

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

JECKSON DELLAGOSTIN SOUZA

**Applying Partial Instruction Set
Architectures and Instruction Offloading to
Enhance Asymmetric Multicores**

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

Advisor: Antonio Carlos Schneider Beck

Porto Alegre
May 2020

CIP — CATALOGING-IN-PUBLICATION

Dellagostin Souza, Jeckson

Applying Partial Instruction Set Architectures and Instruction Offloading to Enhance Asymmetric Multicores / Jeckson Dellagostin Souza. – Porto Alegre: PPGC da UFRGS, 2020.

181 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Antonio Carlos Schneider Beck.

1. Heterogeneity. 2. Partial isa. 3. Overlapping isa. 4. Energy efficiency. 5. Instruction offloading. 6. Shared resources. 7. Shared execution unit. I. Schneider Beck, Antonio Carlos. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Rui Vicente Oppermann

Vice-Reitora: Prof^a. Jane Fraga Tutikian

Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves

Diretora do Instituto de Informática: Prof^a. Carla Maria Dal Sasso Freitas

Coordenadora do PPGC: Prof^a. Luciana Salete Buriol

Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

*“The most difficult thing is the decision to act,
the rest is merely tenacity.”*

— AMELIA EARHART

AGRADECIMENTOS

Este trabalho é para o meu pai, que não pode ver sua conclusão, mas participou de todo o seu desenvolvimento. Muito obrigado pelo apoio, incentivo, carinho e alegria que tu me deu por todos esses anos. Este trabalho também é para minha mãe, que sempre esteve ao meu lado e que nunca duvidou que eu chegaria aqui. Muito obrigado por nunca ter questionado minhas decisões e por ser um modelo de garra que eu pude seguir.

Este trabalho também é para minha esposa, que por muitas vezes acreditou mais na conclusão dele do que eu mesmo. Ela é minha inspiração de força e persistência. Muito obrigado por dividir esse caminho comigo e por toda a ajuda que tu sempre me deu.

Este trabalho também é para meu sobrinho e afilhado. Tenho esperança que no futuro em que ele vai viver, a ciência e o conhecimento que essa tese representam sejam mais valorizados do que hoje.

Todo o apoio que eu recebi nesses anos de formação acadêmica foram essenciais para o desenvolvimento desta tese e a maior fonte desse apoio sempre foi minha família.

Agradeço a todos os demais que estiveram juntos comigo durante esse processo. Meus amigos, família, colegas de laboratório e orientador. Vocês todos tiveram um papel essencial nessa tese, seja colaborando com meu crescimento pessoal ou profissional.

Por fim, também agradeço as agências financiadoras de pesquisa do Brasil e Rio Grande do Sul, incluindo a CAPES, o CNPQ e a FAPERGS. Sem elas, não haveria pesquisa no Brasil e tenho esperança de que elas serão cada vez mais fortalecidas e bem representadas pelos excelentes pesquisadores do nosso país.

ABSTRACT

For generations, General-Purpose Processors (GPPs) have implemented specialized instructions in the form of Instruction Set Architecture (ISA) extensions aiming to increase the performance of emerging applications. Nonetheless, these extensions impose a significant overhead in the area and power of the processor, as they require custom components to implement the specialized datapaths. Examples are the Single Instruction Multiple Data (SIMD) and Floating-Point (FP) instructions, which pipelines can represent more than half of the total area of the core. Exploiting the fact that these instructions are not used as often as the instructions from the base ISA, we propose solutions to reduce the amount of support for instructions extensions in Asymmetric Multicores (AMCs) (multicores that usually implement cores of high performance - big cores - and high energy/area efficiency - small cores), enhancing their area and energy efficiency. This thesis proposes two complementary methods that can be combined to achieve such efficiency. We start by introducing the Partially Heterogeneous ISA (PHISA) multicore. PHISA is composed of heterogeneous cores of single base ISA, but asymmetric functionality. In other words, some of the cores in the multicore system do not fully implement the costly instruction extensions, but all share a mutual base ISA. Therefore, by replacing full-ISA by partial-ISA cores and migrating tasks when necessary, it is possible to free valuable area and power from the processor design, while maintaining support for the extended instructions. While migrating jobs can be efficient in single-threaded workloads, this might not be the case in parallel applications. Migrations can increase the time a thread requires to reach a synchronization point, introducing a bottleneck in the execution. For these applications, we propose to increment PHISA with the **Tightly Coupled Instruction Offloader (TUNE)** component. The TUNE architecture implements a PHISA system in which the big core is a partial-ISA and responsible for executing only the serial regions of the applications. The small cores, on the other hand, are all full-ISA and responsible for the parallel regions. Whenever the serial region requires to execute a non-implemented instruction, TUNE offloads this operation to the small cores in a transparent manner. In this thesis, we show how PHISA and TUNE can be used to improve performance and energy consumption in both serial and parallel applications, compared to other traditional heterogeneous designs.

Keywords: Heterogeneity. partial isa. overlapping isa. energy efficiency. instruction offloading. shared resources. shared execution unit.

Da Aplicação de Conjuntos Parciais de Instruções e Despacho Externo de Instruções para Aumentar a Eficiência de Processadores Multi-Núcleo Assimétricos

RESUMO

Por gerações, os processadores de propósito geral implementam instruções especializadas na forma de extensões de conjuntos de instruções ISA com o objetivo de aumentar o desempenho de aplicações emergentes. Contudo, tais extensões impõem um custo significativo na área e potência do processador. Um exemplo está nas instruções do tipo instrução única, múltiplos dados SIMD e de ponto flutuante FP, cujas *pipelines* podem representar mais da metade da área total de um núcleo do processador. Aproveitando o fato de que tais instruções não são tão comumente usadas como as da ISA base, são propostas soluções para reduzir a quantidade de suporte que é dado a extensões de instruções em processadores multinúcleo assimétricos AMC (sistemas que usualmente implementam núcleos de alto desempenho - núcleos grandes - e alta eficiência de área/energia - núcleos pequenos), aprimorando sua eficiência em área e energia. Inicialmente, é introduzido o sistema multinúcleo de ISA parcialmente heterogênea PHISA. PHISA é composto de núcleos heterogêneos com uma única ISA base, mas funcionalidades diferentes. Em outras palavras, alguns dos núcleos deste sistema heterogêneo não implementam completamente as caras extensões de ISA, mas ainda assim todos compartilham uma ISA base mútua. Desta forma, ao substituir núcleos de ISA completa por núcleos de ISA parcial e migrando tarefas sempre que necessário, é possível liberar recursos valiosos de área e potência do projeto do processador, sem abrir mão completamente do suporte as extensões de ISA. Por outro lado, enquanto a migração de tarefas é eficiente em aplicações de *thread* única, este pode não ser o caso em aplicações paralelas. Migrações podem aumentar o tempo em que uma das múltiplas *threads* precisa para atingir seus pontos de sincronização, criando assim um gargalo em sua execução. Para tais aplicações, é proposto aprimorar o sistema PHISA com um despachador de instruções fortemente acoplado TUNE. A arquitetura com TUNE implementa um sistema PHISA cujo núcleo grande implementa parcialmente a ISA e é responsável pela execução das regiões seriais das aplicações. Os núcleos pequenos, por outro lado, implementam toda a ISA completa do sistema e são responsáveis pela execução das regiões paralelas da aplicação. Sempre que a região sequencial da aplicação precisar executar uma instrução não implementada no núcleo grande, o TUNE irá despachar estas operações para os núcleos pequenos de forma transparente. Nesta tese, é

mostrado como o PHISA e TUNE podem ser usados para melhorar o desempenho e consumo energético ambos de aplicações seriais e paralelas, quando comparado a projetos tradicionais de AMCs.

Palavras-chave: heterogeneidade, isa parcial, isa sobreposta, eficiência energética, despacho externo de instruções, recursos compartilhados, unidade de execução compartilhada.

LIST OF ABBREVIATIONS AND ACRONYMS

AI Artificial Intelligence.

AMC Asymmetric Multicore.

AVX Advanced Vector Extensions.

CISC Complex Instruction Set Computer.

COTS Commercial Off-The-Shelf.

DSE Design Space Exploration.

DSP Digital Signal Processing.

EDP Energy-Delay Product.

FIFO First In First Out.

FP Floating-Point.

FPU Floating-Point Unit.

FS Full System.

GPP General-Purpose Processor.

HP-C Hardware Performance-Counter.

HPC High-Performance Computing.

ILP Instruction-Level Parallelism.

IPC Instruction per Cycle.

ISA Instruction Set Architecture.

LLC Last Level Cache.

MAC Multiply-Accumulate.

MPSoC Multi-Processor System-on-Chip.

NN Neural Network.

OoO Out-Of-Order.

OS Operating System.

PHISA Partially Heterogeneous ISA.

RF Register File.

RISC Reduced Instruction Set Computer.

SE System-call Emulation.

SIMD Single Instruction Multiple Data.

SOA State-of-the-art.

SSE Streaming SIMD Extensions.

SVE Scalable Vector Extension.

TDP Thermal Design Power.

TLP Thread-Level Parallelism.

TMR Triple Modular Redundancy.

TP Task Parallelism.

TUNE **T**ightly **C**oupled **I**nstruction **O**ffloader.

LIST OF FIGURES

Figure 1.1 Instruction breakdown by category. Highlighted instructions are from NEON extension	18
Figure 1.2 Area and Power breakdown by processor component in an ARM A15 accordingly to McPAT.....	20
Figure 1.3 (1)Traditional A15 core. (2)A15 core without NEON instructions (partial-ISA). (3)Partial-ISA A15 core + 2 full-ISA A7 cores - TDP budget. (4) Partial-ISA A15 core + 4 full-ISA A7 cores - Area budget.....	22
Figure 1.4 TUNEd PHISA system with SIMD/FP instruction offloading.....	23
Figure 2.1 High level view of the scheduling prediction scheme in (ANNAMALAI et al., 2013).	28
Figure 2.2 Coupling Binary Translation with a Reconfigurable Accelerator.....	28
Figure 2.3 big.LITTLE cluster organization.....	30
Figure 2.4 big.LITTLE scheduling modes.....	31
Figure 2.5 DynamIQ cluster organization.....	32
Figure 2.6 Structure of a vector unit containing four lanes.....	34
Figure 2.7 ISA affinity for different applications on designs optimized for (bars from left to right) - (bar 1) Single-thread performance, (bar 2) Multi-programmed workload performance, (bar 3) Single-threaded workload EDP, (bar 4) Multi-programmed workload EDP.....	35
Figure 2.8 ISA growth (in number of instructions) over time for different processor architectures.	36
Figure 2.9 Histogram (in logarithmic scale) of dynamic instructions sorted by frequency with respect in Windows 95 and compared to their corresponding frequency in Windows 7. Spikes show differences in the usage pattern.....	37
Figure 2.10 Partial ISA processors implement a subset of the full ISA.	37
Figure 2.11 ISA extensions usage and execution time.....	42
Figure 2.12 A heterogeneous system with a reduced-ISA core.....	43
Figure 2.13 Cojoined-cores sharing resources such as cache memory and FP units.....	45
Figure 2.14 AMD Bulldozer microarchitecture.....	46
Figure 2.15 Sun Niagara (Ultrasparc T1) microarchitecture.	47
Figure 2.16 Area breakdown by processor component in an ARM A15 accordingly to McPAT.....	48
Figure 2.17 Dual core A9 Floorplan. Highlighted are the areas of one entire A9 CPU and the NEON unit components.	48
Figure 3.1 Example of PHISA configuration. The resources freed by an instruction extension are used to increase the core count.	54
Figure 3.2 A typical Out-Of-Order (OoO) processor datapath. In green, the parts of the datapath that can be trimmed or simplified when removing the floating-point support from the processor.....	55
Figure 3.3 Scheduler events between full and partial ISA cores.	57
Figure 3.4 Cortex A15 Pipeline organization.....	63
Figure 3.5 Area breakdown by processor component.....	63
Figure 3.6 PHISA system using A15 and A7 cores examples.....	64
Figure 4.1 Typical parallel application execution.	65
Figure 4.2 TUNEd PHISA system with SIMD/FP instruction offloading.....	67

Figure 4.3 NEON instruction offloading from a A15 core to A7 cores. The NEON units from A7 cores are shared with the A15.	69
Figure 4.4 The vector registers from the A15 core lanes are split into shorter registers and distributed over the A7 cores, along with the operation.....	69
Figure 4.5 Potential speedup according to TUNE mechanism model. Bars represent the speedup over a single A15 core and the shaded areas are the % of <i>PR</i> and <i>SRF</i> of each application.....	74
Figure 5.1 The simulation tool-chain used in this work.....	80
Figure 5.2 Representation of the execution traces of a workload in both a big and a little core, generated with the gem5 simulator. The current host core of the workload determines which of the traces will be consumed for a given slice of the execution.....	82
Figure 5.3 A high-level view of the PHISA scheduler simulator. On the top, the set of inputs necessary for execution. Above, the different modules of the simulator and their interaction with each other and with the inputs.....	85
Figure 6.1 Example of a PHISA configuration.	88
Figure 6.2 A15 cores with progressive replacement of full with partial ISA cores. Performance, Energy and EDP are normalized to the A15(4F0P) configuration ...	94
Figure 6.3 Evaluation PHISA multicores against a single-core baseline under a 700mW power budget. Performance, Energy and EDP are normalized to the A15(1F0P) configuration	96
Figure 6.4 Core usage in configuration A15(0F1P)A7(2F0P) running scenario 6. High step means the core is in usage, low step is idle. Solid bars are constant idle-active changes. Dots represent migrations.	96
Figure 6.5 Evaluation of PHISA multicores against a DynamIQ baseline. The baseline has the same amount of cores as the PHISA configurations, thus there is no power/area budget. Performance, Energy and EDP are normalized to the A15(1F0P)A7(2F0P) configuration.	98
Figure 6.6 Evaluation of PHISA multicores allowing emulation against a DynamIQ baseline. The baseline has the same amount of cores as the PHISA configurations, thus there is no power/area budget. Performance, Energy and EDP are normalized to the A15(1F0P)A7(2F0P) configuration	100
Figure 6.7 Evaluation of PHISA multicores with and without emulation against a DynamIQ baseline under a 800mW power budget. Performance, Energy and EDP are normalized to the A15(1F0P)A7(2F0P) configuration.....	101
Figure 6.8 Behaviour on high NEON usage PHISA multicore (with and without emulation) and DynamIQ.	102
Figure 6.9 Evaluation PHISA multicores against a single-core baseline under a 700mW power budget. Scheduling of tasks follows a performance optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the A15(1F0P) configuration.....	104
Figure 6.10 Evaluation of PHISA multicores against a DynamIQ baseline under a 800mW power budget. Scheduling of tasks follows a performance optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the A15(1F0P)A7(2F0P) configuration.	104
Figure 6.11 Scheduler migrations using the performance policy in the traditional DynamIQ configuration A15(1F0P)A7(2F0P) during execution of scenario 1. Bars represent each application being run over time in each processor core.	105

Figure 6.12 Scheduler migrations using the performance policy in the PHISA configuration A15(0F2P)A7(4F0P) during execution of scenario 1. Bars represent each application being run over time in each processor core.	106
Figure 6.13 Evaluation PHISA multicores against a single-core baseline under a 700mW power budget. Scheduling of tasks follows a energy consumption optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the A15(1F0P) configuration.	107
Figure 6.14 Evaluation of PHISA multicores against a DynamIQ baseline under a 800mW power budget. Scheduling of tasks follows a energy consumption optimization policy for all configurations, including the baseline Performance, Energy and EDP are normalized to the A15(1F0P)A7(2F0P) configuration.	107
Figure 6.15 Scheduler migrations using the energy policy in the traditional DynamIQ configuration A15(1F0P)A7(2F0P) during execution of scenario 1. Bars represent each application being run over time in each processor core.	108
Figure 6.16 Scheduler migrations using the energy policy in the PHISA configuration A15(0F2P)A7(4F0P) during execution of scenario 1. Bars represent each application being run over time in each processor core.	108
Figure 6.17 Example of a TUNEd PHISA configuration.	109
Figure 6.18 Speedup according to the developed system simulation. Bars are the speedup over a single A15 core and areas are the % of <i>PR</i> and <i>SRF</i> of each application.	113
Figure 6.19 Energy savings for the traditional system and TUNE normalized by the energy the single A15 core. Bars below 1 means that the energy consumption was lower than the baseline.	115
Figure 6.20 Evaluation of PHISA multicores and state-of-the-art (LEE et al., 2017) against a big.LITTLE baseline under a $4.7mm^2$ area budget. Scheduling of tasks follows a performance optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the big.LITTLE configuration.	117
Figure 6.21 Evaluation of PHISA multicores and state-of-the-art (LEE et al., 2017) against a big.LITTLE baseline under a $4.7mm^2$ area budget. Scheduling of tasks follows a energy consumption optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the big.LITTLE configuration.	118
Figure 6.22 Speedup for the state-of-the-art Niagara system and TUNE normalized. Bars are the speedup over a single A15 core and areas are the % of <i>PR</i> and <i>SRF</i> of each application.	120
Figure 6.23 Energy savings for the state-of-the-art Niagara system and TUNE normalized by the energy the single A15 core. Bars below 1 means that the energy consumption was lower than the baseline.	121
Figure C.1 Divisão da área e potência por componente de um processador ARM A15 de acordo com o McPAT.	176
Figure C.2 Divisão do tipo de instruções em diversas aplicações <i>single-thread</i> . São destacadas as instruções de extensões NEON.	177
Figure C.3 (1)Núcleo A15 tradicional. (2)Núcleo A15 sem instruções NEON instructions (ISA parcial). (3)Núcleo A15 de ISA parcial + 2 núcleos A7 de ISA completa - mesma potência máxima (TDP). (4) Núcleo A15 de ISA parcial + 4 núcleos A7 de ISA completa - mesma área.	177
Figure C.4 TUNEd PHISA com despachador de instruções SIMD/FP.	179

LIST OF TABLES

Table 1.1 Application region of interest characterization in terms of parallel region size and SIMD/FP ratio in the serial region.	19
Table 2.1 Examples of extensions from the x86, ARM, Power and RISC-V architectures.	33
Table 2.2 Areas of full (including L1 caches) A7, A9, and A15 cores and their NEON units. Total and NEON unit areas from the A7 and A15 were reported by McPAT. A9 total area is reported from (KOPPANALIL et al., 2011) and its NEON unit area is estimated from the layout. The NEON unit areas in this table do not include potential areas from wiring, routing, decoding, and other components related to this unit.	49
Table 2.3 Characteristics of the relevant works compared against the PHISA and TUNEd PHISA systems. N/A means that the given characteristic was not applicable to the given study.	53
Table 4.1 Application region of interest characterization in terms of parallel region size and SIMD/FP ratio in the serial region.	73
Table 4.2 Model variation heatmap for the TUNE AMC (1 A15 without NEON + 8 full A7) with a fixed offloading cost of 2x normalized by a full A15 single core. In the rows the ratio of the parallel region increases by 10%, while the columns increase the percentage of SIMD and FP operations in the serial region.	75
Table 4.3 Model variation heatmap for a manycore system with 16 full A7 cores, normalized by a full A15 single core. In the rows the ratio of the parallel region increases by 10%, while the columns increase the percentage of SIMD and FP operations in the serial region (no effect in this scenario).	76
Table 4.4 Model variation heatmap for the TUNE AMC (1 A15 without NEON + 8 full A7) with a fixed parallel region ratio of 80% normalized by a full A15 single core. In the rows, cost of offloading instructions increases by a power of 2, while the columns increase the percentage of SIMD and FP operations in the serial region.	76
Table 6.1 Workloads in each scenario.	89
Table 6.2 Multicore configurations. *PHISA core using emulation.	91
Table 6.3 The experiments and their goals.	92
Table 6.4 Area and power of full and partial A15 and A7 processors.	94
Table 6.5 Application region of interest characterization in terms of parallel region size and SIMD/FP ratio in the serial region.	111
Table 6.6 Simulated application characteristics for Polybench applications. Columns show, in order: How many times a single A15 is faster than a single A7 (A7 slowdown); The parallel speedup of 8 A7 cores over a single A7; The overhead caused by offloading instructions in the TUNEd PHISA system.	112
Table 6.7 PHISA vs State-of-the-art configurations.	116
Table 6.8 TUNEd PHISA vs Niagara configurations.	119
Table B.1 Setup 1.	160
Table B.2 Setup 2.	161
Table B.3 Setup 3.	162
Table B.4 Setup 4.	163
Table B.5 Setup 5.	164

