 Partial Core with the same ISA

In this next set of experiments, we evaluate the capabilities of partial cores with the same ISA. Different from the previous experiments that focused on the improvements of the offloader in an ideal scenario, i.e., with the offloader sending instructions to an idle little core, this experiment evaluates a multi-core scenario as depicted in Figure 3.9. Our goal is to assess how incrementing the number of partial cores in a system impacts the overall system's performance and energy consumption compared to the standard big.LITTLE. We use the decoupled offloader to allow a partial core to maintain full ISA support by offloading its NEON instructions to a paired little core. This approach allows the partial cores to still execute NEON applications without recurring to emulation or thread migration to a full core. We evaluate systems with different combinations of big, little, and partial cores. We consider the ARM big.LITTLE (4 bigs+4 littles) as the baseline system in our experiments. We evaluate scenarios 1f3p, 2fp2, and 3f1p, where "f" and "p" stand for the number of full and partial cores in a system. Therefore, the baseline is equivalent to 4f0p. We consider that the full cores also have the decoupled offloader equipped with the Performance arbiter since it provides the best results, as concluded in Section 5.1.

For each system, we assign four workloads to big and little cores. The workloads represent common embedded system tasks (multitasking, image, health, and math) coming from benchmarks found in (TAN et al., 2016; GUTHAUS et al., 2001; POUCHET,

Table 5.2: Workload scenarios evaluated, their benchmarks, and the initial workload schedule for each system.

Workload	Benchmarks	3f1p	2f2p	1f3p
multitaskting	basicmath; bitcount; qsort; stringsearch	f	f	f
image	histogram; susan-s; 2dconv; fft-i	f	f	p
health	fft; dtw; ecg; aes-e	f	p	p
math	heat-3d; 2mm; gemm; correlation	p	p	p

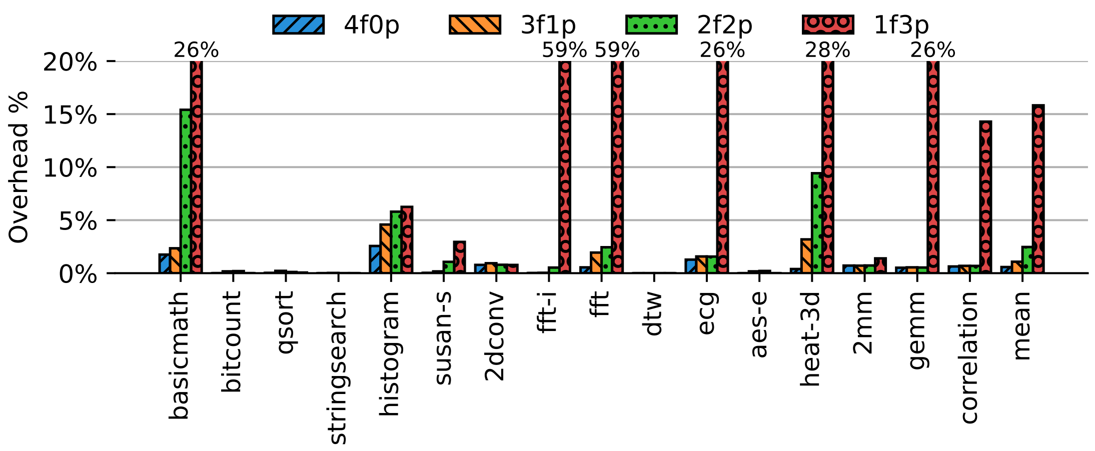
2015)[1]. Each workload comprise 4 jobs, which are executed in order. We evaluate the full system in this experiment, including the little cores, since each pair of big and little cores receive the same workload. This allows us to evaluate the overhead caused in the little core due to the interference of sharing its datapath. Moreover, it is an additional source of overhead to the offloaded instruction stream since it executes concurrently with little's instructions. We give priority to little's instruction stream instead of the offloader instruction stream, as discussed in Section 4.2.

At the beginning of the simulation, an application is assigned to each core. The little cores receive the same workload as their paired big cores. When a core finishes a benchmark execution, the simulator assigns the next benchmark of the batch to the same core. We simulate traces of 10M instructions of each job in the batch. Figure 5.9 depicts the rate of NEON instructions executed in each trace showing that we evaluate a broad range of NEON rate programs, from low to high utilization. Our initial thread mapping prioritizes assigning high NEON apps to partial cores to force the worst-case scenario in our experiments. Table 5.2 summarizes the workloads, their benchmarks, and the initial mapping.

Figure 5.2 shows the time overhead of the 4f0p, which has full cores equipped with the offloader, and the partial cores systems. Ideally, the overhead of each thread execution in partial cores systems must not be higher than the 4f0p execution. Thus, the scheduling heuristic plays an important role in achieving this task since it is responsible for thread mapping between full and partial cores. Overall, the overhead difference between the 4f0p and 3f1p experiments is low for all applications. The biggest overhead occurs in *heat-3d* application since the 4f0p suffers only a small time penalty of 0.39% compared to 3.18% in 3f1p. This difference occurs due to our initial thread mapping, which maps heavy NEON usage apps to partial cores, causing the scheduler to move the math program (*heat-3d*) to a full core. After the thread swap, the subsequent jobs in each workload execute with negligible overhead compared to the 4f0p system. The mean overhead is

[1]We use two benchmarks from TAN et al. to create representative utilization scenarios.

Figure 5.10: Execution time overhead.



Source: The author.

1.08%, close to 0.57% of the ST experiment. Therefore, substituting a full core for a partial core does not severely penalize the system's performance in this case.

When considering the 2f2p system, the main affected applications are *basicmath* and *heat-3d*. This overhead occurs due to frequent thread migrations between both applications. Although having a bigger percentage of NEON instructions, *heat-3d* has two distinct phases. The first has almost no NEON instructions, whereas the second and longer phase uses NEON intensively, thus, leading the scheduler to swap both applications more often. However, for the other applications, the overhead increase compared to 4f0p execution is low, as shown by the 2.46% mean overhead of the 2f2p. The job execution order explains the low overhead. Since we simulate 10M instructions of each benchmark, the subsequent jobs initiate almost simultaneously. It is important to notice that there is no more than 2 NEON demanding workloads simultaneously. Thus, the 2f2p system can run all workloads with satisfactory performance.

However, the number of demanding NEON threads running simultaneously severely impacts the performance of the 1f3p system. For this scenario, the mean overhead increase is 15.83% compared to big.LITTLE. That is expected since all high NEON usage workloads must share the single full core. The results show that 3f1p and 2f2p can provide good execution time overhead with the considered workloads.

Applications as *bitcount*, *qsort*, *stringsearch*, *2dconv*, *dtw*, and *aes-e* have few NEON instructions and are low penalized by the increase of partial cores, i.e., being partial core friendly. Despite that, a system with partial-ISA cores would depend on OS support to migrate the thread to a full core or emulate the required NEON instructions, slowing down the application or even the entire system. Our solution, partial cores with

Table 5.3: Energy and EDP compared to big.LITTLE.

	Energy	EDP
4f0p	88.7%	89.3%
3f1p	89.8%	90.8%
2f2p	90.2%	92.4%
1f3p	98.8%	115.5%

Table 5.4: Area compared to 4f+4l.

3f1p	2f2p	1f3p	f	p	l
85.4%	70.7%	56.1%	21.2%	13.7%	3.8%

full ISA, handles these cases efficiently and reduces the pressure for full cores. With the reported execution time overheads, we conclude that a system designer must pick the number of partial cores with the target workloads in mind since the mean overheads of each system depend strongly on the applications.

Table 5.3 shows the energy and EDP of each system to run all workloads. Ideally, the energy consumption of executing the workloads in parallel must be as close as possible to the 4f0p execution and is limited by the factor of 80%, which is the system's power consumption when working with the offloader. The 4f0p consumed 88.7% of the energy of the baseline (big.LITTLE) while the partial core systems consumed 89.8%, 90.2%, and 98.8% for 3f1p, 2f2p, and 1f3p, respectively. The 3f1p and 2f2p systems are close to the ideal case (4f0p), whereas the 1f3p provides timid gains over the baseline. However, energy consumption must come with a low execution time overhead to provide real gains. The EDP results for 3f1p, 2f2p, and 1f3p are 90.8%, 92.4%, and 115.5%. 3f1p and 2f2p systems provide real gains because they achieve better energy results at the cost of a low execution time increase. However, the 1f3p fails to provide any EDP gains since its result is 15.5% above the baseline.

Table 5.4 shows the area consumption for each system and core type compared to big.LITTLE as the baseline. We model the area using synthesis reports of Boom and Rocket cores, as discussed in Section 4.3. We consider that the partial core is a BOOM processor without FP hardware. One full core occupies circa 21.2% of the baseline (4 bigs + 4 littles), while partial occupies only 13.7%. A partial core area is 35.2% lower than a full core area. A designer can take advantage of the area reductions provided by partial cores to increase the system's TLP or increase the number of specialized accelerators with the freed area. When considering a system crafted for niche scenarios like Neural Networks, which are well suited for data parallelism, it is possible to use partial cores to free space for accelerators. In this case, the accelerators run the heavy SIMD kernels, thus, reducing the load of NEON instructions executed by CPUs. To illustrate this, we also synthesize the Gemmini (GENC et al., 2021) systolic array. Its area represents circa 21.5% of the full system. Thus, the area reduction provided by 2f2p and 1f3p systems is more than enough to include the accelerator.

Different from the experiments in Section 5.1, we no longer consider little cores as idle. Thus, the offloaded instruction stream competes for the NEON resources with little's thread. The impact of FU sharing in little cores depends mainly on how they are shared. As discussed in 4.2, we consider the NEON ISA to use two FUs in little's datapath: The FP/SIMD for computing NEON instructions and the memory access FU to memory NEON instructions. In our FU sharing model, we always prioritize little's running thread over the offloaded instruction stream, which might increase the overhead of the offloader. In this case, if little's thread issues into a shared FU required by the offloaded stream, the offloader must wait for the next cycle to issue. Furthermore, we consider that only a single instruction of the offloaded stream must be in the execution stage at a time, which also increases the overhead of the offloader but results in a simpler implementation. Even with these considerations, we found that the impact on little's thread and on the offloaded instruction stream is negligible. This happens because the little InO core does not have the ability to sustain a high utilization rate of the shared FUs due to its limited ILP capabilities, and the offloaded stream only executes one instruction at a time. Thus, FU-sharing conflicts rarely occur.

6 CONCLUSION

In this work, we started with the observation that FP/SIMD instructions tend not to be used in many applications. In our analysis, We consider an AMC design inspired by ARM big.LITTLE, which features two types of core into the same chip: A big A15 (OoO) and a little A7 (InO) processor. Both cores support the same ARMv7 ISA and thus, support the same binaries. The ARMv7 ISA defines the NEON extension, which adds FP/SIMD to processors. However, this extension is implemented with different implementation costs in both cores. The big core provides two expensive NEON FUs to allow high-performance execution, whereas the little core has only a single and more energy-efficient NEON unit. Moreover, both NEON FUs in big occupy more area than four full A7 cores.

Given that, we introduced two approaches to improve the NEON costs in designs based on big.LITTLE. First, we introduced the *decoupled offloader* as an alternative to share FUs between big and little cores. We observed that some applications that execute few NEON instructions might not require the NEON high performance available in big. Thus, using the big's expensive NEON units result in energy waste in these cases. A solution is to allow the big core to use little's NEON unit through FU sharing. However, common FU sharing strategies are coupled and require that the big core issue instructions into the slow little's FU and wait for the result to return. Instead of this coupled sharing, we proposed a decoupled sharing alternative using a decoupled offloader. This way, the big core can offload its NEON instructions to little without waiting for the result of offloaded instructions. We discussed the necessary hardware modifications to implement this idea, considering the NEON ISA, its advantages, and drawbacks. Since applications may have different phases, and thus, with different NEON usage, we proposed a hardware arbiter to detect at runtime when the offloader should be used.

Furthermore, we used the decoupled offloader to propose *partial cores with full ISA*. We created partial cores from big cores by removing its NEON hardware and using the offloader to maintain the full ISA support. Thus, we enabled a new level of heterogeneity since partial cores have similar performance to big cores in integer applications at the cost of slower support in FP/SIMD and using less area. We discussed how this idea could be implemented and the necessary OS modifications to support it.

We evaluated both ideas using different benchmarks sources and compared them considering the ARM big.LITTLE as the baseline. We showed that decoupled offloader was able to reduce energy consumption with low time overhead, thus improving the EDP for many applications. We evaluated the partial cores with 4 workloads containing 4 jobs each. We considered 4 different combinations of full and partial cores. In our evaluation, systems 1f3p and 2f2p provided energy gains and EDP gains at low time overheads. Moreover, we showed how the freed area could be used to include accelerators in the design to improve gains further.

6.1 Limitations

We now discuss some limitations of this work.

Exception handling: The decoupled offloader strategy can only be used with coprocessor like ISAs that do not raise exceptions when executing offloaded instructions. As discussed in Section 3.1, the NEON ISA allows the programmer to choose whether data processing NEON instructions raise exceptions or not. When not raising exceptions, the programmer can query the FPSCR register to know if an exception occurred without unexpected changes in the program flow. When that is not the case, one should fall back to a coupled sharing approach.

Cache utilization: We use an in-house simulator that works on traces produced by gem5 to simulate the offloader. Although the in-house simulator models the offloader impact, its simulation model fails to measure the impact on big and little caches. The problem occurs because offloaded instructions use little's cache instead of big's, and the simulator only measures the overhead caused by the offloader.

Area and energy modeling: In our analysis, we rely on area and energy results available in literature, which rely on McPAT simulations and on the synthesis of BOOM and Rocket cores. Although they allow estimating A7 and A15 area and energy metrics, one must keep in mind that they are not the real values. To the best of our knowledge, there is no detailed publicly available data on the area and energy consumption of these cores. Besides processor modeling, we do not take into account the area and energy impacts of adopting decoupled offloader, i.e., the cost to include its hardware structures into the system.

New benchmarks: We evaluated the decoupled offloader in different scenarios, using applications that range from low to high NEON usage. However, it would be interesting to analyze benchmarks that represent trending applications.

6.2 Future Work

In addition to the limitations listed in the previous section, future work can adapt the offloader to other coprocessor like ISA extensions such as RISC-V Vector extension and ARM’s Scalable Vector Extension (SVE). When considering partial cores with full ISA, future work can investigate more robust scheduler-aware alternatives. In this work, we propose a simple scheduling heuristic that can be embedded in any OS scheduler since it only has to decide which type of big core is better according to the NEON usage. However, when there are more demanding NEON applications assigned to big processors than the number of full cores, the programs can suffer severe overheads. Future work can address this problem by executing more demanding NEON applications in little cores instead of partial cores.

REFERENCES

Apple. **Apple unleashes M1**. 2020. <https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/>.

ARM. **ARM Cortex-A Series Programmer's Guide**. [S.l.], 2014.

ASANOVIć, K. et al. **The Rocket Chip Generator**. [S.l.], 2016. Available from Internet: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>.

BECKER, P. H. E.; SOUZA, J. D.; BECK, A. C. S. Tuning the isa for increased heterogeneous computation in mpsocs. In: **2020 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2020. p. 1722–1727.