Table B.6 Performance Policy - Single A15	165
Table B.7 Performance Policy - big.LITTLE.....	166
Table B.8 Energy Policy - Single A15	167
Table B.9 Energy Policy - big.LITTLE.....	168
Table B.10 PHISA vs SOA - Performance Policy	169
Table B.11 PHISA vs SOA - Energy Policy	170
Table B.12 Performance	172
Table B.13 Energy	173
Table B.14 Performance vs Niagara-like	174
Table B.15 Energy vs Niagara-like	175
Table C.1 Caracterização das regiões de interesse de aplicações em termos do tamanho da região paralela e da quantidade de instruções SIMD/FP na região serial.	178

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS.....	8
1 INTRODUCTION.....	17
1.1 Hypothesis.....	20
1.1.1 PHISA: a solution for single-threaded workloads	21
1.1.2 TUNEd PHISA: a solution for multi-threaded workloads.....	22
1.2 Contributions of this Thesis	24
1.3 Structure of this Thesis.....	24
2 BACKGROUND.....	25
2.1 Single-ISA Heterogeneous Processors.....	25
2.1.1 The Hardware Approach.....	26
2.1.2 The Scheduler Approach.....	28
2.1.3 Industry Implementations	30
2.2 The Impact of the ISA on the Processor	32
2.3 Overlapping- and Partial-ISA	36
2.3.1 Operating System Support	38
2.3.2 Schedulers	39
2.3.3 Microarchitecture Support	40
2.3.4 Partial-ISA Implementation	41
2.4 Sharing Resources Between Cores	44
2.5 On the Area of SIMD/FP Units.....	47
2.6 Contributions Over the State-of-the-art	49
2.6.1 Contributions over single-ISA heterogeneous processors	50
2.6.2 Contributions over the impact of the ISA	50
2.6.3 Contributions over overlapping- and partial-ISAs.....	51
2.6.4 Contributions over sharing resources.....	51
2.6.5 Wrapping up.....	52
3 PHISA MULTICORE.....	54
3.1 The PHISA System	54
3.2 Scheduling.....	55
3.2.1 Minimalist Scheduler	56
3.2.2 Minimalist Scheduler with emulation.....	58
3.2.3 Scheduling Policies.....	58
3.3 PHISA in a COTS Processor.....	62
4 TUNE ARCHITECTURE.....	65
4.1 The Offloader	65
4.2 TUNE in a COTS processor	68
4.3 TUNE Models.....	70
4.3.1 Performance Model.....	70
4.3.2 Application Characterization	71
4.3.3 Discussion	72
5 SIMULATION METHODOLOGY.....	78
5.1 Gem5	78
5.2 McPAT.....	79
5.3 PHISA Simulator	80
5.3.1 Profiling and tracing workloads execution phases	80
5.3.2 Modeling area and power using McPAT	83
5.3.3 Modeling the <i>PHISA Simulator</i> for a multi-task simulation.....	84

6 EVALUATION.....	88
6.1 Evaluation of PHISA with single-threaded workloads.....	88
6.1.1 Evaluation Methodology.....	89
6.1.2 Results.....	93
6.1.2.1 Impact of Partial ISA Cores.....	93
6.1.2.2 Full Core vs PHISA Multicore - Sharing a Power Budget.....	95
6.1.2.3 PHISA vs Traditional Heterogeneous Systems (DynamIQ).....	97
6.1.2.4 Task Migration or NEON Emulation.....	98
6.1.2.5 The impact of emulation.....	99
6.1.2.6 PHISA with emulation vs DynamIQ - Power Parity.....	99
6.1.3 Analisis of PHISA on High NEON Usage.....	101
6.1.4 Scheduling Policies Impact.....	103
6.1.4.1 Scheduling for Performance.....	103
6.1.4.2 Scheduling for Energy.....	106
6.1.5 Summarizing the results.....	107
6.2 Evaluation of TUNEd PHISA with multi-threaded workloads.....	109
6.2.1 Evaluation Methodology.....	109
6.2.2 Performance results.....	110
6.2.3 Energy consumption results.....	114
6.2.4 Summarizing the results.....	115
6.3 PHISA and TUNE vs the State-of-the-art.....	116
6.3.1 PHISA vs State-of-the-art.....	116
6.3.2 TUNEd PHISA vs State-of-the-art.....	119
6.3.3 Summarizing the results.....	120
7 CONCLUSIONS.....	122
7.1 On PHISA - binary support through migration.....	122
7.2 On TUNEd PHISA - binary support through offloading.....	124
7.3 Limitations.....	125
7.4 Open Challenges.....	125
8 PUBLICATIONS.....	128
REFERENCES.....	131
APPENDICES.....	139
APPENDIXA.....	140
A.1 Processor configuration files from gem5.....	140
A.1.1 A15 core.....	140
A.1.2 A7 core.....	145
A.1.3 A15 core with extra SIMD/FP latency (for TUNE).....	149
A.1.4 A7 core with extra SIMD/FP latency (for Niagara).....	154
APPENDIXB.....	159
B.1 PHISA Multicores raw results values.....	159
B.2 TUNEd PHISA raw results values.....	171
APPENDIXC - RESUMO EM PORTUGUÊS.....	176
C.1 Introdução.....	176
C.2 Arquiteturas desenvolvidas.....	179
C.2.1 PHISA Multi Núcleos.....	179
C.2.2 TUNEd PHISA.....	180
C.3 Metodologia.....	181
C.4 Resumo dos resultados.....	181

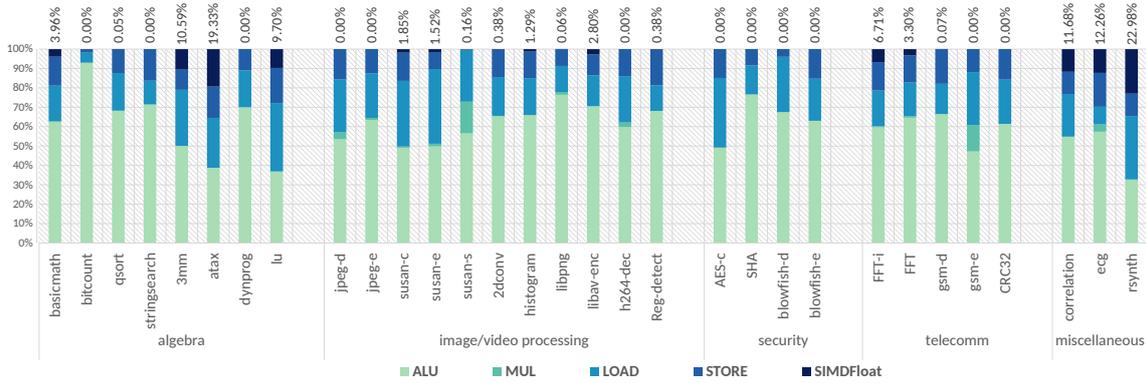
1 INTRODUCTION

General-Purpose Processors (GPPs) have become highly complex, adopting many performance strategies such as Out-Of-Order (OoO) execution, multithreaded processing, and application-specific instruction sets. Such complex designs, however, are usually limited by a power constraint, independently from the system application - be it a server of High-Performance Computers (HPC) or a simple embedded system. To deliver high-performance throughput at small energy budgets, processor manufacturers are currently relying on single-Instruction Set Architecture (ISA) heterogeneous processors, such as the ARM big.LITTLE(ARM, 2016), and - more recently - ARM DynamIQ(ARM, 2018). These processor's designs employ distinct cores with different microarchitectural properties - both in performance and power - in the same die. While the big core is usually a complex out-of-order core, the little is normally implemented as a simple in-order core.

These processors designs are an efficient strategy both in the perspective of the area and power usage, and in product deployment, as manufacturers can use readily available core designs to create them. This reusability is a common approach in the industry, as manufacturers tend to update their products incrementally and to use existing and validated tools from previous processor generations. Therefore, to boost the performance of emerging applications, instead of completely changing the microarchitectural organization, designers of GPPs generally rely on tailoring the processor ISA. Each architectural iteration of GPPs adds newer instructions in the form of extensions (e.g., Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) in the x86, and Thumb, NEON and Scalable Vector Extension (SVE) in the ARM), increasing the complexity of the microarchitecture. Nonetheless, not all applications will take advantage of such instructions. For instance, x86 AVX Single Instruction Multiple Data (SIMD) instructions are specifically used for highly vectorized applications and ARM's SVE is targeted for High-Performance Computing (HPC).

This is no different for NEON instructions (Floating-Point (FP) and SIMD operations extension) in ARM architectures, as can be seen in Figure 1.1. This Figure shows the percentage of dynamic instructions executed in several single-threaded workloads from different benchmark suits in an ARM processor. It demonstrates how underused NEON instructions (categorized as SIMDFloat in the figure) are. While some applications do present a considerable amount of SIMD and FP operations, such as the voice synthesizer *rsynth* and the vector multiplier *atax*, most of the others use few or non of these instruc-

Figure 1.1: Instruction breakdown by category. Highlighted instructions are from NEON extension



Source: The Author

tions.

Furthermore, there are specific phases in applications that tend to use more of these operations. For instance, table 1.1 shows several parallel applications, the size of their parallel regions (as a ratio of the time spent executing in parallel vs serial execution) and the amount of FP and SIMD operations executed only in the serial region (as a ratio against the number of all other instructions). In this table, we can observe that even in applications with larger serial regions, the amount of SIMD and FP operations in these phases is usually small.

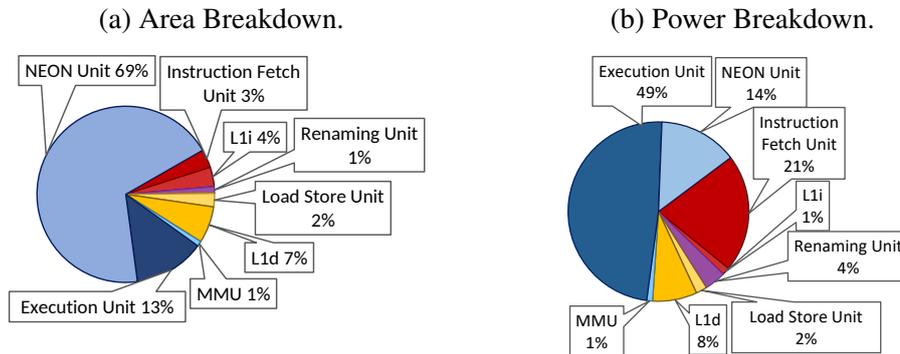
When one considers the current Commercial Off-The-Shelf (COTS) embedded processors, the trend of incrementing the ISA can only worsen the situation presented in our observation, in which specific operations are not used in most applications. This is further aggravated when we analyze the impact that these extensions can introduce in current embedded processor designs. Figure 1.2 shows the area (1.2a) and power (1.2b) breakdown of the ARM A15 processor, extracted from McPAT models (LI et al., 2009). The A15 has two NEON pipelines with ten stages and a wide FP-specific register file that represent about 69% of the total core area. Furthermore, these components also comprise about 14% of the dynamic peak power of the A15 processor. Note that this area and power analysis **includes the L1 instruction and data caches** of the core, which makes the impact of such units even more evident. Even when we consider processors of other architectures, such as the RISC-V, that implement only simple Floating-Point Units (FPUs) (without SIMD support), the area of this unit can represent 37% of the total core area (BECKER; SOUZA; BECK, 2019).

It is understandable why such operations are included in current processors. These instructions can provide huge performance improvements and are much more power effi-

Table 1.1: Application **region of interest** characterization in terms of parallel region size and SIMD/FP ratio in the serial region.

	Benchmark	Parallel Ratio	Serial SIMD/FP Ratio
parsec	bodytrack	99.10%	0.05%
	ferret	98.24%	0.10%
	dedup	98.18%	0.00%
	facesim	97.56%	0.14%
	cholesky	96.56%	0.88%
	freqmine	95.38%	0.01%
	parvec	swaptions	99.99%
fluidanimate		99.67%	0.05%
streamcluster		99.60%	0.01%
canneal		99.58%	0.02%
blackscholes		99.55%	0.01%
vips		98.94%	0.14%
polybench		bicg	100.00%
	fdtd-apml	99.99%	0.00%
	convolution-2d	99.99%	0.00%
	gemm	99.97%	0.06%
	symm	99.95%	0.00%
	syrk	99.90%	0.04%
	syr2k	99.89%	0.04%
	atax	99.73%	0.06%
	2mm	99.27%	1.03%
	mvt	98.82%	1.11%
	gesummv	98.76%	0.46%
	3mm	98.68%	1.66%
	doitgen	98.41%	0.83%
	trmm	82.25%	6.74%
	correlation	81.95%	10.10%
	gramschmidt	78.70%	11.72%
	covariance	77.40%	17.33%
lu	68.71%	0.09%	
splash2x	ocean_ncp	99.85%	0.01%
	barnes	99.74%	0.02%
	ocean_cp	99.63%	0.04%
	radix	99.37%	0.00%
	raytrace	99.19%	0.12%
	lu_cb	98.48%	0.11%
	water_nsquared	97.92%	0.20%
	lu_ncb	97.86%	0.15%
	radiosity	97.81%	0.22%
	fft	97.38%	0.21%
	water_spatial	94.12%	0.66%
	cholesky	75.95%	2.77%

Figure 1.2: Area and Power breakdown by processor component in an ARM A15 accordingly to McPAT.



Source: The Author

cient than software emulation (FP operations can be emulated using the standard integer functional units, while SIMD can be serialized). In (LEE et al., 2017) the authors show that some applications can experience performance drops of up to 23x when NEON instructions are not available in hardware.

Therefore, we find ourselves in a scenario in which application-specific instructions are necessary to maintain non-functional requirements (such as performance). On the other hand, **these instructions are too expansive to implement in terms of area and power**, especially in systems that must be as efficient as possible. When this scenario is considered in an ever-growing ISA future with processors of manycores, it can become a major issue. Thus, we propose a reevaluation in the way these components are implemented in a processor.

1.1 Hypothesis

From the previously described observations of applications behavior and the current scenario in designs of embedded multicore GPPs, we formulate the following hypothesis:

Hypothesis. *Current processors implement non-essential components, designed exclusively for application-specific instructions, which are rarely used but are still needed to keep non-functional requirements. Such components introduce a high area and power overhead, which is multiplied in a multicore processor. By exchanging some of these components for generic in-order cores and applying thread migration, instruction emulation, and offloading, it is possible to improve the non-functional requirements of the system, while keeping the processor's binary compatibility.*

In other words, our hypothesis is that if we need to implement instruction extensions to maintain processor performance, there must be more efficient designs than simply replicating every component in every core of the processor. Although this hypothesis resembles a scheduling problem, our objective is not to provide an accelerator rich processor in which tasks are balanced according to their needs. Our goal is to demonstrate how GPPs based on current COTS can be efficiently designed to further optimize non-functional requirements of energy consumption and performance. For that goal, we develop techniques that include migrating tasks and offloading instructions between GPPs, which aim to use the most of the processor resources.

1.1.1 Partially Heterogeneous ISA (PHISA): a solution for single-threaded workloads

In this thesis, we introduce the PHISA multicore, a processor in which parts of the ISA are removed from some cores, while the remaining cores are fully implemented for application-specific execution. The goal is to trim the application specific components of the cores, reducing area and power but without compromising performance. Performance is kept by migrating applications from partial- to full-ISA cores. If a task running in a partial core requires a special instruction that is not implemented, it can migrate to a full core to take advantage of the accelerated instruction. Figure 1.3(1) and (2) illustrate how a partial-ISA is idealized. In this example, by removing the NEON components of the A15 core, it is possible to reduce its area (from 3.5mm^2 to 1.2mm^2) and Thermal Design Power (TDP) (from 700mW to 600mW).

Nonetheless, migrating tasks between cores can introduce high overheads in the execution, depending on the frequency of these switches. Thus, we also propose to use the extra power and area provided by PHISA cores to introduce more cores to the system that can be used to improve the non-functional requirements of the processor. Differently from the application specific components removed, extra cores can be more efficiently used by all the applications running in the system. This is illustrated in the figure 1.3 (3) and (4). Scenario (3) shows a design in which the TDP of all cores (partial-ISA A15 and full A7 combined) are the same as the TDP from the full A15 (scenario (1)). Nonetheless, as the A7 processor is much smaller than the A15, it is possible to fit more cores. This is illustrated in scenario (4), in which the freed area is almost filled, but the system TDP is

higher than the full A15 processor.¹

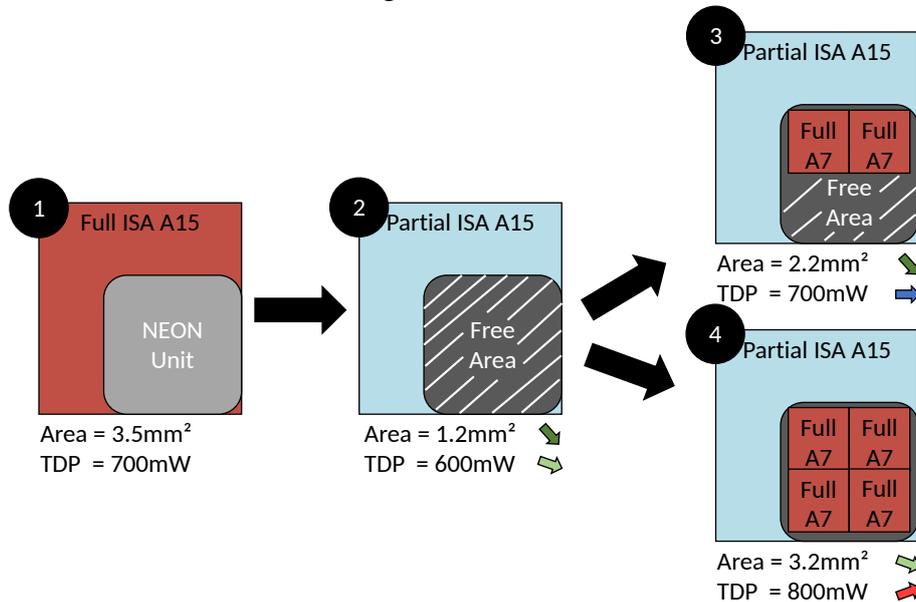
In other words, we propose that it is possible to trade high Instruction per Cycle (IPC) from a core - given by the ISA extensions - for more Task Parallelism (TP). The IPC is a metric that directly measures the performance of a core - the average number of instructions it can commit per cycle -, while the TP is related to the number of processes a processor can execute simultaneously.

1.1.2 Tightly Coupled Instruction Offloader (TUNE)d PHISA: a solution for multi-threaded workloads

This exchange of Instruction-Level Parallelism (ILP) for TP can be also very beneficial for multi-threaded applications, as the extra small cores can naturally deliver more performance to parallel regions, while serial regions can benefit from a single high-performance big core. Furthermore, we have seen in table 1.1 that the specific FP and SIMD are uncommon in the serial regions, suggesting that a processor using a partial-ISA big core could accelerate the serial regions at almost the same rate as if using a full-ISA

¹The presented designs consider the area of the L1 instruction and data caches for each core. However, we consider that the L2 cache will be shared and have the same size for all designs, independently of the number of cores.

Figure 1.3: (1)Traditional A15 core. (2)A15 core without NEON instructions (partial-ISA). (3)Partial-ISA A15 core + 2 full-ISA A7 cores - TDP budget. (4) Partial-ISA A15 core + 4 full-ISA A7 cores - Area budget.



Source: The Author

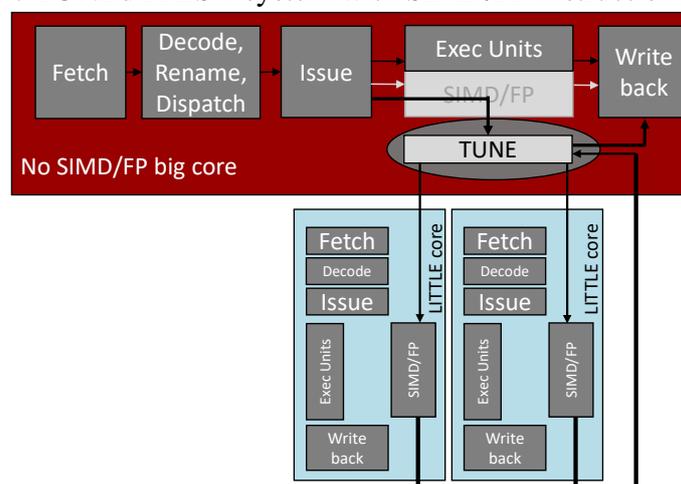
big core.

However, in parallel applications, the thread migration strategy used by PHISA might not deliver the best performance. Migrations can cause high unbalance between threads, which delay the execution of synchronization (join) points, and increase the application execution time.