BIENIA, C. et al. The parsec benchmark suite: Characterization and architectural implications. In: . New York, NY, USA: Association for Computing Machinery, 2008. (PACT '08), p. 72–81. ISBN 9781605582825. Available from Internet: <https://doi.org/10.1145/1454115.1454128>.

bigLITTLE. 2011. <https://www.arm.com/why-arm/technologies/dynamiq>.

BINKERT, N. et al. The gem5 simulator. **ACM SIGARCH Computer Architecture News**, ACM, v. 39, n. 2, p. 1, aug 2011. ISSN 01635964. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2024716.2024718>.

BORODIN, D.; SIAUW, W.; COTOFANA, S. D. Functional unit sharing between stacked processors in 3d integrated systems. In: **2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation**. [S.l.: s.n.], 2011. p. 311–317.

BUTLER, M. et al. Bulldozer: An approach to multithreaded compute performance. **IEEE Micro**, IEEE Computer Society Press, Washington, DC, USA, v. 31, n. 2, p. 6–15, mar. 2011. ISSN 0272-1732. Available from Internet: <https://doi.org/10.1109/MM.2011.23>.

CONSTANTINOU, T. et al. Performance implications of single thread migration on a chip multi-core. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 4, p. 80–91, nov 2005. ISSN 0163-5964. Available from Internet: <https://doi.org/10.1145/1105734.1105745>.

DynamIQ. 2017. <https://www.arm.com/why-arm/technologies/dynamiq>.

ENDO, F. A.; COUROUSSé, D.; CHARLES, H.-P. Micro-architectural simulation of embedded core heterogeneity with gem5 and mcpat. In: **Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools**. New York, NY, USA: Association for Computing Machinery, 2015. (RAPIDO '15). ISBN 9781605586991. Available from Internet: <https://doi.org/10.1145/2693433.2693440>.

GENC, H. et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In: **Proceedings of the 58th Annual Design Automation Conference (DAC)**. [S.l.: s.n.], 2021.

GUTHAUS, M. et al. MiBench: A free, commercially representative embedded benchmark suite. In: **IEEE WWC-4'01**. [S.l.]: IEEE, 2001. p. 3–14.

HOMAYOUN, H. et al. Dynamically heterogeneous cores through 3d resource pooling. In: **IEEE International Symposium on High-Performance Comp Architecture**. [S.l.: s.n.], 2012. p. 1–12.

HU, Z. et al. Microarchitectural techniques for power gating of execution units. In: **Proceedings of the 2004 International Symposium on Low Power Electronics and Design (IEEE Cat. No.04TH8758)**. [S.l.: s.n.], 2004. p. 32–37.

INTEL. **Intel AVX-512**. 2013. <https://software.intel.com/en-us/blogs/2013/avx-512-instructions>.

INTEL. **Intel Deep Learning Boost**. 2021. <https://www.intel.com/content/www/us/en/artificial-intelligence/deep-learning-boost.html>.

INTEL. **Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2**. [S.l.], 2021. Available from Internet: <https://cdrdv2.intel.com/v1/dl/getContent/671427>.

KUMAR, R. et al. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In: **Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.** [S.l.: s.n.], 2003. p. 81–92.

KUMAR, R.; JOUPPI, N.; TULLSEN, D. Conjoined-core chip multiprocessing. In: **37th International Symposium on Microarchitecture (MICRO-37'04)**. [S.l.: s.n.], 2004. p. 195–206.

KUMAR, R. et al. Efficient power gating of simd accelerators through dynamic selective devectorization in an hw/sw codesigned environment. **ACM Trans. Archit. Code Optim.**, Association for Computing Machinery, New York, NY, USA, v. 11, n. 3, jul 2014. ISSN 1544-3566. Available from Internet: <https://doi.org/10.1145/2629681>.

LEE, W. et al. Exploring heterogeneous-isa core architectures for high-performance and energy-efficient mobile socs. In: **Proceedings of the on Great Lakes Symposium on VLSI 2017**. New York, NY, USA: Association for Computing Machinery, 2017. (GLSVLSI '17), p. 419–422. ISBN 9781450349727. Available from Internet: <https://doi.org/10.1145/3060403.3060408>.

LI, S. et al. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In: **2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2009. p. 469–480.

LI, T. et al. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In: **SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing**. [S.l.: s.n.], 2007. p. 1–11.

LOPES, B. C. et al. Shrink: Reducing the isa complexity via instruction recycling. In: **2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2015. p. 311–322.

LUKEFAHR, A. et al. Composite cores: Pushing heterogeneity into a core. In: **2012 45th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.: s.n.], 2012. p. 317–328.

MARTINS, M. et al. Open cell library in 15nm freepdk technology. In: **Proceedings of the 2015 Symposium on International Symposium on Physical Design**. New York, NY, USA: Association for Computing Machinery, 2015. (ISPD '15), p. 171–178. ISBN 9781450333993. Available from Internet: <https://doi.org/10.1145/2717764.2717783>.

NVIDIA. **Variable SMP – A Multi-Core CPU Architecture for Low Power and High Performance**. [S.l.], 2011. Available from Internet: <https://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf>.

PADMANABHA, S. et al. Dynamos: Dynamic schedule migration for heterogeneous cores. In: **2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2015. p. 322–333.

PADMANABHA, S. et al. Mirage cores: The illusion of many out-of-order cores using in-order hardware. In: **2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2017. p. 745–758.

PARKHURST, J. et al. From single core to multi-core: Preparing for a new exponential. In: **2006 IEEE/ACM International Conference on Computer Aided Design**. [S.l.: s.n.], 2006. p. 67–72.

PATTERSON, D. The trouble with multi-core. **IEEE Spectrum**, v. 47, n. 7, p. 28–32, 53, 2010.

POUCHET, L.-N. **PolyBench/C – The Polyhedral Benchmark suite**. 2015. <http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>.

POWELL, M. et al. Gated-v<sub>dd</sub>: A circuit technique to reduce leakage in deep-submicron cache memories. In: **Proceedings of the 2000 International Symposium on Low Power Electronics and Design**. New York, NY, USA: Association for Computing Machinery, 2000. (ISLPED '00), p. 90–95. ISBN 1581131909. Available from Internet: <https://doi.org/10.1145/344166.344526>.

RODRIGUES, R.; KOREN, I.; KUNDU, S. Does the sharing of execution units improve performance/power of multicores? **ACM Trans. Embed. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 14, n. 1, jan 2015. ISSN 1539-9087. Available from Internet: <https://doi.org/10.1145/2680543>.

RODRIGUES, R. et al. Performance and power benefits of sharing execution units between a high performance core and a low power core. In: **2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems**. [S.l.: s.n.], 2014. p. 204–209.

ROTEM, E. et al. Alder lake architecture. In: **2021 IEEE Hot Chips 33 Symposium (HCS)**. [S.l.: s.n.], 2021. p. 1–23.

RUKMABHATLA, N. et al. **Intel performance hybrid architecture & software optimizations Development Part Two: Developing for Intel performance hybrid**

architecture. [S.l.], 2021. Available from Internet: <https://cdrdv2.intel.com/v1/dl/getContent/685865>.

SOUZA, J. D. et al. Enhancing multithreaded performance of asymmetric multicores with simd offloading. In: **Proceedings of the 23rd Conference on Design, Automation and Test in Europe**. San Jose, CA, USA: EDA Consortium, 2020. (DATE '20), p. 967–970. ISBN 9783981926347.

SOUZA, J. D. et al. Improving multitask performance and energy consumption with partial-isa multicores. **Journal of Parallel and Distributed Computing**, v. 153, p. 1–14, 2021. ISSN 0743-7315. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0743731521000289>.

Sun Microsystems. **OpenSPARC T1 Microarchitecture Specification**. 2008. Https://www.oracle.com/servers/technologies/opensparc-t1-page.html.

TAN, C. et al. Locus: Low-power customizable many-core architecture for wearables. In: IEEE. **Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2016 International Conference on**. [S.l.], 2016.