In this thesis, we also develop a **Tightly Coupled Instruction Offloader (TUNE)** as a way to provide support for the removed FP and SIMD instructions in the big core used for serial regions. This design is proposed for Asymmetric Multicores (AMCs), where the big core **is an important component kept to accelerate the serial regions**. The TUNED PHISA (a PHISA processor using TUNE as offloader) uses the extra area provided by removing SIMD and FP instructions to increase the amount of small cores in the system. These extra cores are used to further increase the performance of the parallel regions of applications, while the partial-ISA big core is used to accelerate serial regions. Whenever a non supported instruction is required by the big core, TUNE transparently offloads this instruction to one (or multiple) small core(s), where they are executed and returned to the big core. The flow of the big core processor is not changed, and TUNE is seen by it as a higher latency SIMD unit. The design overview is shown in figure 1.4.

In the TUNEd PHISA system, the big core is only used when the small cores are idle (serial regions) and vice-versa (parallel regions). These guarantees that the NEON instructions from the big core will not interrupt or affect the execution in the small cores. Furthermore, as all cores are never active at the same time, the system will never go above the original TDP budget.

Figure 1.4: TUNEd PHISA system with SIMD/FP instruction offloading.



Source: The Author

1.2 Contributions of this Thesis

In summary, this thesis brings the following contributions to the state-of-the-art in efficient multicore processor designs:

- **PHISA multicores:** We develop an area and power aware design to optimize the components of a processor by providing different ISA support across the cores. In this design, we show how transparent task migration can maintain binary compatibility across the system when running single-threaded applications.
- **TUNEd PHISA:** We develop a version of PHISA that is optimized for multi-threaded applications. In a TUNEd PHISA system, transparency is kept through instruction offloading from partial- to full-cores.
- **Models and analysis:** We present scalability models for the systems and performance and energy consumption analysis over several scenarios and different environments. We also compare our solution to other published state-of-the-art and commercial designs.

1.3 Structure of this Thesis

The remaining of this work is organized as follows. Chapter 2 gives a detailed bibliographic revision on single-ISA heterogeneous processors, as well as studies on the impact of the ISA, overlapping- and partial-ISA processors proposed in previous works, and designs using shared functional units. In chapter 3, we introduce the PHISA multicore system, its particularities, scheduler policies and how it can be applied in a COTS. Chapter 4 presents TUNE, the details of the offloader, performance model and how to apply in a COTS. Chapter 5 present the simulation tools used and developed in this thesis. In chapter 6, we analyze the simulated results of both the PHISA multicores and the TUNEd PHISA system over several different scenarios and against state-of-the-art solutions. We compile all the conclusions draw by this thesis in chapter 7. Finally, chapter 8 lists all publications derived from this thesis, along with other publications from the Ph.D.

2 BACKGROUND

In this chapter, we present a state of the art review related to this thesis. We include the most recent and relevant works and industrial designs in single-ISA heterogeneous processors in section 2.1. In section 2.2 we show works that discuss the impact of the ISA in different processors. Section 2.3 brings studies that propose overlapping- or partial-ISAs, focusing on the operating system support, scheduler, and microarchitectural challenges. We further present works and designs that use shared hardware to perform different operations, transform the processor cores, and offload instructions in section 2.4. We also bring works that discuss the area instruction extensions can occupy in COTS processors in section 2.5. Finally, in section 2.6 we discuss the contributions that this thesis introduces in the current state-of-the-art.

2.1 Single-ISA Heterogeneous Processors

Single-ISA heterogeneous processors have been proposed by Kumar et al. (2003) as an alternative to power efficiency. The authors show how a mix of in-order and OoO Alpha processors with different issue widths can be used to adapt the system power usage accordingly to the application requirements. One of the main advantages of this technique is that it is transparent to the application and does not require special tools to deal with many different ISAs, as in the case of accelerator rich processors or Multi-Processor System-on-Chips (MPSoCs). On this heterogeneous environment, a runtime system manager - such as the Operating System - can identify the resource requirements of the applications and schedule threads to cores that fulfill these requirements while minimizing energy consumption. As the entire system uses the same ISA, threads can easily migrate between cores using a shared memory space, as in traditional multicore processors.

In the following subsections, we will present state-of-the-art works that have been proposed on the single-ISA heterogeneous processors subject. As we will show, these are all heavily based on the idea presented by Kumar et al. (2003), that OoO and in-order processors can be used jointly to maximize energy efficiency.

2.1.1 The Hardware Approach

Since the first proposal of Single-ISA heterogeneous processors, many authors have studied different hardware approaches to optimize the way these processors work. One of the issues of these systems is the **overhead introduced by the migration** of threads between big and small cores. To reduce this overhead works usually employ coarse-grained migrations (of about 100K instructions or more (GUTIERREZ; DRES-LINSKI; MUDGE, 2014)), which reduce the possibility of fully exploiting the potential of heterogeneous systems.

On the other hand, Lukefahr et al. (2012) have proposed an approach in which the cores implement two types of execution units, one reassembling an OoO pipeline and the other an in-order. These two units **share common resources** in the pipeline, such as the data and instruction L1 caches, the fetch unit and the branch predictor. During runtime, the system decides either to use the OoO for performance or the in-order pipeline for energy-efficiency. Accordingly to the authors, this design results in fast migrations between the two pipelines, which allows for the exploitation of fine-grained migrations (in the order of 1K instructions).

In a different approach, the authors of (PADMANABHA et al., 2015) have proposed a clever way to **reuse** the internal scheduler of Out-Of-Order (OoO) processors - which is responsible to find the dependencies of instructions and exploit ILP - in in-order cores. The approach implements an OoO core along with an in-order one. Applications first run at the OoO processor, which saves the traces of the executed instructions. These traces are then reused in the in-order processor, avoiding re-executing the dependency checks and saving energy.

The Morphcore proposed in (KHUBAIB et al., 2012), is an adaptive architecture able to exploit both the TLP of high parallel regions and the ILP of serial parts. The system is composed of a large core that can change execution modes: from single threaded out-of-order to simultaneous multithreaded in-order execution. The strategy requires no instruction or data migration between threads, and the overhead introduced is due to execution mode changes. Similarly, the authors in (IPEK et al., 2007) propose Core Fusion, an architecture in which multiple simple cores are reconfigured into one large core to adapt for the application characteristics.

In (SRINIVASAN et al., 2016), the authors propose a technique for using **morph cores** to create heterogeneity inside a core. The work performs a Design Space Explo-

ration (DSE) on many components of a core to understand which ones have bigger impacts in applications execution. Then, the authors use this data to create a core which can dynamically change its properties - such as issue and fetch width, buffers sizes, clock period and execution order - to adapt to the application at hand.

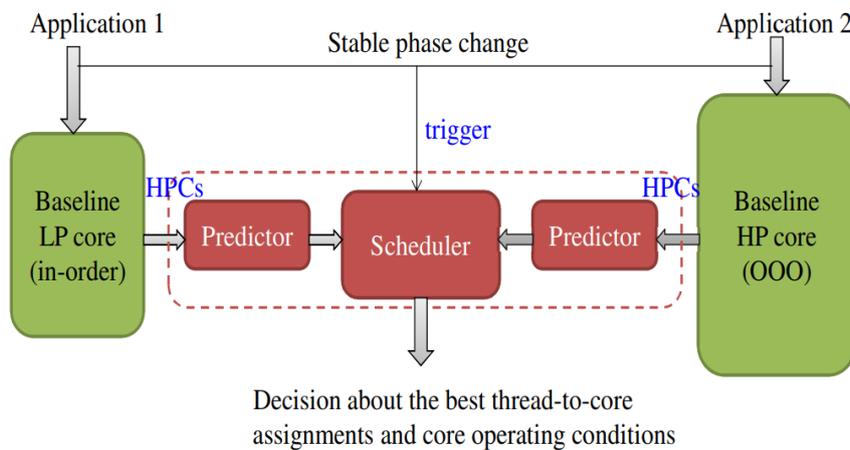
Suleman et al. (2009) argue that in a heterogeneous processor small cores are efficient in heavy parallel sections of the application, but lack the performance for executing the critical sections without creating a bottleneck. They propose a system composed of many small cores and one big core. The latter is used to accelerate the critical sections of the application, while the former execute the parallel regions. The authors also claim that the system can be easily scaled to use more big cores accordingly to requirements.

Works have also extrapolated the traditional **monotonic** (when the cores can be strictly ordered by their performance) "big and small" cores classification, using many cores that exploit different levels of ILP. These **non-monotonic** designs have led to many DSE works (NAVADA et al., 2013; MONCHIERO; CANAL; GONZALEZ, 2008; LIY et al., 2006), which aim to find fine-grained heterogeneity that can cover more performance-energy constraints in a Pareto curve.

Another common strategy to create non-monotonic cores is to use **Dynamic Voltage and Frequency Scale** (DVFS) on cores that have different microarchitectures. In (ANNAMALAI et al., 2013), Annamalai et al. propose an approach that combines DVFS and thread scheduling to increase the throughput per watt of a heterogeneous multicore system. The technique estimates the throughput/watt of an application phase at different voltage and frequency levels and maps the thread to the best matching core, along with the right level of operating voltage and frequency. Figure 2.1 show a high-level schematic of the approach, in which the Hardware Performance-Counters (HP-Cs) feeds learning data to scheduling predictors. These predictors guide the decisions of the scheduler at each system phase change based on data such as IPC and instruction and data cache miss/hit.

It is also possible to create heterogeneous microarchitectures with ISA compatibility by using **binary translators**(BECK; Lang Lisbôa; CARRO, 2013; BORIN; WU, 2009). These are components that dynamically convert binaries from a target ISA to an alternate one. The approach is used in (SOUZA et al., 2016), in which reconfigurable accelerators of different sizes - thus, of heterogeneous performance and energy characteristics - are coupled to SPARC processors along with a binary translator. The translator monitors all the code executed at runtime and converts hotspots and kernels to execute on the accelerator, as illustrated in Figure 2.2. This approach is transparent to the program-

Figure 2.1: High level view of the scheduling prediction scheme in (ANNAMALAI et al., 2013).



Source: (ANNAMALAI et al., 2013)

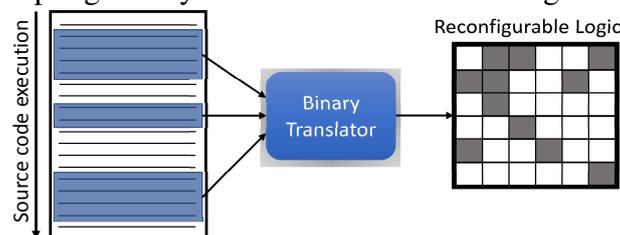
mer, as the entire application can be compiled directly to the SPARC architecture, as in a traditional single-ISA system.

Apart from providing performance heterogeneous cores, another component that is essential in single-ISA heterogeneous processors is the implemented scheduler. The scheduler is responsible for allocating tasks in the most efficient core, and if it takes wrong scheduling decisions, it can hurt the system performance. In the following subsection, we discuss different works for scheduling in heterogeneous processors.

2.1.2 The Scheduler Approach

Mittal in his survey (MITTAL, 2016) compiled many recent works on asymmetric multicore processors, including those of single-ISA. In this survey, it is noticeable that many works have focused on efficient ways to schedule tasks on heterogeneous processors, which is - in fact - one of the biggest challenges of such systems. A heterogeneous architecture cannot be fully exploited without a scheduler that migrates threads accord-

Figure 2.2: Coupling Binary Translation with a Reconfigurable Accelerator.



Source: The Author

ingly to the necessities of the applications. Furthermore, to fully exploit the heterogeneity, the scheduler cannot be static and must know how to adapt to changes in the application behavior (ANNAMALAI et al., 2013).

An efficient scheduler for heterogeneous processors must optimize the task allocation for a given metric. Generally, the scheduler is modeled to give the best performance, energy consumption, or a trade-off between these both. To migrate the tasks efficiently, these schedulers rely on models of the metrics to estimate how the different cores will behave. Sondag and Rajan (2009) present a technique that performs offline analysis of basic blocks to detect phase transition boundaries in applications. This information is then used to group cores of similar performance into clusters. Based on this, it is possible to match every application phase to its most suitable core.

Khan and Kundu (2010) use an empirical model on basic blocks to detect phase changes using performance counters such as IPC, speculative IPC, cycle count, and power. These are used as input for a linear regression model that estimates the application performance and power in all the cores of the system and uses this information for dynamic scheduling. Cong and Yuan 2012 also use a regression model to estimate energy consumption and Energy-Delay Product (EDP) based on hardware counters.

Works have also focused on optimizing the scheduling of the Operating System (OS) kernel on heterogeneous systems. In (MOGUL et al., 2008) and (HRUBY; BOS; TANENBAUM, 2013), the authors have noticed that there is not a considerable difference in executing the OS on a big core than in a small core. This is because the OS code (kernel, virtualization helper, and device interrupts) usually causes the core to go idle while waiting for an external response, or simply is not performance intensive. Thus, the authors in (MOGUL et al., 2008) propose running such OS code in a dedicated small core for improving energy and area efficiency, while user application code is allocated to big cores.

As already discussed, migrating threads correctly is essential to exploit the heterogeneous system's capabilities fully. However, each migration incurs into overheads of performance, as the whole state of the core must be saved in memory to be reloaded in the new target core. This cold start in the new core leads to many initial cache misses and performance drop. In (GUTIERREZ; DRESLINSKI; MUDGE, 2014) the authors compare the trade-offs between using private and shared Last Level Cache (LLC) on the energy efficiency of heterogeneous processors. They show that the shared LLC incur in a lower number of coherence misses but cannot be powered off to save energy. They

also show that if the switching interval of threads is high enough (100K instructions or more), the performance difference between shared and private LLC is negligible. Thus, a private LLC can be employed to save energy, as it can also be powered off when the core is not in use. Brown et al. (2011) present a technique to improve performance after the thread migration. The method consists of recording the access behavior of the thread to predict future data and instruction accesses. During migration, this information is used to prefetch data into the cache of the destination core.

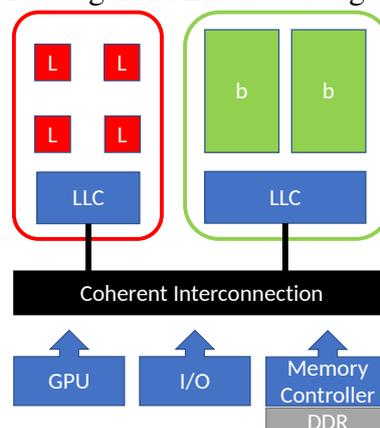
To build a complete heterogeneous system, one must combine both the hardware and scheduling strategies. In the following subsection, we discuss two relevant industry implementations for single-ISA heterogeneous processors and how they have approached both the hardware and scheduling implementations.

2.1.3 Industry Implementations

The single-ISA heterogeneous processors have led to industry technologies such as the ARM big.LITTLE (ARM, 2016). In these processors, clusters separate the LITTLE (small, energy efficient, in-order models) from the big (larger, high single thread performance, out-of-order) cores, while their communication is maintained by a cache coherent interconnection, as shown in figure 2.3. There are three modes in which the threads can migrate between the cores of a big.LITTLE system (ARM Ltd., 2013; JEFF, 2013):

- **Cluster Migration:** Figure 2.4a, in this mode, only one of the two clusters (big and LITTLE) can be active at the same moment. If the current applications require high performance, then the big cluster is activated. However, if the usage drops to

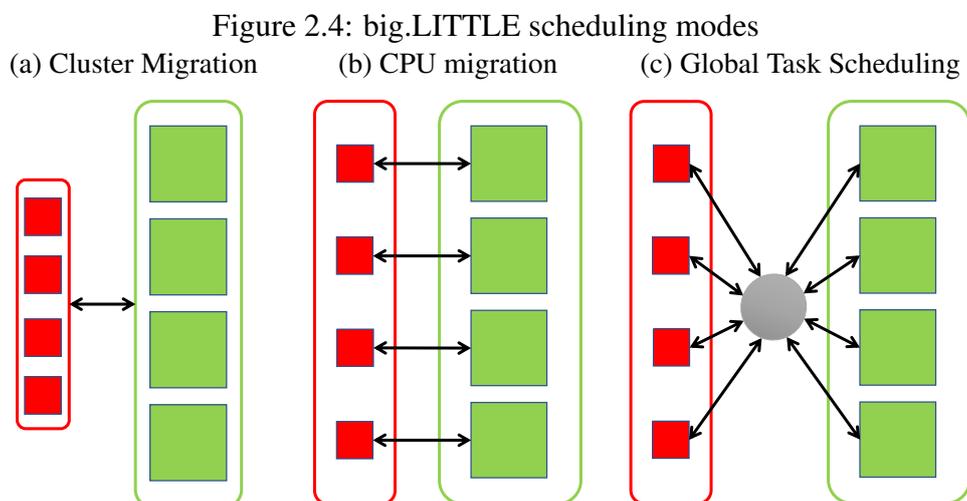
Figure 2.3: big.LITTLE cluster organization.



Source: The Author

a certain level, the threads migrate to the LITTLE cluster, and the big is turned off. The migration in this mode is controlled directly by the big.LITTLE system, driven by the dynamic voltage and frequency scaling (DVFS) levels - which, in turn, are determined by the Operating System. Thus, the OS does not explicitly controls the migration processes but can influence them.

- CPU Migration:** Figure 2.4b, in this model, although clustering is still determined by core size, each big core is paired with a LITTLE core, and only one of these cores can be active at the same moment. The pairs are independent of each other, so each pair can have a different type of core active at the same time (pair 1 has a big core active, while pair 2 has a LITTLE). In this mode, thread migration is also transparent to the Operating System and is decided accordingly to the DVFS level of each core - thus, in a finer granularity than Cluster Migration. As each core from the big cluster must be paired to one from the LITTLE, this model requires the same number of cores in both clusters.
- Global Task Scheduling (GTS):** Figure 2.4c, while the models mentioned above are transparent to the Operating System they both require some of the cores to stay inactive while their counterparts are executing. The GTS model exposes all the available cores, and their compute performance to the OS and its scheduler. Through this mode, the processing capabilities of the system are fully unlocked, at the expense of increasing the complexity in the OS scheduler, as it becomes responsible for deciding the allocation of each task. In case not all cores are needed, unused processors can be powered off, and if all processors of a cluster are off, then



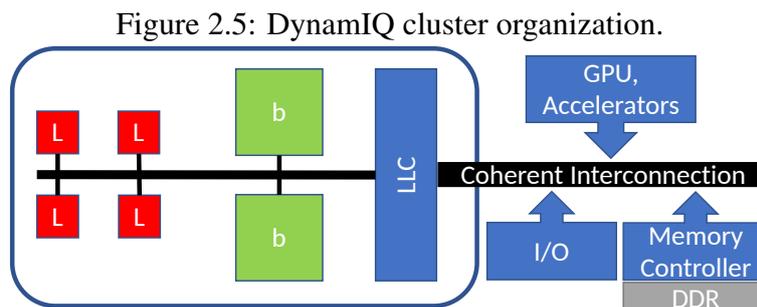
Source: The Author

the entire cluster can be powered off, further decreasing energy consumption. This model also allows the use of different numbers of big and LITTLE cores.

Recently, ARM has released a successor for the big.LITTLE technology called DynamIQ(ARM, 2018). This new technology is highly supportive of the Global Task Scheduling and brings even more flexibility, as now both big and LITTLE cores form one single cluster. With single clusters, the system designer can cover more use cases, such as having a single big core for heavy single threaded performance coupled with seven LITTLE cores for multithreaded workloads. Furthermore, instead of communicating through an interconnection, the cores can share a LLC inside the cluster, drastically improving task migration performance. The coherent interconnection is used in this model to communicate to the main memory, closely coupled accelerators and external I/O devices. Figure 2.5 shows a diagram of DynamIQ core organization.

2.2 The Impact of the ISA on the Processor

The Instruction Set Architecture (ISA) is an interface layer that bridges the communication between software and hardware in any computing system. In GPPs, the ISA is incremental, i.e., it can be extended to reflect new features in different generations of processors. For instance, the x86 architecture has several extensions, including the traditional x87 for FP operations, the SIMD-oriented MMX, SSEx and AVX, and the cryptography AES and SHA instructions. All of these instructions have been introduced to improve the performance of emerging applications. For example, the first vector instructions (MMX) were introduced to accelerate multimedia and 3D applications, which were popular among personal computer users. Nonetheless, many major architectures include different extensions - not only the x86 -, either from periodical updates in functionality or as ways to customize the processor design. In table 2.1 we present examples of ISA



Source: The Author

Table 2.1: Examples of extensions from the x86, ARM, Power and RISC-V architectures.