TARAM, M.; VENKAT, A.; TULLSEN, D. Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency. In: **2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2018. p. 624–637.

VENKAT, A.; BASAVARAJ, H.; TULLSEN, D. M. Composite-isa cores: Enabling multi-isa heterogeneity using a single isa. In: **2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2019. p. 42–55.

VENKAT, A.; TULLSEN, D. M. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In: **Proceeding of the 41st Annual International Symposium on Computer Architecuture**. [S.l.]: IEEE Press, 2014. (ISCA '14), p. 121–132. ISBN 9781479943944.

WATERMAN, A. et al. **The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1**. [S.l.], 2016. Available from Internet: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html>.

XI, S. L. et al. Quantifying sources of error in mcpat and potential impacts on architectural studies. In: **2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2015. p. 577–589.

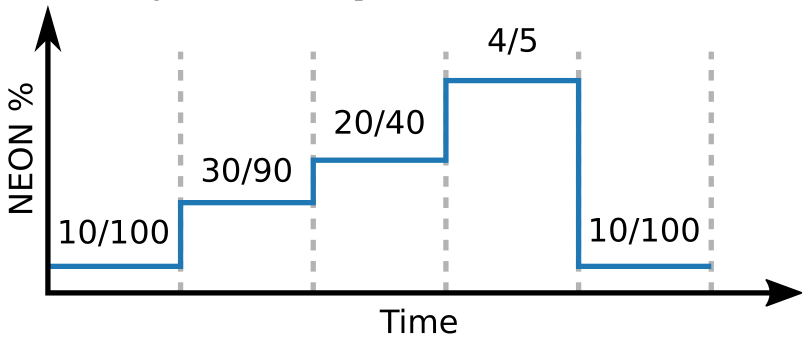
ZHAO, J. et al. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.

Appendices

AppendixA BENCHMARKS

This appendix presents the distribution of NEON instructions in polybench and mibench benchmarks. Figure A.1 depicts the methodology to measure the mean NEON usage throughout benchmark execution. We divide the number of NEON instructions by the total number of instructions in intervals of 1000 cycles, the same as used in the arbiters (Section 4.4). It is important to notice that some intervals might have high NEON usage because the total number of instructions is low, as is the case of the fourth interval, which 4 NEON instructions represent 80% of the total. This problem is discussed in Section 3.2.

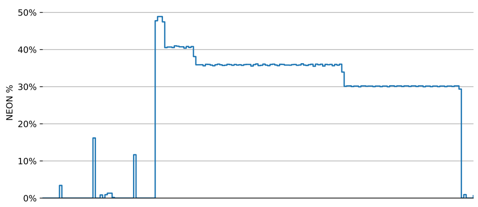
Figure A.1: Example of NEON distribution.



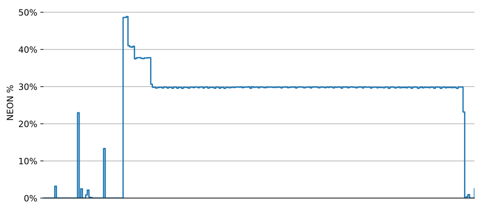
Source: The author.

A.1 Polybench

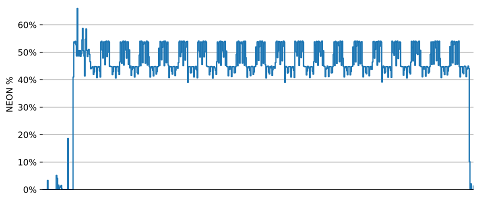
Figure A.2: NEON distribution in Polybench I.



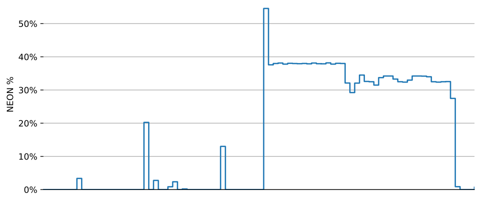
(a) 2mm



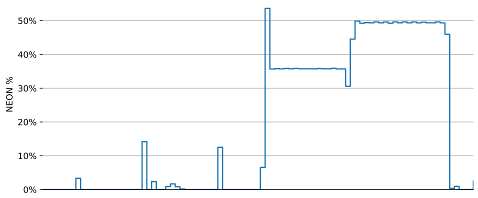
(b) 3mm



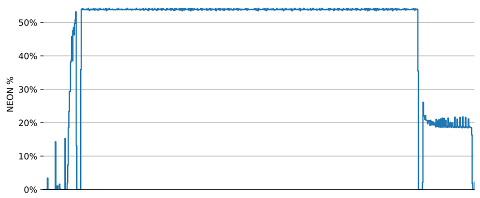
(c) adi



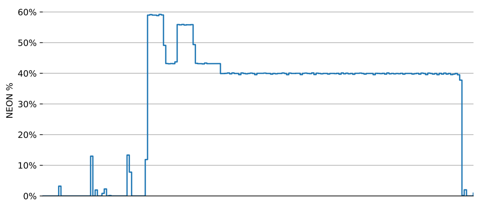
(d) atax



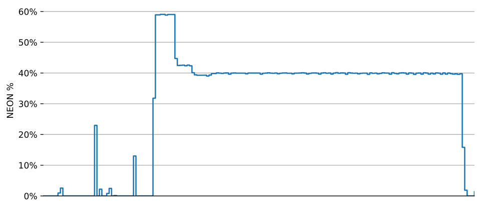
(e) bicg



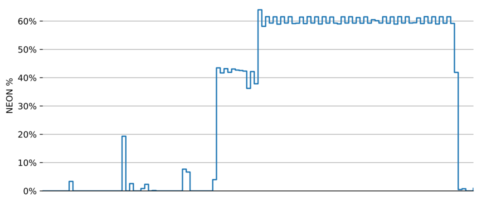
(f) cholesky



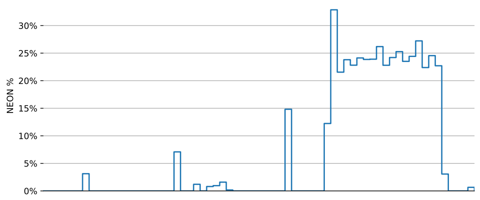
(g) correlation



(h) covariance



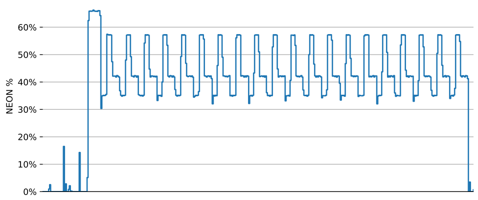
(i) doitgen



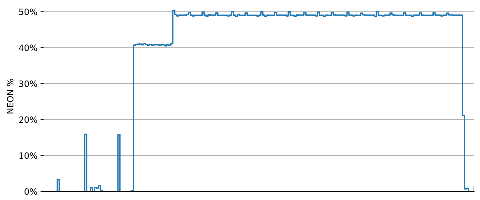
(j) durbin

Source: The author.

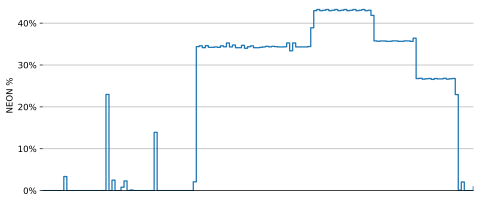
Figure A.2: NEON distribution in Polybench II.