		<i>Extensions</i>				
		Floating Point	SIMD	Compacted	DSP	Crypto/Security
<i>Architecture</i>	x86	x87	MMX, 3DNow!, SSE, AVX	-	SSE2	AES, SHA
	ARM	VFP	ARMv6, NEON	Thumb	DSP-E	TrustZone
	Power	v.2.05	Altivec	-	DSP	v.2.07
	RISC-V	F, D, Q, L	P, V	C	P	-

extensions for different architectures.

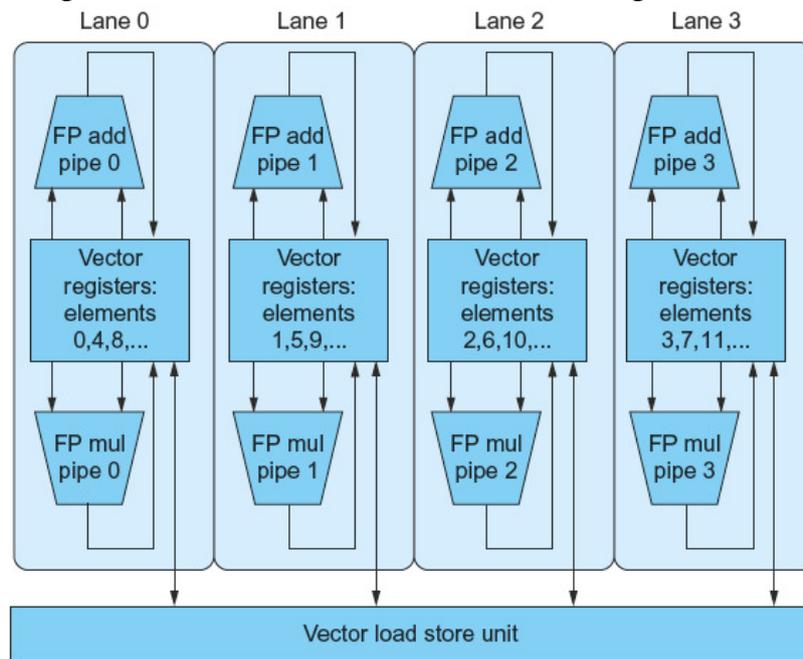
Float point is a real number representation used for computer arithmetic. The larger the number of bits for internal representation, the higher is the precision allowed for the real number. In the current IEEE standard for FP arithmetics (IEEE-754, 2008), FP representation can range from half-precision, using 16-bit registers, to octa-precision using large 256 bit registers. Most GPP architectures include FP extensions as they are normally used in signal processing and engineering and scientific applications.

Single Instruction Multiple Data (SIMD) instructions use vector registers to execute the same instruction over many values. For instance, a 128-bit register can store up to sixteen 8-bit characters, or four 32-bit integers. These operands are then split for execution, either sequentially in a pipeline or parallelly in multiple execution lanes (PATTERSON; HENNESSY, 2013). Figure 2.6 shows an example of a vector lane for executing SIMD FP adds, multiplications and load/store. These instructions are also common on GPP as they can greatly increase the performance in vector computations, such as in *for loops*.

Compacted instructions are used to reduce the compiled code-density. For instance, the Thumb instruction set uses 16-bit long instructions instead of the traditional 32-bit word (or 32-bit in a 64-bit processor). To achieve this, the instruction must either use implicit operands or restrict the access to some of the registers. These are more commonly used in embedded GPPs that have limited resources to reduce the memory footprint of the code. Digital Signal Processing (DSP) instructions are heavily used for signal processing, such as audio decoding, and Cryptography/Security are generally used to ensure confidentiality and system integrity.

Each ISA has its advantages when expressing the processor functionalities. For instance, the ARM's Thumb is efficient in reducing memory footprint, while the x86 compiles many features, such as wide vector processing and cryptography. Venkat and Tullsen (VENKAT; TULLSEN, 2014) show that a system can exploit the many traits of different ISAs to improve the effectiveness of heterogeneous processors. The authors

Figure 2.6: Structure of a vector unit containing four lanes.

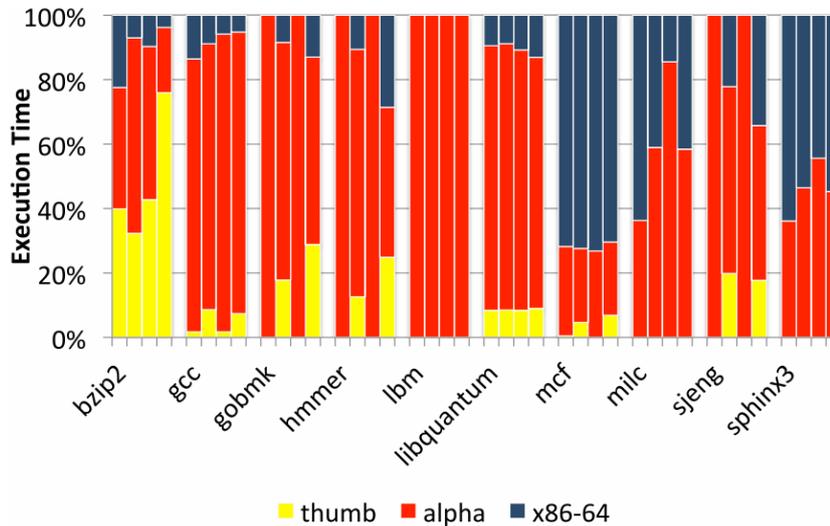


Source: (PATTERSON; HENNESSY, 2013)

combine cores of three ISAs (32 bit Thumb, X86-64 and Alpha 64 bits) in a single system, classifying their performance according to aspects such as FP and SIMD operations, register pressure, code density, and dynamic instruction count. This environment with many different ISAs introduces several challenges in memory management and process migration. Each core of different ISA has a distinct runtime state, and also the memory layout (virtual address space and page table hierarchy) is dependent on the architecture, so it is not possible to migrate threads between different architectures without some intervention. To overcome this problem, the authors employ a *fat binary* that combines target-specific code sections with target-independent data sections along with binary translation between ISAs. Furthermore, to overcome the problem of address translation between 32- and 64-bit architectures, the authors employ a special memory management unit and common page table structure based on the one used in X86-64 for all the three ISAs.

The authors also show that the applications (used in their experiment) present distinct ISA affinities in different phases of execution. In other words, an application can perform better using ISA A in one code region, and then change its behavior to something that is optimally executed with ISA B. Figure 2.7 presents the affinities found by the authors on single- and multi-threaded applications when their heterogeneous system is optimized for performance and EDP. The results show that most applications have phases in which different ISAs would perform better. However, these are mostly related to the

Figure 2.7: ISA affinity for different applications on designs optimized for (bars from left to right) - (bar 1) Single-thread performance, (bar 2) Multi-programmed workload performance, (bar 3) Single-threaded workload EDP, (bar 4) Multi-programmed workload EDP.



Source: (VENKAT; TULLSEN, 2014)

features each ISA - combined with its microarchitecture - can deliver. For instance, pure floating point applications (such as *lbm* in the figure) avoid using thumb, as this ISA does not support FP operations. When the FP operation is also vectorized, the x86-64 is preferred, as it delivers support for SIMD execution.

In an extension of the previously discussed work, Venkat et al. 2019 propose the composite-ISA cores, an architecture that can tailor cores to use specific ISA features to maximize the performance of the system. The authors deeply explore the design space of many systems using different applications to show the potential of systems composed of cores with tailored ISA features and also different microarchitectures.

Blem et al. in (BLEM; MENON; SANKARALINGAM, 2013; BLEM et al., 2015) have shown that Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC) ISAs have no particular impact on performance and power in modern processors. The authors have examined multiple workloads in different processors using ARM (RISC), MIPS (RISC) and X86 (CISC) instruction sets and observed that, rather than the complexity of the instruction set, what really impacts performance and power is the capacity of expressing richer semantics. In other words, the strength of an ISA is characterized by the presence or absence of specializations such as float point and SIMD instruction on one set over the other.

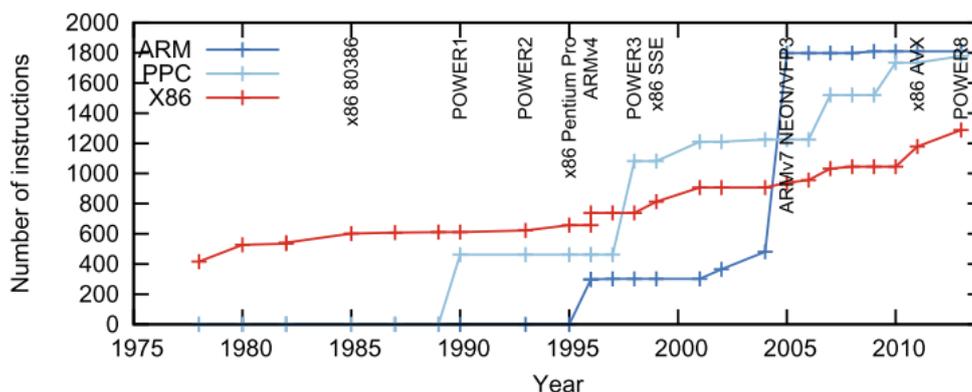
Nonetheless, (LOPES et al., 2015) show that ISA extensions that add specialized

instructions can also be a source of performance degradation in a processor. ISAs such as the x86 have so many instructions that the processor decoder itself needs several stages in the pipeline to decode each instruction (FOG, 1996). These specializations in the ISA reflect the need of the emerging applications on the time of the extension introduction. However, applications change, and newer instructions can take over the functionalities of older ones. Lopes et al. 2015 perform an extensive analysis of ISA aging and the cost in the decoder for keeping old operations in the X86 architecture. In the figure 2.8, the authors show the growth in the number of instructions in different ISAs, with an impressive increase in the ARM architecture when the NEON and Vector FP operations were introduced. They also present a study on the instructions usage in two OSs: Windows 95 and Windows 7. Figure 2.9 shows that 38% of the instructions in the x86 set are not used in both OSs and that there are many instructions that, while were frequently used in the Windows 95 environment, became deprecated in the newer OS. The authors propose a technique to remove and recycle instructions that are not used by compilers anymore. Removed instructions that are eventually fetched for execution in the processor must be emulated through software for backward compatibility.

2.3 Overlapping- and Partial-ISA

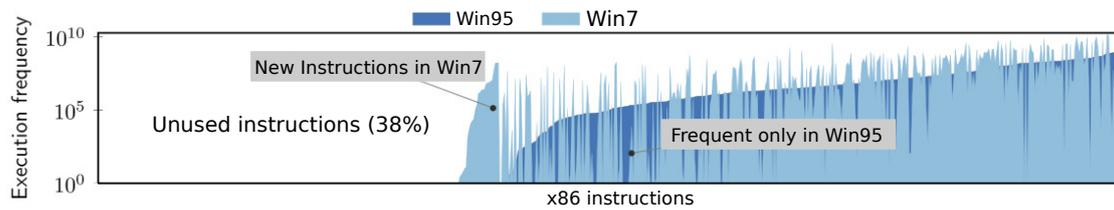
The works mentioned in the previous section show that a diverse and renewed ISA is essential for processor performance, although most of the added instructions are used only for specific applications. Thus, it is important to adapt the newer GPPs so that they

Figure 2.8: ISA growth (in number of instructions) over time for different processor architectures.



Source: (LOPES et al., 2015)

Figure 2.9: Histogram (in logarithmic scale) of dynamic instructions sorted by frequency with respect in Windows 95 and compared to their corresponding frequency in Windows 7. Spikes show differences in the usage pattern.



Source: (LOPES et al., 2015)

can deliver performance to emerging applications. However, current designs implement the full ISA capability of a processor, even when these specialized applications do not occur as often as they would support. The outcome of such designs is a processor that invests a great number of resources in components that are usually underused.

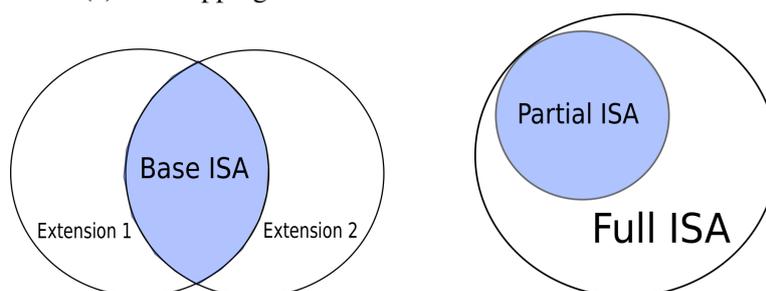
To try to balance instruction usage and avoid resource waste, processors of overlapping-ISA cores have been used on previous works (LI; BRETT; KNAUERHASE, 2010)(REDDY et al., 2011). These are processors in which the cores individually implement different extensions, but all share a base ISA. In figure 2.10a, a diagram examples the coverage of overlapping-ISAs, in which each core implement a distinct ISA extension, but the base ISA overlaps both. For instance, in an X86 processor, the first core can implement the whole SSE and AVX instruction set, while leaving other extensions to the second core. However, both of them implement the base x86 instruction set (integer, boolean and memory operations). On the other hand, in the figure 2.10b one of the cores defines the whole ISA and its extensions, while the second implements just a subset of the instructions (REDDY et al., 2011), namely a partial-ISA implementation.

In the next sections, we discuss works that have explored the requirements for im-

Figure 2.10: Partial ISA processors implement a subset of the full ISA.

(a) Overlapping-ISA set.

(b) Partial ISA set.



Source: The Author

plementing overlapping- and partial-ISA designs. We present these in OS, scheduler, and microarchitectural support. We also present the current state-of-the-art implementations of such systems.

2.3.1 Operating System Support

When a system provides cores with partial or overlapping ISAs, there must be a consensus on how the applications will use these cores. In other words, a task must be allocated to a core that implements the instruction it has currently fetched. The allocation decision, however, can be resolved in different levels of the system. In (REDDY et al., 2011), Reddy et al., categorize functional asymmetries that can be exposed by the application software (e.g., by the programmer or a library) and those that can be handled directly by the Operating System (OS). By using a heterogeneous-aware software approach and leaving the burden of allocation to the programmer, the complexity of the scheduler can be reduced, although this would greatly affect the software cost of new systems (each different heterogeneous processor would require a specific version of the function libraries). The authors also noticed that it is essential that legacy code, programmed to be oblivious to the system heterogeneity, must be able to extract the best performance from these processors. Therefore, the OS must also give support for scheduling legacy applications in cores with different ISA extensions. The authors broadly discuss many topics that must be considered when porting an OS to the heterogeneous environment, including - but not limited to - page tables, physical addresses decoding, paging caches, memory topology, management of non-overlapping instructions, performance monitoring, and virtualization.

Li et al. (2010) have further studied the OS changes needed for overlapping-ISAs and have categorized these challenges in two sets: **correctness** and **performance**. OSs discover processor features during the bootstrap and then assume the same features for all cores. In a functional heterogeneous processor, this assumption is invalid, as some applications may fail in one core, but execute correctly in others. Thus, one of the challenges is to handle these different core features and to ensure the **correctness** of the execution by allocating tasks in cores that can execute their instructions. The **performance** challenge lies in the scheduling of threads, as in most heterogeneous processors. The OS must ensure fairness when sharing usage time of the high-performance cores between workloads, especially when different users are running concurrent applications. This is further

complicated when workloads have unexpected behaviors. For instance, it is expected that workloads, in general, will run faster on high-performance cores, but this might not hold for I/O-bound applications.

One of the major contributions of (LI; BRETT; KNAUERHASE, 2010) is arguably the mechanism that allows for detection and migration of instructions that cannot be executed in a partial-ISA core, named Fault-and-Migrate. In a traditional full-ISA system, whenever a non-implemented instruction is fetched, the core would trigger a fault to the OS, which in turn would send a signal kill to terminate the application. The fault-and-migrate mechanism handles this instruction fault from the core differently, by activating a scheduling mechanism that migrates to the workload to a new core, capable of executing the target instruction.

Additionally, the authors in (LI; BRETT; KNAUERHASE, 2010) have also highlighted the need for support in transparent workload migration when executing in kernel mode while in a non-full-ISA core. The first situation is when specific code blocks, such as in critical sections, are non-preemptible and cannot be transparently migrated between cores. Another scenario occurs when the code is preemptible, but the faulting instruction (from a non-implemented ISA) is a privileged instruction that changes the CPU behavior. Migrating transparently in this situation could lead in the OS assuming that state change occurred in the wrong core. The authors have assumed in their work that all the cores are able to execute every instruction from the OS kernel in their overlapping-ISA processor. In a partial-ISA implementation, this problem is easily solved by executing kernel mode code in the full-ISA cores.

2.3.2 Schedulers

As with any heterogeneous processor, an efficient scheduler plays a major role in the performance and energy consumption on a partial-ISA system. As already mentioned, in (LI; BRETT; KNAUERHASE, 2010) the authors present the fault-and-migrate mechanism, which is essential for the support of workload migration when unimplemented instructions are fetched by a core. This same work also presents a complete, and fair new scheduler designed for overlapping-ISA processors. The algorithm supports both functional and performance asymmetry, using information on the extensions implemented by each CPU and their performance rating. This data can be provided to the OS through simple hardware changes (more on this in the section 2.3.3). Fairness is ensured by guar-

anteeing that every thread will receive a time slice to execute on fast cores proportional to its priority.

In (KNAUERHASE; BRETT, 2013) the authors propose Kinship, along with a rigorous theoretical formulation of metrics that allow matching dynamic workloads to diverse resources more efficiently than previous works. The Kinship is designed to work in systems that show both performance and functional asymmetry (also through the fault-and-migrate mechanism) and that use the different overlapping-ISA cores as accelerators for specific applications.

Opposed to migration, another common strategy in case of faulting instructions is emulation. If the faulting instructions are sparsely distributed (will execute for a few cycles and then return to basic ISA), emulating these instructions can introduce lower overhead than migrating the entire workload to a different core. In (AMINOT et al., 2015), the authors propose a FPU speedup estimation model to guide schedulers in the decision of migrating or emulating in systems that specifically remove the floating-point operations from its ISA. The goal of the work is to find a more accurate way to balance FPU usage and emulation.

2.3.3 Microarchitecture Support

Overlapping- and Partial-ISA designs require some microarchitectural support for correct execution. For instance, in the fault-and-migrate mechanism, there must be a way for the processor to signal an unsupported instruction fault for the operating system. This signal already exists in most processors as an invalid opcode exception (e.g. UD fault in the Intel architecture (LI; BRETT; KNAUERHASE, 2010)). However, this signal might also be used by regular applications (through the undefined instruction *ud2*) to secure certain code paths from executing, forcing the application termination. Thus, to support fault-and-migration, the hardware must distinguish between faulting instructions that must trigger a migration from those that are used on the current normal execution flow. Furthermore, in an overlapping-ISA environment, to migrate the workload to a supporting core, the OS must know which subset of the ISA each core implements.

To couple with the aforementioned issues, (LI; BRETT; KNAUERHASE, 2010) proposes minimal changes in the way the processor handles faults and identify instruction extensions. The processor can provide, through hardware support, a mechanism for the OS to find the many ISA extensions in all the processors and in which they are present.

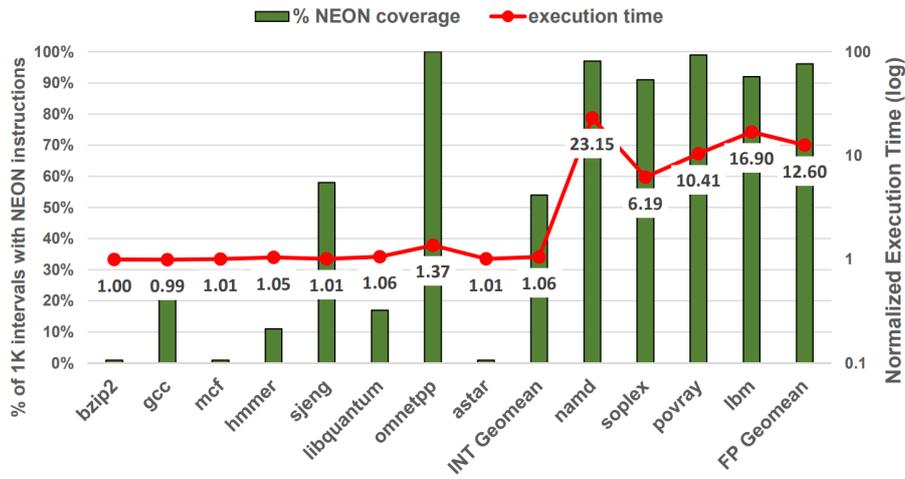
This mechanism is already available in some processors (e.g. the CPUID instruction in the Intel architecture). Using this approach, the OS can build a mapping of the ISA-per-core during the boot process. Thus, the OS would migrate only if the faulting instruction belongs to one of the mapped ISAs in the different cores, otherwise it would consider as a normal faulting execution (not implemented in any core). However, deciding if a given instruction is invalid or actually belonging to a unimplemented ISA is a more complex job. One solution would be to implement the cores in order to decode instructions even from unimplemented ISAs, although this would add unnecessary complexity to the processor. Another option would be leaving the OS in charge of decoding a faulting instruction and deciding where it must be allocated, at the cost of some performance loss.

In a partial-ISA processor, these situations can be easily handled, as the options for migration are binary (either to a partial or full core). If an instruction faults in a partial core (that does not implement the whole ISA), it is either because of a migration or a "normal" termination event. However, if it faults in a full-ISA core, the only valid option is the termination event. Thus, the OS can assume that every fault in a partial core triggers a migration to a full core, and the faults in the full core, a termination. This introduces an unnecessary migration in the case of a termination instruction in the partial core, which can be mitigated by the OS by caching such instructions for future checking (LI; BRETT; KNAUERHASE, 2010). Such termination instructions are much more unlikely to happen than the migration faults, thus it is more efficient to introduce an eventual miss in migration than including complex checks to completely avoid such wrongful migrations.

2.3.4 Partial-ISA Implementation

Regarding partial-ISA processor implementations, the work in (LEE et al., 2017) studies the usage of different ISA extensions in an ARM processor. Figure 2.11 shows the analysis for usage and execution time (when emulating) the sets of NEON, predicate and Load/Store Multiple instructions. The authors argue that, although some sets are rarely used, they can still cause a significant performance drop if removed. In the case of NEON instructions (Figure 2.11a), the impact in performance can reach up to 23x in specific applications. Observe that in Figure 2.11a the authors present the coverage of NEON operations in every 1K instructions (if at least one NEON operation is executed every 1K instruction, then the coverage is 100%), not the usage. Thus, a designer cannot

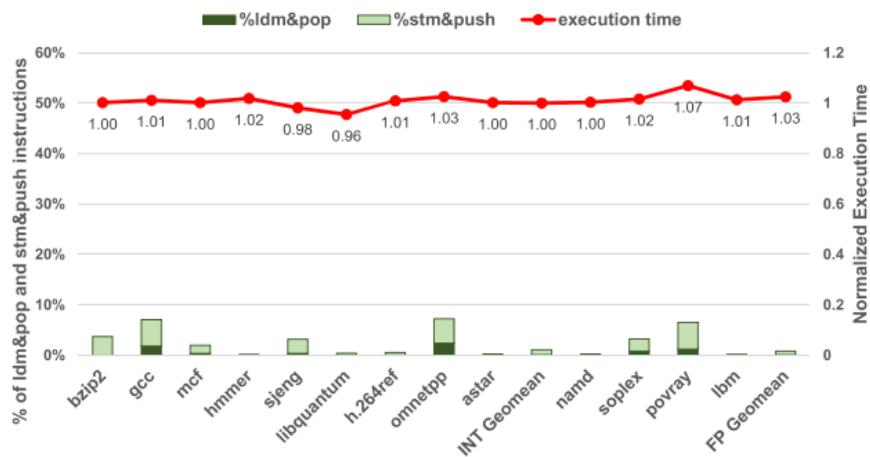
Figure 2.11: ISA extensions usage and execution time.
 (a) Performance of ISA without NEON instructions.



(b) Performance of ISA without predicate instructions.



(c) Performance of ISA without Load/Store Multiple instructions.



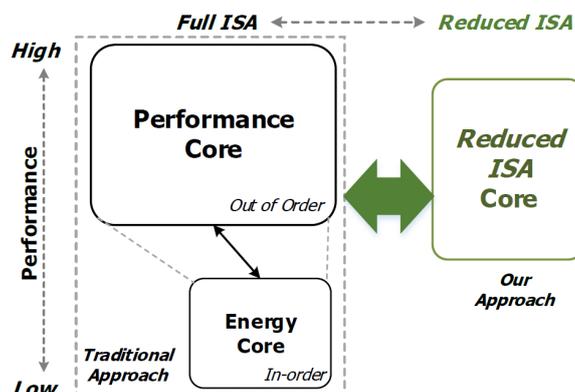
Source: (LEE et al., 2017)

simply remove such extensions from a system and expect the same performance from every application.

Based on the aforementioned results, Lee et. al (2017) propose a system composed of two ARM A15 processors, in which one of these implements the entire ARM ISA (which they call a full core) and the other does not implement the NEON, predicate, DSP and load/store multiple instruction sets (which they call a reduced core). In this system, the reduced core is given full priority to execute applications, in order to save energy. Whenever a non-implemented instruction is fetched, the core starts an emulation phase. If the number of emulated instructions exceeds a certain amount during a period, then the workload is migrated to the full core. While one of the cores is executing, the other stays idle, so it implements a cluster-like approach from the big.LITTLE technology. Figure 2.12 shows how the proposed system structure compares to the traditional big.LITTLE approach. The reduced core maintains most of the performance from the OoO core (apart from the removed instructions), but with lower energy consumption. This reduced core, however, is not as energy efficient as an in-order core.

In this thesis, we base our work on concepts from the state-of-the-art. We infer that the challenges presented in section 2.3 (Overlapping and Partial-ISA) - along with their solutions - are enough to provide a functional heterogeneous environment. We have also used schedulers based on the fault-and-migrate mechanism that aims to evaluate the potential increase in the performance and energy-efficiency of the system. These schedulers are used in the evaluations of the PHISA system without the use of TUNE (instruction offloading).

Figure 2.12: A heterogeneous system with a reduced-ISA core.



Source: (LEE et al., 2017)

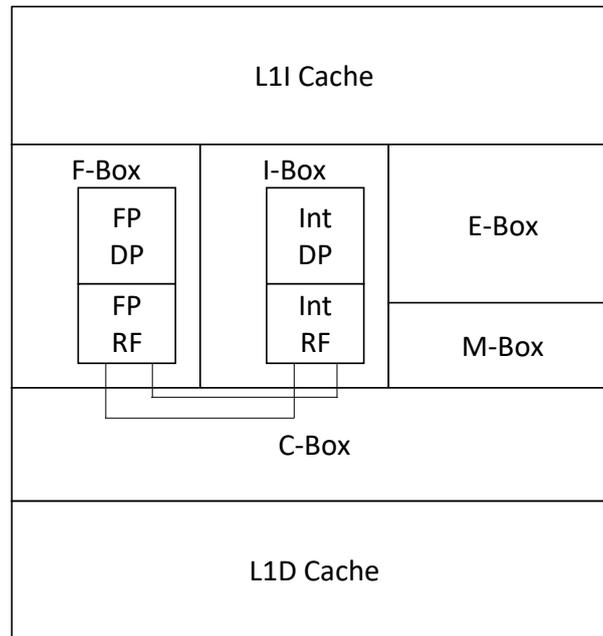
2.4 Sharing Resources Between Cores

The feasibility of physically sharing the SIMD/FP unit has already been studied by Kumar et al. (KUMAR; JOUPPI; TULLSEN, 2004) using cojoined-cores. Their work shows that the layout of multicore processors can be designed to keep components that can be shared in positions that minimize their distance between the cores. Figure 2.13a shows the typical components of a core, while figure 2.13b shows how a dual-core processor can be designed to share the FPU and L1 caches. By keeping the F-Box in the borders of the core, it is possible to design the floorplan to maintain the unit distance equal for all cores, minimizing the routing impact.

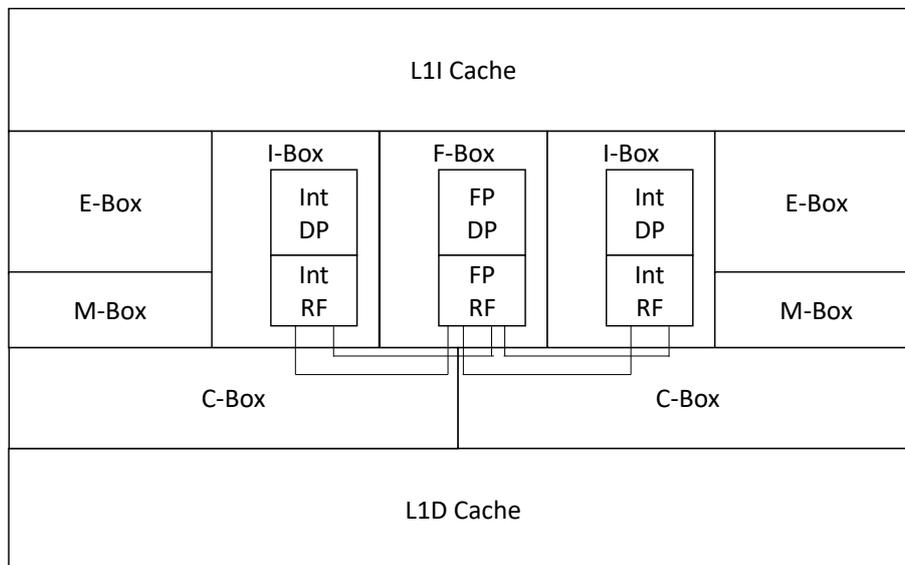
Commercial core designs like the AMD Bulldozer (SHIMPI, 2011), have adopted the cojoined approach for sharing pipeline stages between two integer clusters. Figure 2.14 shows how the microarchitecture is designed, tightly sharing pipeline stages such as the instruction fetch and dispatch and components such as the FPU and instruction cache between the integer components. In this organization, two integer threads can run simultaneously in the same core (one in each integer cluster), but only one SIMD/FP thread can run at a time. Applications that heavily rely on these specific operations will, therefore, experience performance losses due to resource contention.

The Sun Niagara (UltraSPARC T1) (SUN, 2019) is another example of COTS processor in which the FPU is shared between cores. Differently from the AMD Bulldozer, the FP in the Sun Niagara is shared in a loosely coupled way, communicating with the cores through the cache crossbar. Figure 2.15 shows an overview of the Niagara microarchitecture, in which eight integer cores share a single FPU.

Figure 2.13: Cojoined-cores sharing resources such as cache memory and FP units.
 (a) Typical core components such as caches, integer unit (I-Box), FPU (F-Box), memory unit (M-Box), and control unit (C-Box).

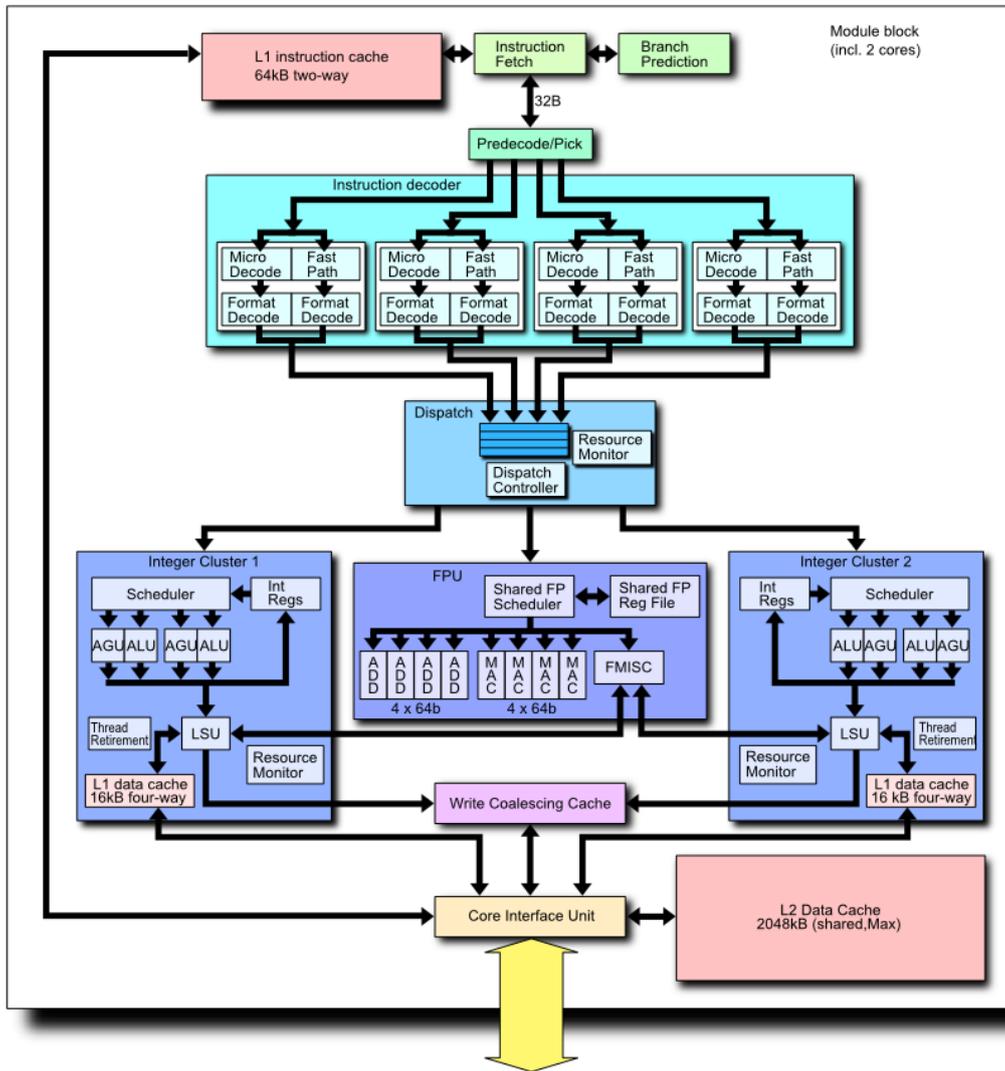


(b) Cojoined-core layout sharing the FPU and L1 instruction and data caches.



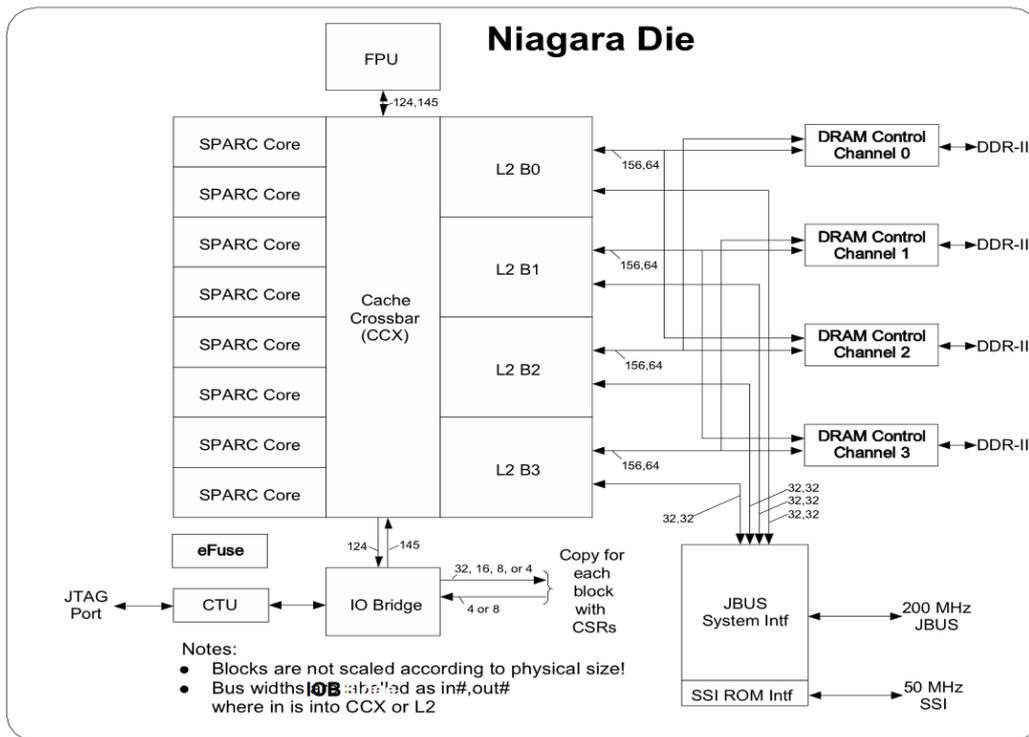
Source: (KUMAR; JOUPPI; TULLSEN, 2004)

Figure 2.14: AMD Bulldozer microarchitecture.



Source: (BULLDOZER, 2011)

Figure 2.15: Sun Niagara (Ultrasparc T1) microarchitecture.



Source: (MICROSYSTEMS, 2007)

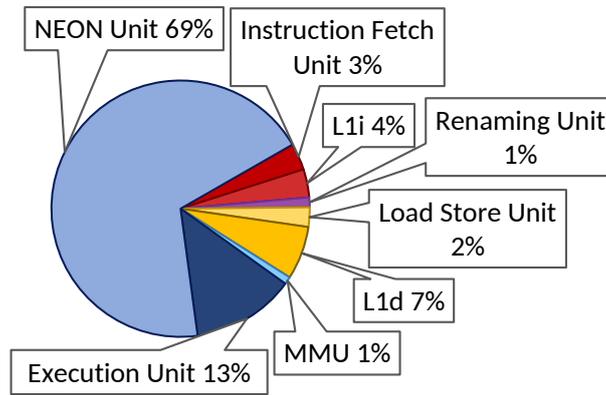
2.5 On the Area of SIMD/FP Units

To protect intellectual propriety, most companies do not provide detailed information on the design of commercial architectures. For this reason, it is a common practice for researchers to estimate data such as the total area or power of a processor by using tools or models. One frequently used tool to estimate the area of a processor and its components is McPAT (LI et al., 2009). McPAT estimates area, power, and timing of different microarchitectures by using known data from real processors and escalating such data to different technologies.

Figure 2.16 shows an example of the area breakdown of an A15 core as modeled by McPAT. McPAT reports a total area of about 3.5mm^2 for the A15 core, including L1 caches and the memory management unit. From this reported area, 69% is related to NEON specific components, such as the FP register file, renaming tables, and the execution pipelines.

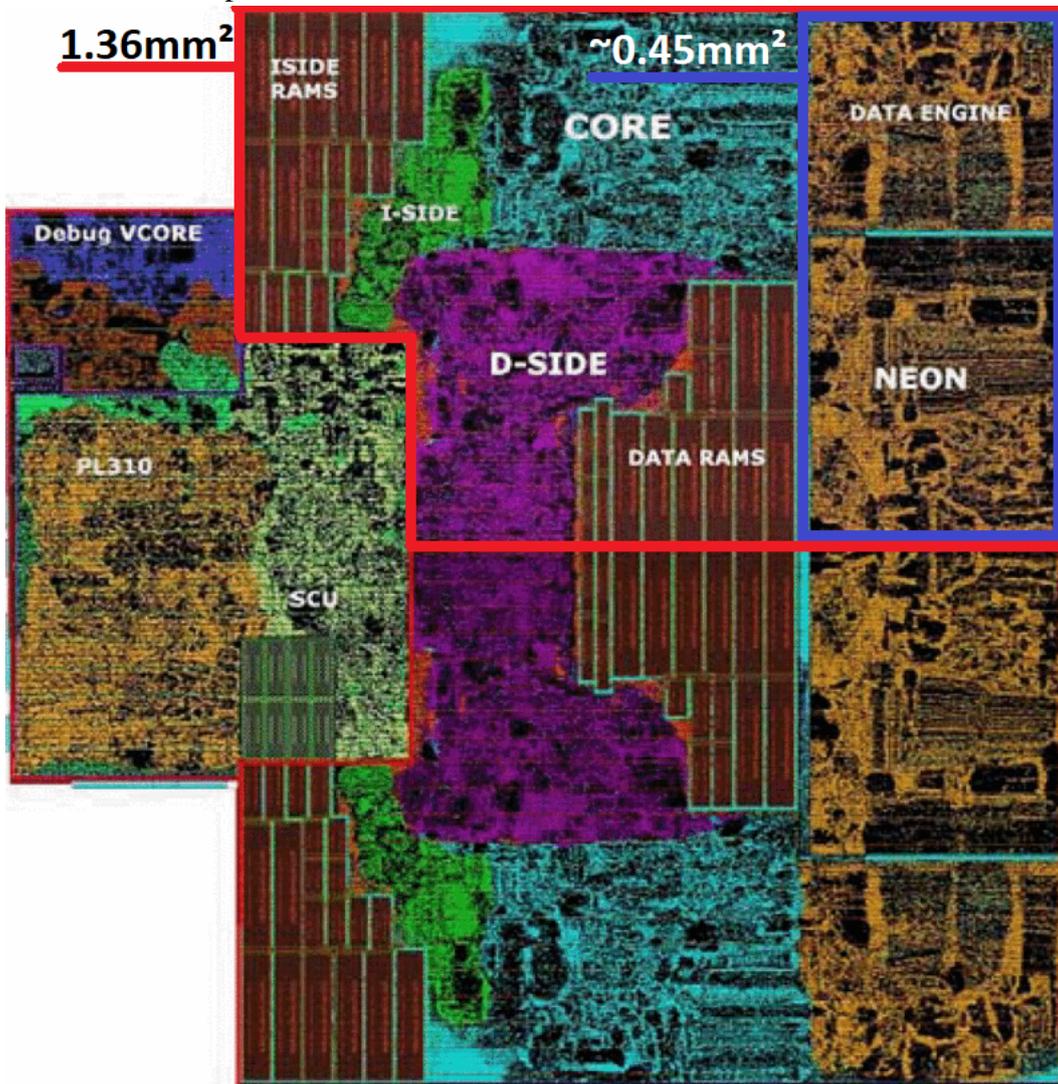
Although only actual hardware synthesis - which is not detailed published by manufacturers - can confirm such ratios to be accurate, the McPAT data can be corroborated by simple extrapolation of publicly available data. Figure 2.17 published by Koppanalil et al. 2011 shows the floorplan of a dual-core A9 processor with its different components

Figure 2.16: Area breakdown by processor component in an ARM A15 accordingly to McPAT.