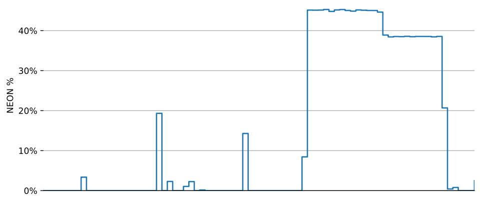
(k) fdtd-2d



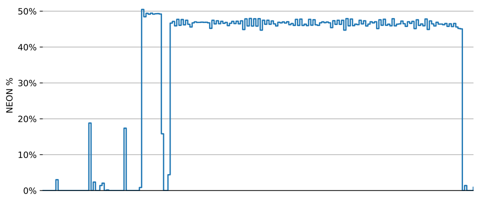
(l) gemm



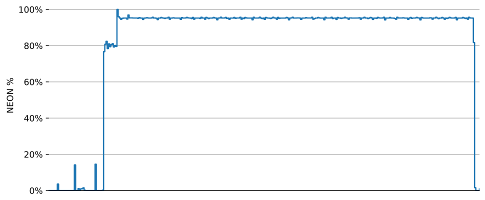
(m) gemver



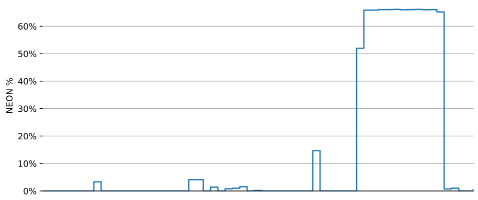
(n) gesummv



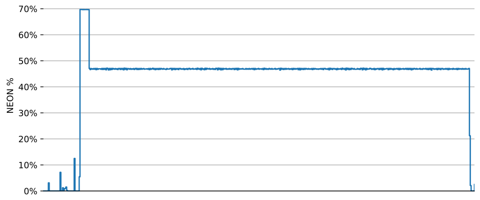
(o) gramschmidt



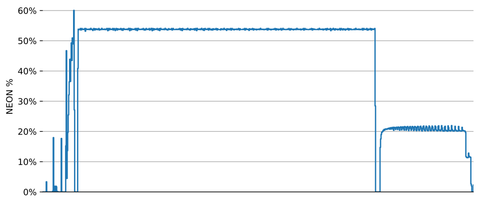
(p) heat-3d



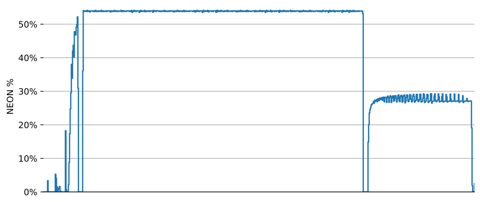
(q) jacobi-1d



(r) jacobi-2d



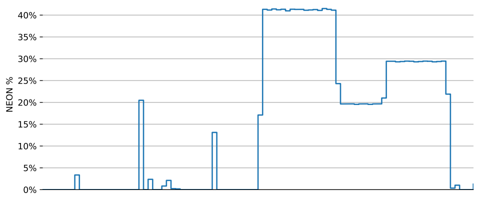
(s) ludcmp



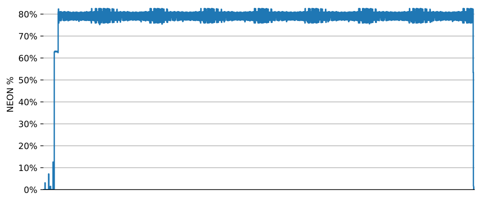
(t) lu

Source: The author.

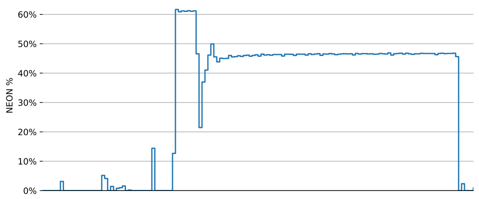
Figure A.2: NEON distribution in Polybench III.



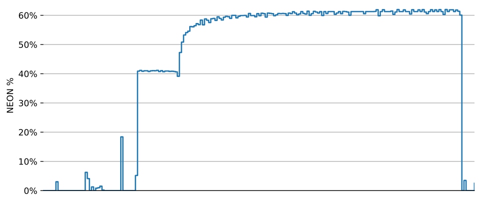
(u) mvt



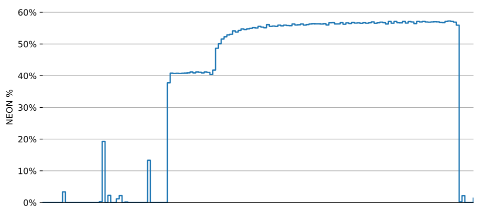
(v) seidel-2d



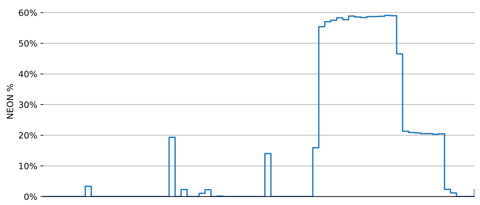
(w) symm



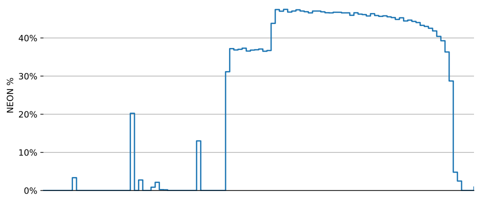
(x) syr2k



(y) syrk



(z) trisolv

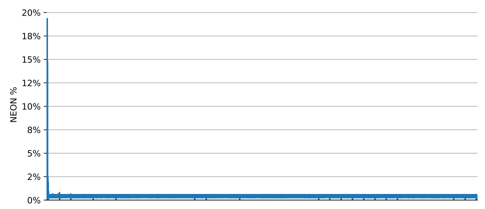


(aa) trmm

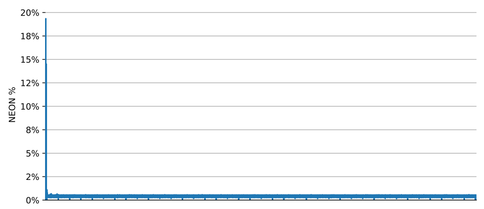
Source: The author.

A.2 Mibench

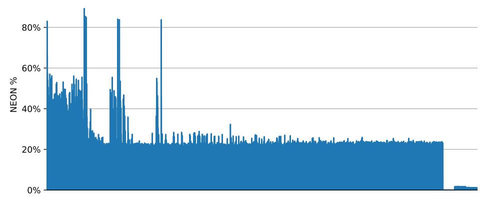
Figure A.3: NEON distribution in Mibench I.



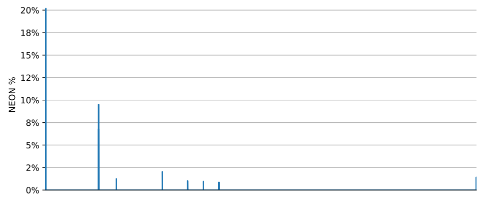
(a) aes-d



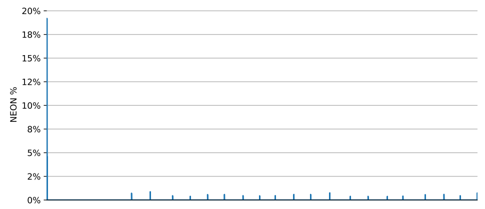
(b) aes-e



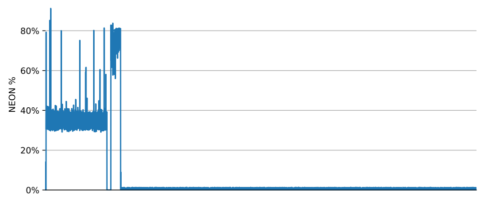
(c) basicmath



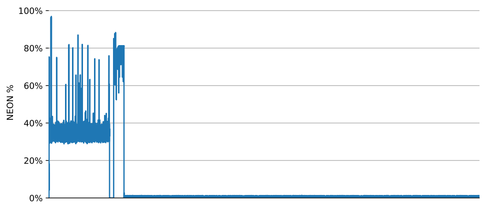
(d) bitcount



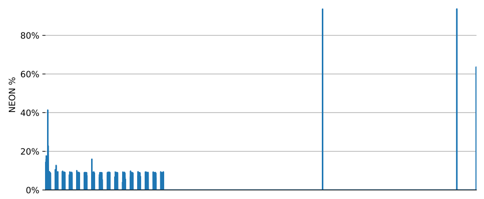
(e) dijkstra



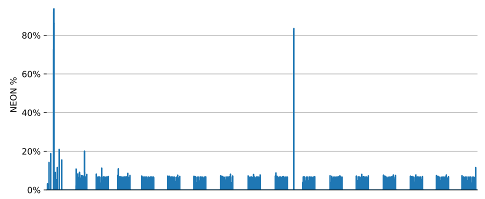
(f) fft



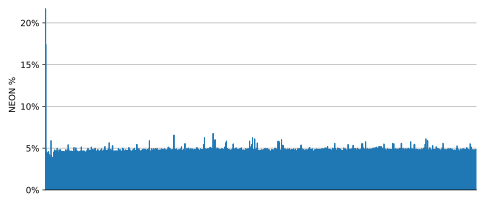
(g) fft-i



(h) jpeg-c



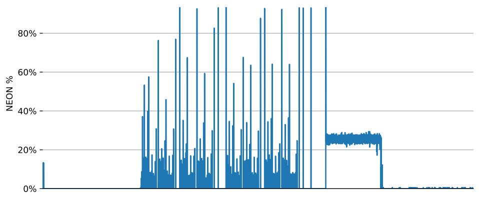
(i) jpeg-d



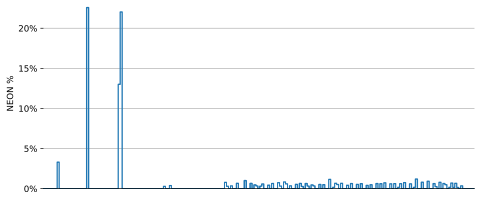
(j) patricia

Source: The author.

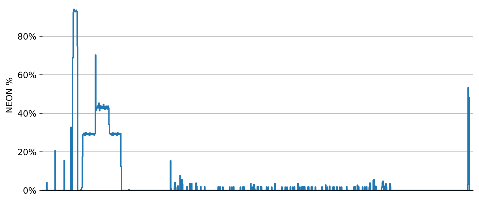
Figure A.3: NEON distribution in Mibench II.



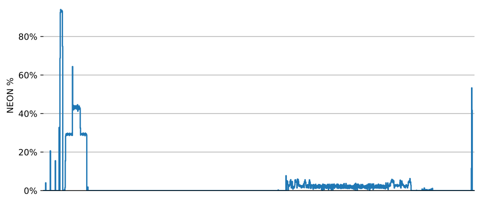
(k) qsort



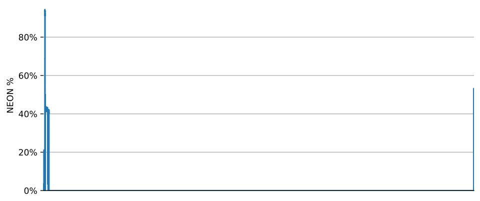
(l) stringsearch



(m) susan-c



(n) susan-e



(o) susan-s

Source: The author.

AppendixB TRACES

This appendix presents an overview of the traces used as input to the in-house simulator and the necessary gem5 modifications to generate them.

B.1 Gem5

The first step to generate traces is to modify gem5 to print them. This work considers the A7 and A15 CPUs, which are modeled differently in gem5. Each CPU model in gem5 has its class in the source code. For instance, the A7 core (InO) model is named *Minor* and the A15 (OoO) model is named *O3*. Both Minor and O3 classes are used to generic model InO and OoO CPUs, respectively. The CPU parameters (e.g., cache size and number of FUs) are input to the generic CPU models to simulate specific processors, such as A7 and A15.

Issue trace. We modify the *issue()* function of the Minor model to produce the A7's issue trace. The issue trace is a table containing 3 columns: Tick, FU index, and operation class (opClass). The Tick is an internal counter of gem5 and defines the simulation time. Since we consider cores running at 2 GHz, each CPU clock cycle equals 500 ticks. The FU index indicates the A7's FU that the instruction is executed. Although the A7 is modeled with multiple FUs, we are only interested in FUs that are capable of executing NEON instructions. Thus, we can safely remove the other FUs from the issue trace. The opClass indicates which operation the instruction does, e.g., if it is a load or store. This is important to retrieve the correct instruction latency since a FU can execute multiple classes of instructions. Table B.1 shows the first 4 instructions issued into the FUs of interest in polybench's *2mm*. The first two instructions are issued 1000 ticks, or two CPU cycles, away from each other. Both are issued into the same FU but have different opClasses. The first instruction (opCLass 47) is an ARM state read, whereas the second (opClass 48) is an ARM state write. Thus, both are using a memory access FU.

Table B.1: First four entries in polybench's *2mm* issue trace.

Tick	FU Index	opClass
161000	5	47
163000	5	48
238000	5	48
238500	5	47

Table B.2: First four entries in polybench's *2mm* commit trace.

Tick	Inst Addr	phyEffAddr	effSize	opClass	Asm
91000	10678	0	851988	1	mov.w fp, #0
91000	1067c	0	393216	1	mov.w lr, #0
254000	10680	55eb0	4	47	ldr_uop r1, [sp, #0]
254000	10680	0	393216	1	addi_uop sp, sp, #4

Commit trace We modify the *commitHead()* function of the O3 model to produce the A15's commit trace. This function computes on the head of the re-order buffer, i.e., the next instruction to commit. Table B.2 shows the commit trace organization and the the 4 first entries to commit in polybench's *2mm*. The commit trace has the following fields:

- **Tick:** Similarly to the issue trace, the Tick marks the simulation time the instruction was committed. We also consider that the A15 runs at 2 GHz, which causes each cycle to be 500 ticks.
- **Instruction address:** Memory address of the instruction.
- **Physical effective address:** Address accessed by the instruction. This field contains the address accessed by memory instructions. Instructions that do not perform memory accesses have the value 0 in this field.
- **Effective size:** Number of bytes accessed by a memory instruction. Non-memory instructions have a junk value in this field.
- **opClass:** Similar to the issue trace's opClass, this field contains the operation class of the instruction.
- **Assembly:** Instruction assembly. We parse the instruction mnemonic to discover its type and operands.

The first two entries of Table B.2 are *mov* instructions. The assembly uses the ARM's ABI nomenclature instead of raw register numbers. Thus, both *mov* instructions initialize the frame pointer (FP) and the intra procedure (IP) registers to 0. The third and fourth entries are micro-operations since they have the *uop* suffix and have the same instruction address, which are generated by a *pop* instruction. The *ldr_uop* loads 4 bytes into the stack pointer (SP) and the *addi_uop* sets the new value of the SP.

AppendixC RESUMO EM PORTUGUÊS

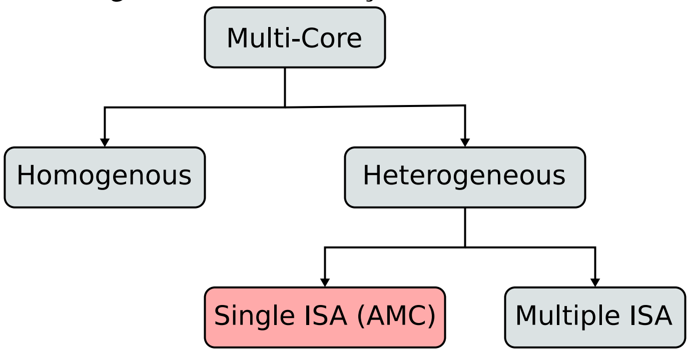
C.1 Introdução

Processadores multi-core surgiram como uma alternativa para melhorar o desempenho de computadores, pois soluções como o aumento da frequência do relógio e redução do processo de fabricação não produziam mais os mesmos ganhos de antes. Atualmente, existem várias categorias de processadores multi-core, como mostrado na Figure C.1. Um multi-core homogêneo possui todos os núcleos iguais, ou simétricos. Dessa forma, todos os núcleos possuem a mesmas *instruction set architecture (ISA)* e microarquitetura. Por outro lado, multi-cores heterogêneos podem apresentar núcleos que possuem a mesma ISA, mas com diferenças microarquiteturais, sendo esses classificados como *asymmetric multi-core (AMC)*, em destaque na Figura. Além disso, também há multi-cores com heterogeneidade a nível de ISA e, portanto, executam instruções diferentes.