Source: The Author

Figure 2.17: Dual core A9 Floorplan. Highlighted are the areas of one entire A9 CPU and the NEON unit components.



Source: (KOPPANALIL et al., 2011)

highlighted. The total area reported in this work for one single core is 1.36mm^2 using a technology node of 32nm. In this same figure, we can estimate that the NEON unit occupies about 0.45mm^2 of the core area. This is approximately the same area of a single full A7 core in a similar technology node, as reported by ARM (ARM, 2011) and also McPAT using a 28nm process. These areas are summarized in table 2.2. Nonetheless, the NEON implementation of the A9 core is a single-issue, 64-bits wide unit, that has lower performance than the A15 implementation (ARM, 2010). If one considers that implementing a 2x wider, dual-issue, complex NEON component (A15 NEON is 128-bits wide, has two issue lanes and a deeper pipeline) can increase the area of this unit by 4x, then the area estimates shown in figure 2.16 for the A15 given by McPAT are in accordance with the published data.

A second experiment that supports the McPAT data was performed by Becker et al. (2019), in which a RISC-V 4-issue BOOM processor, based on the same characteristics as an A15 core, was synthesized. This processor has a dual issue FPU (only FP operations are supported, no SIMD support) with double precision (64-bits) that occupies (along with extra FP structures such as the Register File (RF)) about 37% of the area of the core. If one considers that improving this unit to support 128-bits wide words and SIMD operations can at least increase its area by 3x, then again, the area estimates for McPAT would be the same as the synthesized data.

2.6 Contributions Over the State-of-the-art

In this thesis, we show how partial-ISA cores can be mixed with full-ISA cores to improve the system efficiency, both in performance and energy. Differently from previous works, we further explore the benefits of partial-ISA systems, developing the following two novel approaches:

Table 2.2: Areas of full (including L1 caches) A7, A9, and A15 cores and their NEON units. Total and NEON unit areas from the A7 and A15 were reported by McPAT. A9 total area is reported from (KOPPANALIL et al., 2011) and its NEON unit area is estimated from the layout. The NEON unit areas in this table do not include potential areas from wiring, routing, decoding, and other components related to this unit.

	A7	A9	A15
Full Core	0.5mm^2	1.36mm^2	3.53mm^2
NEON unit	0.12mm^2	0.45mm^2	2.36mm^2

- **1 PHISA Multicores:** A processor composed of partial- and full-ISA, capable of executing multiple single-threaded workloads, and that uses thread migration to maintain binary compatibility in the system.
- **2 TUNEd PHISA:** A PHISA processor that implements an instruction offloader. It is capable of bringing the advantages of PHISA to multi-threaded applications, and uses offloading to maintain binary compatibility.

In the following sections, we discuss the contributions that these approaches introduce over the previously discussed works. Each following subsection discusses concisely the contributions over the previously discussed sections, and we finalize with a summary of the overall contributions.

2.6.1 Contributions over single-ISA heterogeneous processors

One of the main contribution that this thesis introduces is the capacity of exploiting the same advantages of single-ISA heterogeneous processors, in a system that implements functionally-heterogeneous cores. Our migration and offloading strategies allow the processor to maintain binary compatibility between full- and partial-ISA cores using different approaches. Our migration strategy can also be augmented to introduce different scheduling policies, further allowing new optimizations, as in a common single-ISA system. On the other hand, the offloading approach is designed to transparently maintain binary compatibility between partial- and full-ISA cores.

2.6.2 Contributions over the impact of the ISA

Our partial-ISA cores help to overcome the typical impact of the growing ISA of modern architectures. Our strategies can be used to remove rarely used components of the cores, actively reducing the area and power footprint of the removed instructions. Our approach can also be complementarily used with strategies that recycle instructions through software emulation. We evaluate scenarios in which emulation can replace thread migration in our system.

2.6.3 Contributions over overlapping- and partial-ISAs

Considering the aforementioned scenarios in the state-of-the-art, in this thesis **we argue that both functional and performance asymmetry can be used to further increase the efficiency of a processor**, either by reducing energy consumption or improving performance. Thus, differently from other works in partial- and overlapping-ISA, we propose to use cores with both different ISA support and performance/energy characteristics. We also propose to maintain the binary compatibility between cores by either **migrating entire tasks** or **offloading non-supported instructions** from partial-ISA cores to full-ISA cores.

Furthermore, we use partial-ISA cores, as for our applications they show the following **advantages** over overlapping-ISA approaches:

- **Compatibility:** Having full cores in the system ease the compatibility issues of applications/systems that require special instruction extensions while executing protected code that cannot be migrated. In an overlapping-ISA system, the kernel code can only use instructions that are present in every core, as seen in section 2.3.1.
- **Complexity:** Although overlapping-ISA systems can be more flexible when providing heterogeneity, partial-ISA systems greatly reduce the complexity associated with scheduling workloads between cores and on solving the problem of *where the given instruction is available*. In a partial-ISA system, the full core will always be able to execute faulting (non-terminating) instructions.
- **Concurrency:** Cores in an overlapping-ISA system can be seen as accelerators for specific classes of instructions. If the system runs multiple workloads that demand the same instruction expansion, it might run in resource concurrency issues. Having multiple full cores in the system decreases the performance impact in this scenario, while still maintaining the advantages of having partial cores for traditional applications.

2.6.4 Contributions over sharing resources

Regarding the sharing of resources, we envision that our PHISA designs can follow the guidelines in (KUMAR; JOUPPI; TULLSEN, 2004), in which cores are built keeping shared resources close to all computing nodes. This can help a TUNE configu-

ration to reduce the total latency of offloading the instructions from partial- to full- cores. When compared to other works that share resources, the PHISA design also has a fundamental difference: instead of having one shared resource for many cores (such as in the Niagara processor), the PHISA designs deliver many resources to many cores. Removing the resources only where they are really impactful ensure a more efficient processor design, while incurring in smaller overheads for the system scheduling. When the TUNED PHISA is considered, we see the inverse trend from the current state-of-the-art: instead of having one shared resource for many cores, we have many resources to be used by one core.

2.6.5 Wrapping up

To summarize the contributions of this thesis, we can highlight the following aspects of both the PHISA and TUNE approaches:

Implementing performance heterogeneity: instead of using cores of same performance (such as full- and reduced-ISA A15 cores only), our approach uses monotonic cores to further optimize processor efficiency. By mixing a big.LITTLE approach (A15 cores + A7 cores) with full- and partial-ISA cores, we can achieve significant improvements in the EDP of the processor that can not be reached even in traditional big.LITTLE systems.

Multiple workload execution: differently from previous similar works on partial-ISA, our method includes a scheduler that allows the usage of all the cores available in the system. We also show that new schedulers can be developed to optimize different features such as performance and energy.

Parallel applications: we also explore the optimization of a partial-ISA system for parallel applications. We characterize such applications to determine which regions mostly benefit from instruction extensions - consequently requiring to run in full cores - and use the extra cores provided by the partial-ISA approach to improve performance in parallel regions.

Task migration vs instruction offloading: we discuss how different applications benefit from either migrating entire tasks from partial- to full-ISA cores or simply offloading individual instructions. We use both strategies to develop a PHISA system with TUNE.

Table 2.3 compiles the main characteristics of the most relevant works (in the sense

Table 2.3: Characteristics of the relevant works compared against the PHISA and TUNEd PHISA systems. N/A means that the given characteristic was not applicable to the given study.

	Organization Heterogeneity	ISA Heterogeneity	Microarchitectural Analysis	Multi-task/thread Analysis	Thread Migration	Instruction Emulation	Instruction Offloading	FU Sharing
(KUMAR et al., 2003)	Yes	No	Yes	No	Yes	No	No	No
(ARM, 2016)	Yes	No	Yes	Yes	Yes	No	No	No
(LOPES et al., 2015)	N/A	N/A	Yes	N/A	N/A	Yes	No	N/A
(LI; BRETT; KNAUERHASE, 2010)	Yes	Overlapping	No	No	Yes	No	No	No
(LEE et al., 2017)	No	Partial	Yes	No	Yes	Yes	No	No
(KUMAR; JOUPPI; TULLSEN, 2004)	No	No	Yes	N/A	No	No	No	Yes
(SHIMPI, 2011)	No	No	Yes	Yes	No	No	No	Yes
(SUN, 2019)	No	No	Yes	Yes	No	No	Yes	Yes
PHISA	Yes	Partial	Yes	Yes	Yes	Yes	No	No
TUNEd PHISA	Yes	Partial	Yes	Yes	No	No	Yes	Yes

of comparison against the proposed systems) analysed in this chapter. In the following chapters, we present the PHISA multicores, its design concept, and the features required for its scheduler - along with different scheduling policies. We also introduce TUNE, its main features, and how it can be employed in current processors.

3 PHISA MULTICORE

To evaluate the potential and feasibility of heterogeneous systems of partial-ISA, we initially show the concept of the PHISA Multicores. The following sections present the PHISA system, its design challenges, and scheduling requirements.

3.1 The PHISA System

The reasoning of a PHISA multicore is that it is possible to remove support from an ISA extension of some cores in a processor while maintaining it in others. This freed area can then be used to add support for more efficient processing elements, for instance, increasing the core count. A high-level overview of this design is shown in figure 3.1, in which the SIMD/FP unit of a big OoO core is replaced by two little in-order cores. The PHISA multicore design removes hardware components specifically used by an ISA extension while leaving the remaining microarchitecture of a core unaltered. This core, which we call a partial-ISA core, keeps its ability to execute instructions from the base ISA. Therefore, performance is only affected for the removed instructions, as parameters such as issue-width¹, execution order and branch prediction are all kept the same.

Each ISA extension adds its particular logic complexity. In this proof-of-concept, we have focused on the SIMD and Floating-Point (FP) instructions, as they are usually implemented in the same block of components that result in a high source of logic overhead in the processors, as discussed in section 2.5. Accordingly to Smith and Sohi 1995, the typical superscalar processor is organized in modular components, as pictured in fig-

¹If the extension removes entire functional units, then the back-end width might be decreased.

Figure 3.1: Example of PHISA configuration. The resources freed by an instruction extension are used to increase the core count.

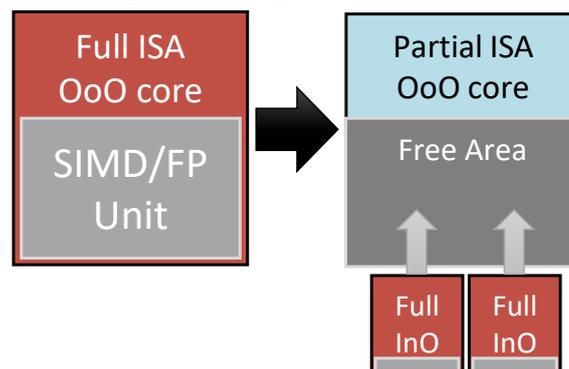
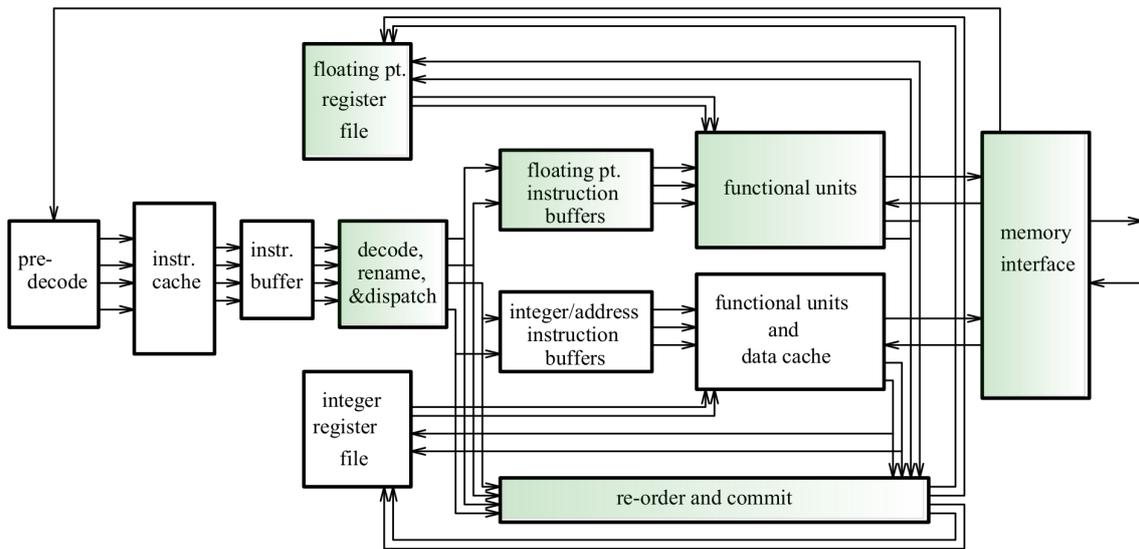


Figure 3.2: A typical OoO processor datapath. In green, the parts of the datapath that can be trimmed or simplified when removing the floating-point support from the processor.



 Datapath impacted by FP support

Source: Adapted from (SMITH; SOHI, 1995) by the author.

ure 3.2. This modularization allows us to exclude the FP extension by safely remove the entire SIMD and FP pipelines from the execution unit of the processor. The decoder stage can also be trimmed by removing support for these instructions, as well as the FP instruction window in the fetch stage of out-of-order designs. Separate FP register renaming table and the FP RF² in the dispatch stage can also be removed. Other general targets for trimming include routing logic, such as write-backs, forwarding, and even the clock-tree of the processor. In figure 3.2, the blocks in green represent the components that can be removed and trimmed during the ISA trimming process. Other ISA extensions (not considered in this work) would incur in more logic trimming in different regions of the processor. For instance, DSP instructions would simplify the integer pipeline of the processor by removing the Multiply-Accumulate (MAC) operations.

3.2 Scheduling

As with any heterogeneous processor, an efficient scheduler plays a major role in the performance and energy consumption of the system. In a PHISA multiprocessor, the scheduler must be aware of which cores are capable of executing the ISA extensions so

²assuming the multiplication unit is adapted to use the integer RF

that it can migrate workloads from partial to full cores when necessary. The basic features that a PHISA scheduler must provide are the two following:

- To detect and/or identify unsupported instructions that are not implemented in partial-ISA cores, but have an implementation in the full-ISA cores.
- To classify cores between partial- and full-ISA cores and migrate tasks accordingly to their current execution needs.

We start by developing a minimalist scheduler that implements only the essential features required for the PHISA system to work. From this simple scheduler, we improve the features to allow instruction emulation and also to apply different optimization policies.

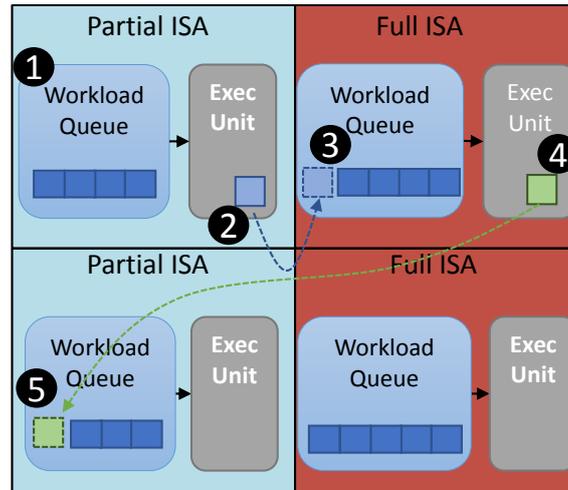
3.2.1 Minimalist Scheduler

The basic developed scheduler implements the two required features listed in section 3.2, as well as a minimum time on core feature and a job-stealing functionality. The minimum time on core is applied in full cores as a preemption phase. Its function is to avoid multiples migrations of the same application and to allow other workloads in the queue a fair slice of time to execute (integer cores are allowed to execute integer applications to completion). Job stealing is used to avoid cores to stay idle when there are still other cores with more than one job in their queue.

Figure 3.3 shows a graphical representation of the scheduler decisions. Each core (which can be a partial- or full-ISA core) keeps a queue of workloads (different applications) to execute (1). When a core is idle, it fetches a workload from its queue in a FIFO manner and executes the workload until a migration event is triggered.

In this scheduler, there are two events in which a workload can migrate from a core to another. The first is when a partial core fetches an unimplemented instruction (2). When a typical processor fetches an instruction that cannot be decoded, it generates a trap to the operating system, which would signal a kill command for the process. In this work, we implement a fault-and-migrate strategy(LI; BRETT; KNAUERHASE, 2010) to handle the reallocation of workloads. Instead of treating the trap with a signal kill, the operating system activates the ISA-aware scheduler that migrates the workload to the less busy full core in the system (with the shortest workload queue)(3). The second event for migration is activated after a workload has been executed for a minimum time in a full

Figure 3.3: Scheduler events between full and partial ISA cores.



core(4). Similarly, the workload will be migrated to the less busy core (which can be either partial or full)(5).

Algorithm 1 shows the pseudo-code for the employed scheduler. The algorithm decides to migrate the workloads accordingly to the event that fired the scheduler: either by an instruction fault - in which the migration occurs to a full core (lines 1-3) -, or after the minimum time on the core - which migrates to any core (lines 4-6). In case the core migrates its last task from its queue (lines 8-9), the scheduler tries to steal and allocate a task from the busiest core in the system.

Many applications present the behavior of interleaving integer and floating-point operations, which would cause frequent back and forth migrations. To handle it, we consider a minimum time a thread must stay on full cores of 160K cycles, as suggested by previous studies as a period that introduces minimal impact on performance (CONSTANTINO et al., 2005). The established minimum time on a core helps to reduce

Algorithm 1: Scheduler pseudo-algorithm

```

Input: event, core
1 if event = instructionFault then
2   | minQueueFromFullCores.pull(core.workload);
3   | core.workload ← core.queue.pop();
4 else if event = minTimeOnCore then
5   | minQueueFromCores.pull(core.workload);
6   | core.workload ← core.queue.pop();
7 end
8 if core.workload = None then
9   | core.workload ← maxQueueFromCores.pop();
10 end

```

these migrations when the application is executing on the full core. Although this is a period of time that is fixed in our scheduler, the reader should be aware that it can affect the behavior and performance of the PHISA system. The impacts of varying the preemption time were not evaluated in this thesis, and should be analyzed in future works. However, as we see in the following chapters, the given preemption period can already provide performance and energy improvements for the PHISA system.

3.2.2 Minimalist Scheduler with emulation

Although the minimum time on core reduces the number of migrations, they will still happen if the interleaved application is rescheduled to a partial core after the minimum time. We extend our scheduler to handle this situation by *(a) prioritizing migration of FP applications to full cores or (b), if available to the system, by triggering FP instruction emulation in software*. In the case of software emulation, the task requiring the non-implemented instruction can still execute in the partial core for a threshold time (we use the same 160K cycles as a threshold). Algorithm 2 shows the modified scheduler using emulation. If the instruction fault happens in a core allowed to emulate, than the event is ignored, and migration will only happen after the minimum time executing.