Além do aumento do número de núcleos, outra alternativa para melhorar o desempenho de processadores é a adição de extensões ao ISA voltadas a nichos específicos de aplicação. Exemplos dessas extensões são *floating-point (FP)*, comumente usada em aplicações científicas, e *single instruction, multiple data (SIMD)*, que permite ao programador explorar o *data-level parallelism (DLP)*.

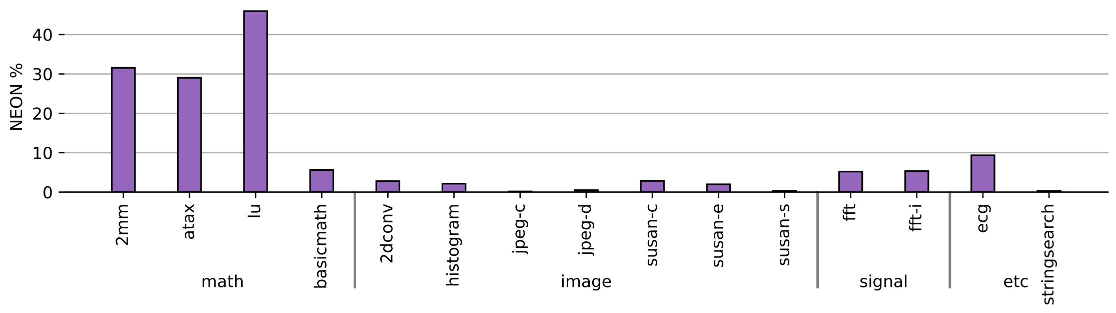
O multi-core AMC big.LITTLE da ARM possui dois tipos de processadores, o *big* (alto desempenho mas alto consumo de energia) e o *little* (alta eficiência energética mas baixo desempenho). O big.LITTLE possui a extensão NEON que adiciona operações FP e SIMD a processadores ARM. Contudo, essa extensão é implementada com diferentes custos em cada tipo de núcleo. Enquanto no big há duas unidades NEON de alto

Figure C.1: Classificação de multi-cores.



Source: The author.

Figure C.2: Quantidade de instruções NEON em diferentes classes de aplicação.



Source: The author.

desempenho, o little conta com apenas uma unidade mais simples. De acordo com trabalhos anteriores, as duas unidades NEON o big ocupam um espaço equivalente a quatro processadores little.

Contudo, como essas extensões são para nichos específicos de aplicação, elas tendem a ser pouco utilizadas em outros tipos de programas. A Figure C.2 apresenta a porcentagem de instruções NEON em diferente classes de aplicações. Enquanto nas aplicações matemáticas há uma predominância de NEON, outras mal utilizam a extensão. Nesse caso, o circuito não utilizado leva a um gasto desnecessário de área e energia. Neste trabalho, nós propomos soluções para amortizar os custos, em área e energia, da extensão NEON em um cenário AMC baseado no ARM big.LITTLE.

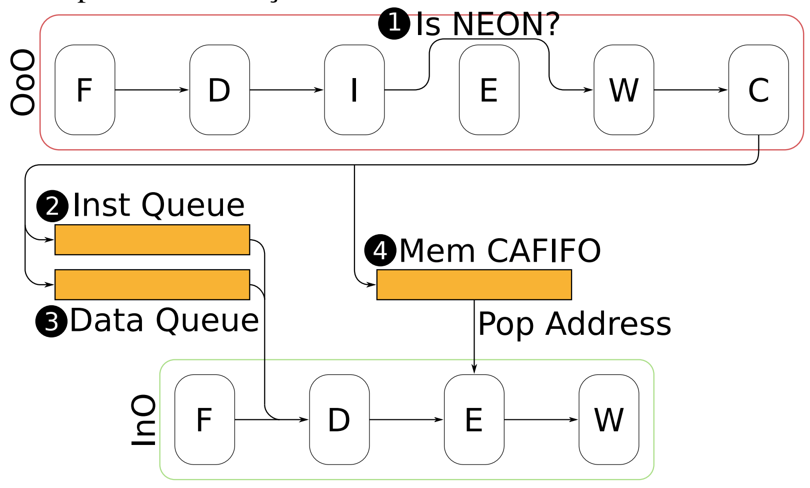
C.2 Trabalho proposto

C.2.1 Despacho desacoplado de instruções

O compartilhamento de recursos é uma alternativa para implementar extensões de ISA caras e que podem ser pouco utilizadas. Em um cenário AMC, uma alternativa para amenizar os custos é desligar as unidades NEON do big e permiti-lo executar suas instruções na unidade NEON do little, que é mais eficiente energeticamente. Em uma abordagem convencional, esse compartilhamento pode ser feito por meio do despacho de instruções. Nesse caso, o big envia suas instruções NEON para o little e espera pela conclusão da execução antes de seguir em frente. O problema dessa abordagem é que ela pode acabar retardando o big caso as instruções demorem demais para ser executadas.

O despacho de instruções desacoplado resolve esse problema ao permitir que o big

Figure C.3: Núcleos OoO (big) e little (InO) com as estruturas necessárias para suportar o despacho desacoplado de instruções.



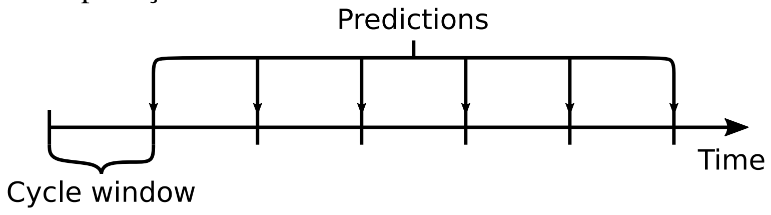
Source: The author.

envie suas instruções NEON para serem executadas no little e não espere pela conclusão delas. A Figura C.3 mostra como essa estratégia pode ser implementada considerando um núcleo big com execução fora de ordem — *out-of-order (OoO)* — e um núcleo little com execução em ordem — *in-order (InO)*. O primeiro passo ❶ para realizar o despacho é ignorar a execução de instruções NEON no big. Ao chegarem ao último estágio de *commit*, as instruções são despachadas para uma fila de instruções ❷ — *instruction queue*. Portanto, as instruções NEON concluem a sua passagem pelo big sem mudar o estado arquitetural. Por sua vez, o little executa as instruções NEON despachadas e armazenadas na fila de instruções pelo big sempre que possível.

Embora essa abordagem seja suficiente para suportar a maioria das instruções NEON, outros componentes precisam ser adicionados ao processador para garantir o suporte completo ao ISA. No caso de instruções que movimentam dados, é preciso adicionar também uma fila de dados — *data queue* ❸ — para armazenar temporariamente os dados consumidos pelas instruções despacahadas. Um exemplo desse tipo de instrução é a *vmov*, que permite ao programador transferir dados entre os bancos de registradores ARM (ISA principal) e os do NEON (extensão do ISA) e os do ARM.