To calibrate our simulator (see more details in chapter 5) to consider the cost of emulating SIMD and FP operations, we have run two different versions of benchmarks. The first version is compiled to use all instructions from both SIMD and FP extensions. The second version is compiled to use only scalar code (no SIMD) and to emulate FP using integer libraries (soft FP). We ran both of these versions and found that the binaries with emulation are, on average, 40x slower. Thus, we consider a multiplicative value of 40 cycles whenever our scheduler decides to emulate instructions in the PHISA system.

3.2.3 Scheduling Policies

Supporting different optimization goals is also an essential feature in the scheduler for heterogeneous processors, which can also be supported by the PHISA system. In this section, we modify our minimalist scheduler to allow prioritization of tasks in cores to improve either system performance or energy consumption.

In this version of the scheduler, we introduce the following modifications:

- The workload queue is unified. Instead of using a queue for each core, all workloads are sent to the same queue, where they can be prioritized.
- Preemption in all cores. Instead of allowing integer applications to run to completion in integer cores, we apply preemption in all cores, so all applications have the same time slice to execute, giving fairness to the system.
- Application annotation. Applications are annotated accordingly to the requirements they need. This is used for the scheduler to decide if the application must be allocated to a full- or partial-ISA core.

Given these modifications, we create two versions of the scheduler, one trying to optimize execution for performance, and the other for energy consumption. For performance optimization, we assume that the system’s OoO cores always present better performance than the in-order cores. For energy, we assume that the in-order cores will always have better energy efficiency. Furthermore, applications that are not specifically marked to use instruction extensions (are executing instructions from the base ISA) will prioritize execution in partial-ISA cores. Algorithm 3 shows the algorithm for the scheduler optimizing for performance, while Algorithm 4 shows the scheduler for energy consumption.

In this new system scheme, all workloads are released from their cores after the preemption phase and sent back to the workload queue. Cores that fetch non-supported instructions will immediately release their workloads and call the scheduler for a new assignment. These workloads are annotated as ISA dependent on their next allocation. The algorithm is executed after the preemption release for all workloads in the queue, in a First In First Out (FIFO) manner, until no core is left idle or the queue is empty.

Algorithm 2: Scheduler pseudo-algorithm with emulation

Input: *event, core*

```

1 if event = instructionFault then
2   | if not core.canEmulate then
3   |   | minQueueFromFullCores.pull(core.workload);
4   |   | core.workload ← core.queue.pop();
5 else if event = minTimeOnCore then
6   | minQueueFromCores.pull(core.workload);
7   | core.workload ← core.queue.pop();
8 end
9 if core.workload = None then
10  | core.workload ← maxQueueFromCores.pop();
11 end

```

Algorithm 3: Choosing the target core with a performance policy scheduling

Data: workload
Result: target core for the workload

```

1 if workload.canExecuteInAnyCore then
2   |   idlePartialBigCores = getIdlePartialBigCores();
3   |   if idlePartialBigCores is Not Empty then
4   |     |   targetCore = idlePartialBigCores.head();
5   |     |   return targetCore;
6   |   end
7 end
8 idleFullBigCores = getIdleFullBigCores();
9 if idleFullBigCores is Not Empty then
10  |   targetCore = idleFullBigCores.head();
11  |   return targetCore;
12 end
13 if workload.canExecuteInAnyCore then
14  |   idlePartialLittleCores = getIdlePartialLittleCores();
15  |   if idlePartialLittleCores is Not Empty then
16  |     |   targetCore = idlePartialLittleCores.head();
17  |     |   return targetCore;
18  |   end
19 end
20 idleFullLittleCores = getIdleFullLittleCores();
21 if idleFullLittleCores is Not Empty then
22  |   targetCore = idleFullLittleCores.head();
23  |   return targetCore;
24 end
25 return Empty;

```

Algorithm 4: Choosing the target core with an energy policy scheduling

Data: workload
Result: target core for the workload

```

1 if workload.canExecuteInAnyCore then
2   | idlePartialLittleCores = getIdlePartialLittleCores();
3   | if idlePartialLittleCores is Not Empty then
4   |   | targetCore = idlePartialLittleCores.head();
5   |   | return targetCore;
6   | end
7 end
8 idleFullLittleCores = getIdleFullLittleCores();
9 if idleFullLittleCores is Not Empty then
10  | targetCore = idleFullLittleCores.head();
11  | return targetCore;
12 end
13 if workload.canExecuteInAnyCore then
14  | idlePartialBigCores = getIdlePartialBigCores();
15  | if idlePartialBigCores is Not Empty then
16  |   | targetCore = idlePartialBigCores.head();
17  |   | return targetCore;
18  | end
19 end
20 idleFullBigCores = getIdleFullBigCores();
21 if idleFullBigCores is Not Empty then
22  | targetCore = idleFullBigCores.head();
23  | return targetCore;
24 end
25 return Empty;
  
```

3.3 PHISA in a COTS Processor

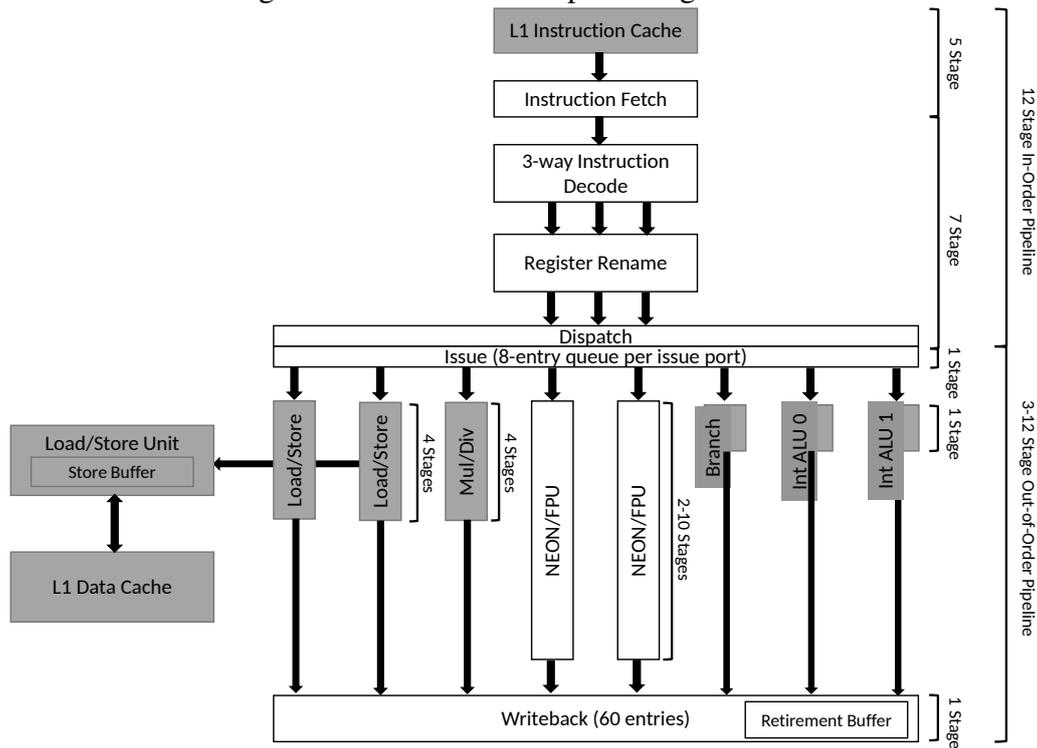
In order to evaluate the PHISA multicores system, we analyze real Commercial Off-The-Shelf (COTS) processors and design partial-ISA cores from them. We chose the A7 and A15 cores from the ARM Cortex-A family, as these are cores commonly used in heterogeneous systems, such as the ARM big.LITTLE. Although there are newer cores in this family (e.g., the A53 and A75), we choose the older ones due to the availability of information in the microarchitecture, area, and power of these cores. In ARM architectures, the SIMD and FP instructions are executed by the NEON unit. The NEON extensions are usually modular, simplifying its removal. These extensions are even considered optional in some ARM processor families, including the A7 and A15 cores. In other recent architectures, such as the RISC-V, ISA extensions are also designed to be modular to ease the process of customizing the processor.

Figure 3.4 shows a diagram of the Cortex A15 pipeline. The A15 is an OoO processor with a dual-issue NEON unit. In ARM architectures, the NEON unit is responsible for executing both the SIMD and FP instructions. The A15 NEON unit is capable of executing 128-bit wide words and has a RF of 32 64-bits wide registers (two registers are combined to process 128-bits words). The A7 processor, on the other hand, is a simple in-order core with a single-issue 64-bit wide NEON unit. Its RF is also smaller, with 16 64-bits registers.

To analyze the potential area and power reduction of a PHISA system using the A15 and A7 cores, we have modeled these same processors in McPAT (LI et al., 2009) using a node technology of $28nm$. Our models consider the entire core (including MMU and instruction and data L1 caches) without L2 caches. Although McPAT models its components according to an A9 processor, we have used an approach similar to the one proposed in (ENDO; COUROUSSÉ; CHARLES, 2015) to model the A7 and A15. The authors show that this approach results in models very close to the real processors. The McPAT models report an area of $0.5mm^2$ with a maximum Thermal Design Power (TDP) of about 53mW for the A7 core and $3.53mm^2$ and 700mW for the A15 core. Figure 3.5 shows the area break down per component in both these cores. As shown, the NEON unit related components are responsible for about 69% of the area in the A15 core and 26% in the A7 core.

In figure 3.4, the white boxes represent the processor components that can be simplified when removing NEON instructions in the A15 core. McPAT allows the modeling

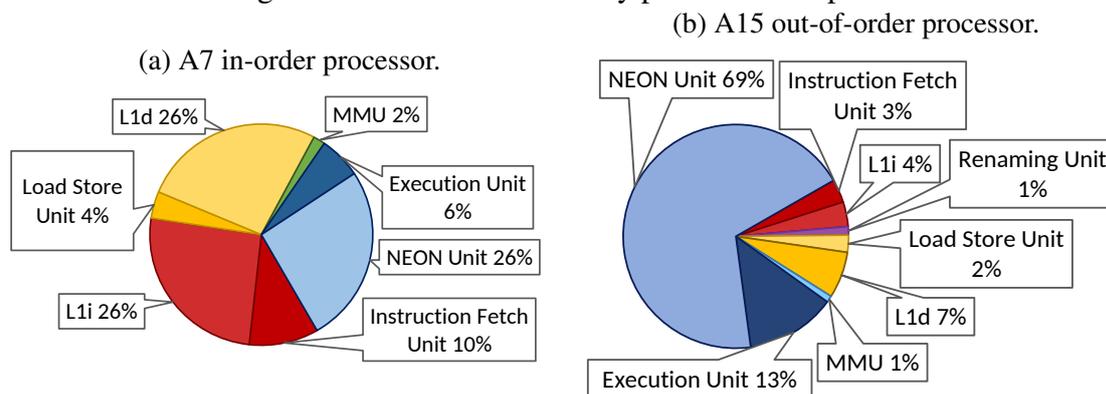
Figure 3.4: Cortex A15 Pipeline organization.



Source: Adapted from (HRUSKA, 2012)

of configurations without FP and SIMD units (by merely setting the FP related tags in the template to zero), which also triggers the exclusion of the FP instruction window, the FP Register File (RF) and the FP register renaming structures. This methodology results in a conservative model, as removing the NEON extension and all its hardware would also affect other structures, such as the instruction decoder and the clock tree (LEE et al., 2017). Thus, the McPAT model very likely represents a pessimistic view of the potential area reductions. By removing the NEON components of the A15 core, its area is reduced to about 1.17mm^2 and TDP to 590mW , while the A7 is reduced to 0.39mm^2 and 46mW .

Figure 3.5: Area breakdown by processor component.



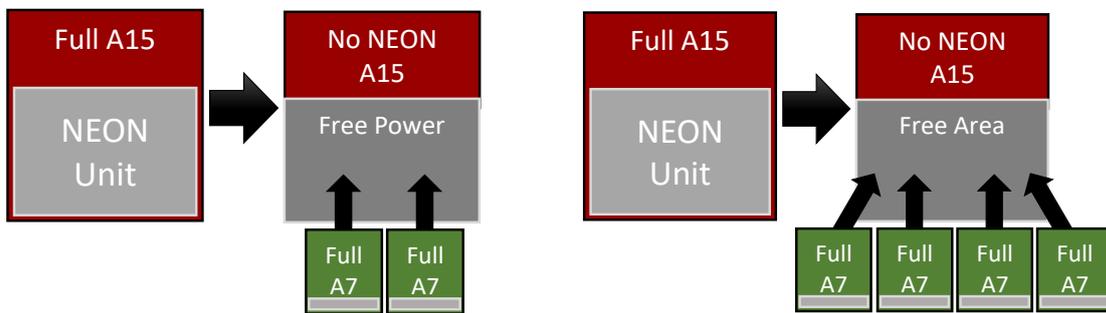
Source: The Author

Therefore, there are many design possibilities using the A15 and A7 to build PHISA systems. For example, if one wants to maintain a power budget in the system, a PHISA system with one partial-ISA A15 core and two full-ISA A7 cores has the same TDP as a single full-ISA A15 core. If low power is not a concern, but the area is, two additional full A7 cores can be added to this PHISA system (one partial A15 + four full A7 cores) in the same area as the single full A15 core. Figure 3.6 depicts these examples, which will be explored in the evaluation of this thesis.

Figure 3.6: PHISA system using A15 and A7 cores examples.

(a) Having the same power budget.

(b) Having the same area budget.



Source: The Author

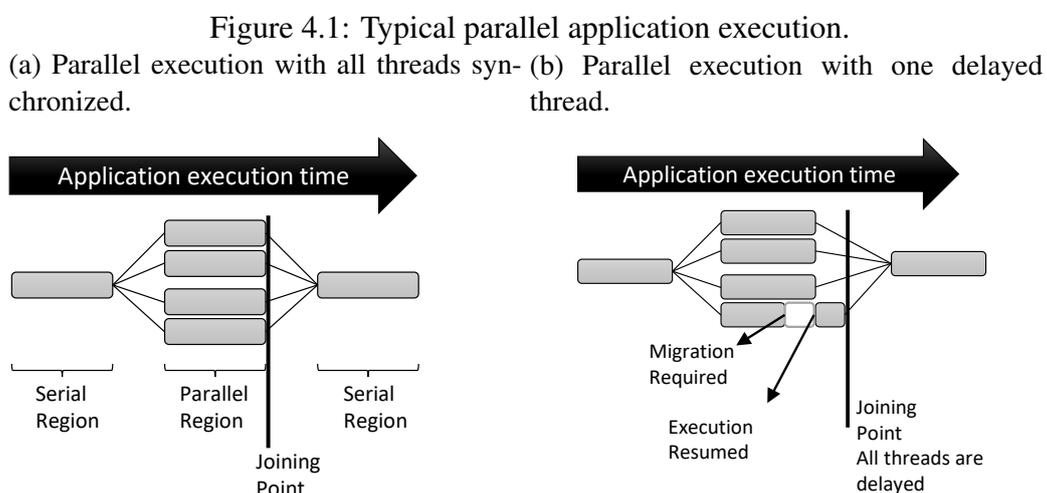
4 TUNE ARCHITECTURE

In this chapter, we present the **Tightly Coupled Instruction Offloader (TUNE)**. As described by its name, TUNE is an instruction offloader that can be coupled to PHISA systems to allow the optimization of multithreaded applications. We first motivate the usage of TUNE and present the concept of the offloader and how it is tightly coupled to a partial-ISA core. Then, we show how a PHISA with TUNE (TUNEd PHISA) system can be designed in a COTS processor. Lastly, we present a performance model that motivates the usage of a TUNEd PHISA system.

4.1 The Offloader

In chapter 3, we have discussed how a PHISA system can be used to optimize the area and power usage of single-ISA heterogeneous systems. To maintain software compatibility between partial- and full-ISA cores, the PHISA system presented used a fault-and-migrate strategy between the cores and the scheduler. For single-threaded applications, migrating threads during instruction faults is an efficient solution, as workloads are usually independent and do not produce bottlenecks between themselves. However, in a parallel application, if a single thread migrates while the others are executing, it will delay the time in which this thread will reach its joining point (e.g., a barrier or a synchronization point), thus creating a bottleneck that delays the entire application. Figure 4.1 depicts this situation.

In a PHISA system, the expensive SIMD/FP units are removed from cores to give



Source: The Author

space for additional cores. We have seen in chapter 3 that removing these units from big cores is usually more meaningful, liberating area and power for multiple extra little cores. In a parallel application environment, having these extra cores is essential for increasing the performance, as the multiple threads will be able to exploit the extra Thread-Level Parallelism (TLP). Furthermore, as discussed in chapter 2, previous works have shown that parallel applications can usually benefit more from many smaller cores than from fewer bigger cores, and that big cores are essential only to execute their serial regions (SULEMAN et al., 2009).

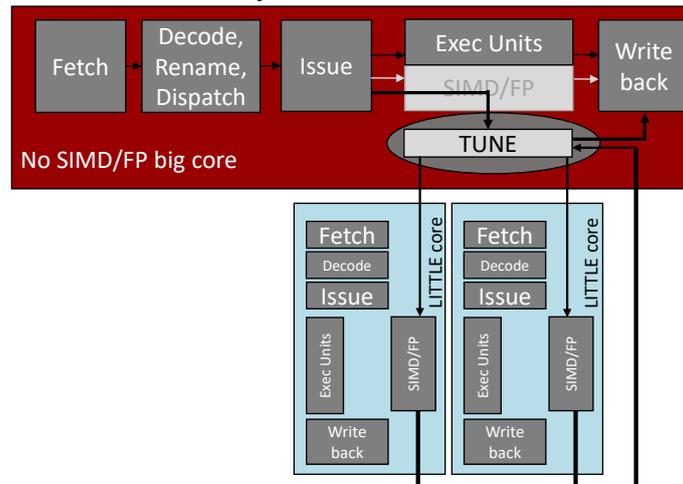
To increase even further our system motivation, we have observed in the characterization of several parallel applications (details in the following section 4.3, in which we model our system) that SIMD and FP operations are heavily concentrated in the parallel regions of these applications. This is naturally expected, as the SIMD instructions are used to exploit parallelism in vector operations, while the FP instructions are used in specific code normally present in kernels.

Therefore, we have applications in which the parallel regions benefit from having many cores, while the serial regions are accelerated in big cores and usually do not require SIMD or FP operations. This is a scenario in which evaluating the concepts of PHISA would be interesting, by creating a processor with a single partial-ISA core to execute serial regions and as many full-ISA little core as possible for the parallel regions. However, to allow serial regions to execute SIMD/FP instructions without migrating threads (executing serial regions in little cores can cause bottlenecks), we introduce the TUNE to our system.

With TUNE, the expensive SIMD/FP pipelines can be traded from the big core to make room for additional smaller cores. These operations are still supported in the big core through an offloader, which is responsible for sharing the SIMD/FP units of the smaller cores with the big core. Our approach for removing these structures follows a similar methodology as in (LEE et al., 2017), where the SIMD and FP instruction extensions (along with additional extensions) are removed from an A15 processor. However, in TUNE, **we only remove the functional units responsible for the SIMD/FP operations**, leaving the decoder, the renaming logic, the issue queue, and the register file unaltered. This results in a design that is easier to implement, keeps the binary compatibility, and simplifies the offloading strategy.

The design we propose is showed in figure 4.2. In the TUNEd PHISA processor, the big core sees the SIMD/FP units from the little cores as if being its own. When a

Figure 4.2: TUNEd PHISA system with SIMD/FP instruction offloading.



Source: The Author

SIMD/FP instruction is fetched, it is still decoded by the partial core, and its operands read from the FP register file in that core. However, after the issue stage, the operands and operator will be forwarded to the units in the full cores. After executing, the data is forwarded back to the partial core, where they will be written back (committed) normally. We use this design so that the flow of the partial core remains the same as if it had its own execution units. Thus, our design does not require extensive changes in the core organization.

The concept of our processor layout is similar to the conjoined-cores proposed by Kumar et al. in (KUMAR; JOUPPI; TULLSEN, 2004), in which the shared units are placed in the floorplan in a way that minimizes the distance from all cores. However, in our models and evaluations, we still consider that there is an extra latency of forwarding the operands from the partial- to the full-ISA cores and returning the computed data back. We consider this latency to be similar to the L2 cache latency, as it is also a commonly shared resource in multicore processors.

It is important to notice that in the TUNEd PHISA system, **the big core will only be used during serial regions**. Therefore, during the offloading process (which occurs only in serial regions), the little cores are always idle, which avoids resource concurrency for the NEON units. This thesis **does not** evaluate an asynchronous scheduler that can work out unbalanced threads to be accelerated in the big cores. Thus, **during parallel regions, only the little cores are active**.