Por fim, é preciso suportar instruções de acesso à memória. Diferente dos outros casos, é necessário manter a coerência entre nos acessos à memória. O problema surge pois, ao despachar as instruções para o little, o big não as executa e confia que o little irá executá-las em algum momento. Como exemplo, suponha que o big despachou uma instrução NEON para escrever o valor X no endereço Y. Do ponto de vista do programador, todas as instruções de leitura após o a escrita NEON devem ver o valor X no

Figure C.4: As predições do árbitro ocorrem ao final de um intervalo de ciclos.



Source: The author.

endereço Y. Portanto, é preciso garantir que o big não execute acessos à memória enquanto houverem instruções NEON pendentes para um mesmo endereço. Para resolver esse problema, nós adicionamos uma *content addressable FIFO (CAFIFO)* ❹, que é uma fila que permite buscar valores contidos nela. Ao fazer um acesso à memória, o big verifica se há despachos NEON pendentes para o mesmo endereço fazendo uma busca na CAFIFO. Se houver, então o big espera até que o little execute a instrução pendente.

C.2.2 Árbitro

Embora o despacho desacoplado de instruções seja uma alternativa para reduzir o gasto de energia em programas com baixo uso de NEON, ainda é necessário lidar com aplicações que usam a extensão intensamente. Nós propomos o uso de um árbitro para analisar a aplicação em tempo de execução e decidir sobre a utilização do despachador de instruções. Neste trabalho, nós analisamos dois tipos de árbitro, um básico e o outro voltado para desempenho, referidos doravante como *Basic* e *Performance*, respectivamente.

O árbitro Basic funciona conforme ilustrado na Figure C.4. O árbitro monitora durante uma quantidade ciclos (*cycles window*) as instruções executadas ou o desempenho do despacho de instruções. Ao fim de um intervalo, o árbitro faz uma predição sobre o comportamento do próximo intervalo. A decisão é baseada na máquina de estados apresentada na Figura 3.7. Quando o despachador de instruções está ligado (*on*), o árbitro avalia o atraso provacado no núcleo big devido ao despacho de instruções. Se o atraso for maior que o limiar de desligamento (*off threshold*), então o árbitro desativa o despacho de instruções e passa a executar instruções NEON dentro do big. Quando o despacho de instruções está desligado (*off*), o árbitro monitora a quantidade de instruções NEON em relação ao total de instruções completadas dentro do intervalo. Caso a representatividade de NEON em relação ao total de instruções seja menor que o limiar de ligar (*on threshold)*, o árbitro liga o despacho de instruções.

Figure C.5: Exemplo do árbitro em ação.

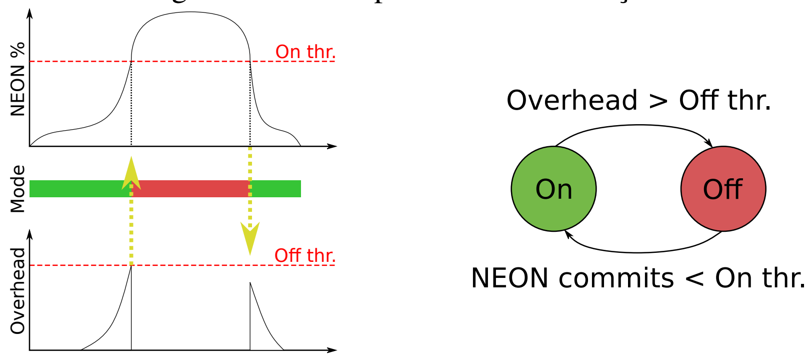
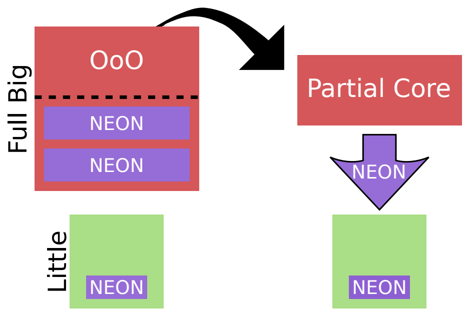


Figure C.6: Núcleos parciais são formados a partir de núcleos big removendo as suas unidades NEON. O suporte ao ISA é mantido por meio do despacho de instruções NEON para os núcleos little.



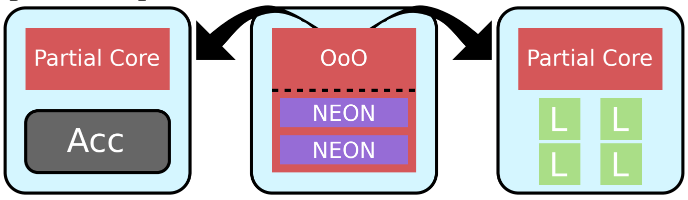
Source: The author.

Contudo, algumas aplicações possuem fases que são difíceis de prever. Isso ocorre, por exemplo, quando a aplicação alterna entre períodos de alto e baixo uso de NEON rapidamente, fazendo o árbitro ligar e desligar o despacho de instruções várias vezes em pouco tempo, causando atraso na execução da aplicação. Para resolver esse problema, nós propomos o árbitro Performance, que estende o comportamento do Basic com algumas modificações. O Performance monitora as últimas predições feitas e, ao detectar que houve um número de mudanças de estado maior que um limiar predefinido, desliga o despacho de instruções por um longo período. Dessa forma, é possível executar aplicações com fases difíceis de prever nas unidades NEON do big.

C.2.3 Núcleos parciais com ISA completo

Como as instruções NEON tendem a ser pouco utilizadas em alguns tipos de aplicação, nós propomos utilizar o despacho de instruções desacoplado para possibilitar *núcleos parciais de ISA completo*. Os núcleos parciais são núcleos big sem as caras unidades NEON, como mostrado na Figura C.6. O suporte às instruções NEON é mantido por meio do despacho desacoplado. Dessa forma, os núcleos mantêm compatibilidade total com

Figure C.7: Núcleos parciais necessitam de menos áreas que núcleos completos. A área pode ser utilizada para incorporar aceleradores ou núcleos little ao sistema.



Source: The author.

o ISA, sendo as diferenças apenas em nível de microarquitetura. Núcleos parciais são equivalentes ao big em no processamento de instruções de inteiro, mas requerem menos área. Conforme ilustrado na Figura C.7 com a área liberada é possível a adição de mais processadores little ou a inclusão de um acelerador.

C.2.4 Resultados e conclusão

Nós avaliamos o despacho de instruções em um cenário ideal, isto é, considerando o big realizando o despacho de instruções para um núcleo little inativo. Dessa forma, foi possível avaliar os ganhos máximos do offloader para os dois árbitros propostos: Basic e Performance. Nós concluímos que ambas configurações de árbitros foram capazes de prover ganhos de energia com baixo impacto na execução das aplicações em comparação com o A15, processador big presente no ARM big.LITTLE. Contudo, o árbitro Performance obteve melhores resultados em aplicações com fases difíceis de prever, como no caso do benchmark *basicmath*.

Além disso, nós avaliamos os núcleos parciais com ISA completo considerando o ARM big.LITTLE (4 bigs + 4 littles) como referência. Nós consideramos os cenários 1f3p, 2f2p, 3f1p e 4f0p, onde "f" representa um núcleo *full* (completo) e "p" representa *partial* (parcial). Nós consideramos que os núcleos full são equipados com o despacho de instruções e o árbitro Performance. Portanto, o cenário 4f0p é equivalente ao big.LITTLE mas é capaz de fazer o despacho de instruções. Embora a adição de núcleos parciais consiga reduzir os custos de área do multi-core, ela deve ser realizada tendo em mente as aplicações executadas, pois se houverem mais aplicações com alto uso de NEON executando simultaneamente do que o número de núcleos full disponíveis, elas competiram por estes núcleos. Dessa forma, atrasando a execução das aplicações e aumentando os custos de energia.