4.2 TUNE in a COTS processor

In chapter 3, we have shown how to design the PHISA system in a ARM COTS processor. Now, we show how to extend this design to build a TUNEd PHISA. As already discussed, the feasibility of sharing the SIMD/FP unit has already been studied by Kumar et. al (KUMAR; JOUPPI; TULLSEN, 2004) and demonstrated in commercial designs like the AMD Bulldozer, that shares a single SIMD/FP unit between two integer modules, and the Sun Niagara (Ultrasparc T1), that shares an FP unit between eight cores.

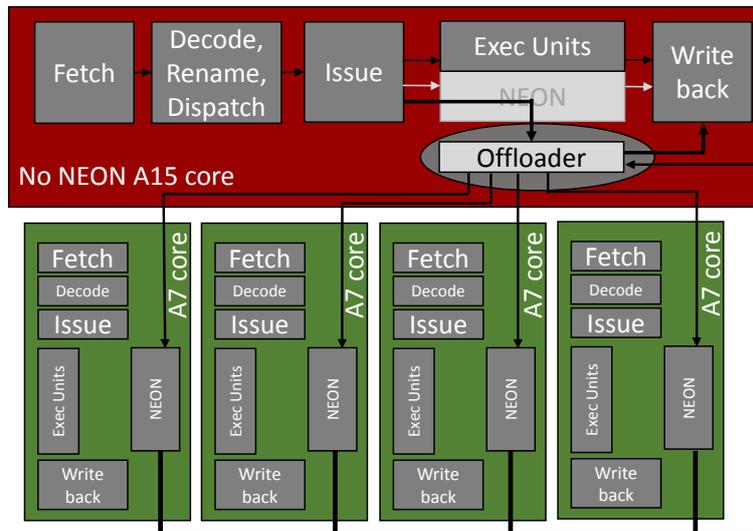
The idea of using a PHISA system in an ARM architecture is to replace the NEON units of some cores to add extra smaller cores. According to our area models, removing the NEON unit from the A15 core frees enough area to add up to four extra **full** A7 cores in the system.

It is important to notice that an A15 core will only offload instructions when executing a serial region of a parallel application. **During these execution phases, the A7 cores are mostly idle, which reduces the possibility of a resource conflict in the usage of the NEON units and keeps the system under the maximum TDP constraint.** Furthermore, we design TUNE as a transparent offloader that does not affect the flow of the processor pipeline: no other structures (such as renaming tables, register file, instruction scheduler) are changed, apart from the NEON execution lane.

Figure 4.3 shows an overview of how the ARM A15 and the A7 cores can share the SIMD/FP units using the TUNE offloading strategy. A traditional A15 core implements two 128-bit wide NEON issue lanes. In our approach, these lanes only are removed from the core, and an **Offloader** (TUNE) is implemented in its place as shown in the figure. NEON instructions fetched by the A15 follow the usual pipeline flow (decoding, renaming, dispatching) until the issue stage, in which the operation will be sent to the Offloader. This approach ensures that all cores in the system (including the reduced A15) will be able to decode NEON instructions, keeping the ISA compatibility in the entire system.

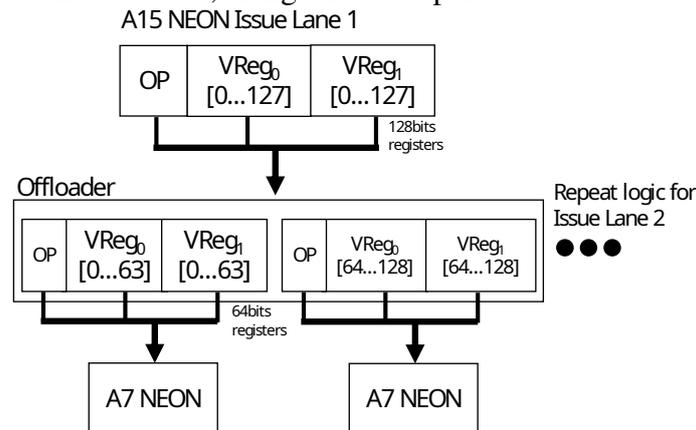
The Offloader is tightly coupled to the NEON units of the newly added full A7 cores, allowing direct sharing of these units with the A15 core. As instructions reach the Offloader ready to execute from the issue queues, there is no need for additional operation decoding or identification in its logic. The Offloader is a simple splitter circuit with buffers to keep the operating frequency, responsible for routing the operands (and the operation) of the SIMD/FP instruction from the vector registers of the big core to many NEON units

Figure 4.3: NEON instruction offloading from a A15 core to A7 cores. The NEON units from A7 cores are shared with the A15.



Source: The Author

Figure 4.4: The vector registers from the A15 core lanes are split into shorter registers and distributed over the A7 cores, along with the operation.



Source: The Author

in the little core. Figure 4.4 shows how the 128-bit wide vector registers of the A15 core are split into two 64-bit wide registers, which are then used to feed the A7 NEON units. Although both processors have vector registers of 64bits, the typical implementation of the A15 combines two registers to achieve operands of 128bits (MALLIA, 2007). When offloading instructions, the 128bits registers are split back into 64bits, as expected by the A7 NEON unit. By using this approach, each of the two 128bits NEON issue lanes in the A15 core will be tightly coupled to two other A7 cores (by their 64bits NEON units), offloading the same instruction for these two cores. Thus, the offloading is always done to a fixed pair of A7s per lane, avoiding any need for schedulers. If the given operation is a scalar FP instruction, then the Offloader will use a single A7 NEON unit (per lane).

As already discussed, the NEON design is usually decoupled from the rest of the processor. This allows for different power gating domains to be easily adapted for keeping only the NEON units powered on during the offloading process (KOPPANALIL et al., 2011). Thus, we expect the overhead in energy consumption of TUNE to be minimal, and mostly related to the routing wiring. As the A15 core offloads the same instruction to a pair of A7s with all its dependencies and operands resolved, both A7 cores will finish their operation in the same cycle. These operations will be redirected back to the Offloader, which recomposes the original vector register and proceeds to the writeback stage normally, updating its reservation station and re-order buffer - just as if it had the original NEON unit. **TUNE thereby keeps the original execution flow of the A15 core:** nothing changes in the perspective of the core design, except that now there is an extra latency for routing SIMD/FP operations.

4.3 TUNE Models

4.3.1 Performance Model

In order to understand the performance potential of our developed TUNEd PHISA design, we present an analytical model based on simple extensions to the widely known Amdahl's law (HILL; MARTY, 2008; AMDAHL, 1967), which express the potential speedup of applications given that a certain ratio of it is accelerated. In the context of multicore processors, in Amdahl's law, the accelerated part of a program is the parallel region, and the potential speedup is bounded by the number of cores available in the processor. Amdahl's law is given by equation 4.1, in which S is the system speedup, PR is the parallel region ratio (percentage of time spent by the application inside a parallel region), and N is the number of cores in the system.

$$S = \frac{1}{(1 - PR) + \frac{PR}{N}} \quad (4.1)$$

For our system model analysis, we consider that the processor is also composed of two distinct types of ARM cores, in which the parallel regions are executed in the small A7 cores. In contrast, the serial region is executed on the big A15 core. To model the performance impact of the two distinct core types, we use Pollack's rule (BORKAR, 2007), which states that the expected increase in the performance of a processor - due

to microarchitectural improvements - is roughly proportional to the square root of its growth in area. Conversely, if a micro-architecture has $2X$ less area (resources) than a second one, a performance drop of $\sqrt{2}$ is to be expected in the smaller one. Equation 4.2 represents the performance slowdown (LS) - thus, the inverse of Pollack's rule - of the A7 core in relation to the A15 core.

$$LS = \frac{1}{\sqrt{\frac{A15}{A7}}} \quad (4.2)$$

Furthermore, the big core used with TUNE trades SIMD and FP (NEON) instructions in favor of having more small cores, requiring to offload these operations when they are executed in serial region. Therefore, the cost of offloading these instructions must be accommodated in the performance model. We do that by separating from the serial ratio ($1 - PR$) the amount of ratio that is actually executing NEON instructions (SRF) and applying an overhead cost (OC). Our final model is described by equation 4.3, in which we apply the slowdown of the A7 cores and the offloading cost to Amdahl's law.

$$S = \frac{1}{(1 - PR) * ((SRF * OC) + (1 - SRF)) + \frac{PR}{N * LS}} \quad (4.3)$$

Summarizing, the parameters are: the speedup (S) normalized to a single big A15 core; SRF is the ratio of the Serial Region that executes SIMD/FP instructions; OC is the Offloading Cost; PR is the ratio of the Parallel Region; N is the number of little cores; and LS is the slow down caused by execution on the little cores.

4.3.2 Application Characterization

To feed data to our model, we characterize the regions of interest of several benchmarks of different classes from PARSEC (BIENIA, 2011), PARVEC (include vectorized versions of some applications from PARSEC) (CEBRIAN; JAHRE; NATVIG, 2015), SPLASH-2 (WOO et al., 1995) and PolybenchACC (GRAUER-GRAY et al., 2012). These benchmarks include parallel workloads optimized to use vector operations, which should stress the SIMD and FP unit of the processor cores. For the applications that are not explicitly optimized to use vector operations, we apply the GCC auto-vectorization framework (GNU, 2020) to produce vectorized code. Table 4.1 shows the list of applications, their ratio of Parallel Region (parameter PR in equation 4.3) and the ratio of

SIMD/FP execution time in the serial region (parameter SRF). For instance, 81.95% of the execution time in the *correlation* application is spent in the parallel region. In the remaining 18.05% spent executing the serial region, 10.10% is spent in executing SIMD/FP operations.

The data presented in table 4.1 is obtained by dynamically executing the applications and observing hardware counters that count cycles executing FP and SIMD operations, unhalted cycles spent executing multiple threads and total execution cycles. The table 4.1 shows that, in most of the applications, the parallel region is large enough so that the serial region does not introduce a big impact in performance. Nonetheless, there are some applications in which the parallel region does not cover most of the execution, and in these scenarios, the big core will be essential to reduce the serial region time. What is most interesting is that **even in applications with longer serial regions (such as polybench *lu* and *covariance*), the amount of SIMD/FP operations in the sequential part is still low,**

To maintain support for SIMD/FP operations in all cores, TUNE offloads these operations to the NEON units of the little cores whenever they occur during serial regions. In this performance model, we assume the operand routing overhead and the difference in operating frequencies to lead to an average slowdown of 2x. The SIMD/FP instructions issued during the parallel regions, on the other hand, are executed directly in the A7 cores and do not require offloading. Furthermore, using Pollack’s rule, and the area estimation models for the A15 and the A7 core (chapter 3, figure 3.5), the slowdown of the little core in relation to the big core (parameter LS in equation 4.3) is estimated to be $\frac{1}{2.64}x$.

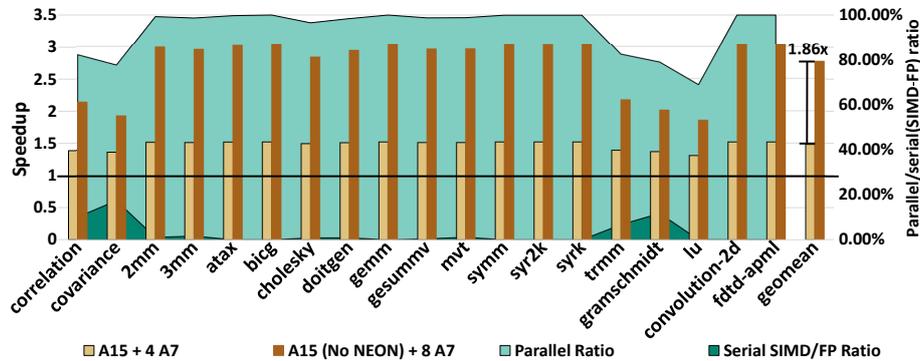
4.3.3 Discussion

To understand the potential gains of TUNE, we use the proposed model along with the discussed parameters to extract the performance of two AMCs with an equal area over a single fully-capable A15 core. The first AMC has a fully capable A15 core and 4 A7 cores and the second has a feature-light A15 core without NEON and 8 A7 cores. The results of this model can motivate the usage of TUNE in AMCs and give hints on its behaviour. Figure 6.8 shows the potential speedups for the polybench applications, based on the model specified by equation 4.3. We have chosen to show only the polybench applications, as they present better corner cases that are interesting to analyze, such as applications with long and medium parallel region ratios and high and low SIMD/FP

Table 4.1: Application **region of interest** characterization in terms of parallel region size and SIMD/FP ratio in the serial region.

	Benchmark	Parallel Ratio	Serial SIMD/FP Ratio
parsec	bodytrack	99.10%	0.05%
	ferret	98.24%	0.10%
	dedup	98.18%	0.00%
	facesim	97.56%	0.14%
	cholesky	96.56%	0.88%
	freqmine	95.38%	0.01%
	parvec	swaptions	99.99%
fluidanimate		99.67%	0.05%
streamcluster		99.60%	0.01%
canneal		99.58%	0.02%
blackscholes		99.55%	0.01%
vips		98.94%	0.14%
polybench		bicg	100.00%
	fdtd-apml	99.99%	0.00%
	convolution-2d	99.99%	0.00%
	gemm	99.97%	0.06%
	symm	99.95%	0.00%
	syrk	99.90%	0.04%
	syr2k	99.89%	0.04%
	atax	99.73%	0.06%
	2mm	99.27%	1.03%
	mvt	98.82%	1.11%
	gesummv	98.76%	0.46%
	3mm	98.68%	1.66%
	doitgen	98.41%	0.83%
	trmm	82.25%	6.74%
	correlation	81.95%	10.10%
	gramschmidt	78.70%	11.72%
	covariance	77.40%	17.33%
lu	68.71%	0.09%	
splash2x	ocean_ncp	99.85%	0.01%
	barnes	99.74%	0.02%
	ocean_cp	99.63%	0.04%
	radix	99.37%	0.00%
	raytrace	99.19%	0.12%
	lu_cb	98.48%	0.11%
	water_nsquared	97.92%	0.20%
	lu_ncb	97.86%	0.15%
	radiosity	97.81%	0.22%
	fft	97.38%	0.21%
	water_spatial	94.12%	0.66%
	cholesky	75.95%	2.77%

Figure 4.5: Potential speedup according to TUNE mechanism model. Bars represent the speedup over a single A15 core and the shaded areas are the % of PR and SRF of each application.



usage (most PARSEC/PARVEC and splash2x applications don't have significant serial regions). The bars in the figure represent the speedup (marked by the Y-axis on the left), while the shaded area in the background (marked by the Y-axis on the right) represents the fraction of the parallel region (PR) and the fraction of SIMD/FP operations in the serial region (SRF) in the application. The model shows that the traditional AMC (A15 + 4 A7) can provide a maximum speedup of about 1.5x in the applications with ratios of PR close to 100%. On the other hand, the TUNE AMC with additional cores (feature-light A15 with no NEON + 8 A7 cores) can provide a maximum potential speedup of 3x. This model clearly demonstrates that applications with a large parallel region will benefit from the additional cores while experiencing virtually no negative impact due to the absence of the SIMD/FP unit in the big core since the serial region is almost negligible. Applications with smaller PR fraction and a larger fraction of SIMD/FP operations in the serial region - such as *correlation*, *covariance* and *gramschmidt* - naturally provide smaller speedups with both the AMCs. Nonetheless, for all of the applications, the model shows a potential increase in the speedup of TUNE. This speedup can be observed even in applications with higher ratios of SRF , surpassing the extra cost of offloading instructions in the serial regions. Considering the geometric mean, TUNE is potentially 1.86x faster on average than the traditional AMC with NEON units in all cores.

We also apply the model to measure the scalability of some interesting configurations. For instance, table 4.2 shows a heatmap of a TUNE configuration (1 A15 core without NEON + 8 A7 cores) with a fixed offloading cost of 2x normalized by a single full A15. In the rows the ratio of the parallel region increases from the top to the bottom, while in the columns the amount of SIMD and FP operations increase from the left to the right. For instance, the first cell from the top left represents an application that has no parallel regions (is completely serial) and no SIMD or FP operations in the serial regions

Table 4.2: Model variation heatmap for the TUNE AMC (1 A15 without NEON + 8 full A7) with a fixed offloading cost of 2x normalized by a full A15 single core. In the rows the ratio of the parallel region increases by 10%, while the columns increase the percentage of SIMD and FP operations in the serial region.

		SIMD and FP ratio in the Serial Region										
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Parallel Region ratio	0%	1	0.909091	0.833333	0.769231	0.714286	0.666667	0.625	0.588235	0.555556	0.526316	0.5
	10%	1.071811	0.977517	0.898473	0.831255	0.773395	0.723066	0.678887	0.639795	0.604961	0.573723	0.545554
	20%	1.154734	1.057082	0.974659	0.904159	0.84317	0.789889	0.742942	0.701262	0.664011	0.630517	0.60024
	30%	1.251564	1.150748	1.064963	0.99108	0.926784	0.870322	0.820345	0.775795	0.735835	0.69979	0.667111
	40%	1.36612	1.262626	1.173709	1.096491	1.028807	0.968992	0.915751	0.868056	0.825083	0.786164	0.750751
	50%	1.503759	1.398601	1.30719	1.226994	1.156069	1.092896	1.036269	0.985222	0.938967	0.896861	0.858369
	60%	1.672241	1.567398	1.474926	1.392758	1.319261	1.253133	1.193317	1.138952	1.089325	1.043841	1.002004
	70%	1.883239	1.782531	1.692047	1.610306	1.536098	1.468429	1.40647	1.349528	1.297017	1.248439	1.203369
	80%	2.155172	2.066116	1.984127	1.908397	1.838235	1.77305	1.712329	1.655629	1.602564	1.552795	1.506024
	90%	2.518892	2.457002	2.398082	2.34192	2.28833	2.237136	2.188184	2.141328	2.096436	2.053388	2.012072
	100%	3.030303	3.030303	3.030303	3.030303	3.030303	3.030303	3.030303	3.030303	3.030303	3.030303	3.030303

(in this case, in the entire application). As the values are normalized by a single A15, the performance in this scenario is 1, i.e., equal to the baseline. In this map we observe that parallel applications with parallel region coverage as low as 10% can already present some degree of benefits with the extra A7 cores, as long as there is no SIMD and FP operations in the serial region. However, the system presents much better performance when the parallel coverage is high (as expected in parallel applications), even when the amount of SIMD and FP in the serial region is also high.

An interesting configuration to evaluate in the model - and verify in which characteristics an application must have to be interesting in an AMC - is the homogeneous manycore processor composed of 16 A7 full cores (no A15 in this case). Because of the small area footprint of the A7, this configuration has almost the same area as the TUNE AMC in table 4.2. However, this configuration cannot accelerate serial regions (there is no big core), so we expect it to have worst performance in applications with smaller parallel regions. Table 4.3 shows the results for this configuration, using the same organization as table 4.2, and also normalized by a single A15 core. As this system does not have a partial core, it does not require instruction offloading, and increasing the amount of SIMD and FP in the serial region of the application does not influence performance. However, because an A7 core has to be used to execute the serial region, the system will only show better performance than the single A15 core when the parallel region covers at least 70% of the application. If we compare the results of the manycore system against the TUNE AMC, we will observe that the manycore system will only have better performance than TUNE if the application has a parallel region larger than 90% and more than 10% SIMD and FP operations in the serial region. This is an interesting result, as it shows that TUNE should be a better solution for applications that either (1) have small parallel regions (over 10%

Table 4.3: Model variation heatmap for a manycore system with 16 full A7 cores, normalized by a full A15 single core. In the rows the ratio of the parallel region increases by 10%, while the columns increase the percentage of SIMD and FP operations in the serial region (no effect in this scenario).

		SIMD and FP ratio in the Serial Region										
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Parallel Region ratio	0%	0.378788	0.378788	0.378788	0.378788	0.378788	0.378788	0.378788	0.378788	0.378788	0.378788	0.378788
	10%	0.417973	0.417973	0.417973	0.417973	0.417973	0.417973	0.417973	0.417973	0.417973	0.417973	0.417973
	20%	0.4662	0.4662	0.4662	0.4662	0.4662	0.4662	0.4662	0.4662	0.4662	0.4662	0.4662
	30%	0.527009	0.527009	0.527009	0.527009	0.527009	0.527009	0.527009	0.527009	0.527009	0.527009	0.527009
	40%	0.606061	0.606061	0.606061	0.606061	0.606061	0.606061	0.606061	0.606061	0.606061	0.606061	0.606061
	50%	0.713012	0.713012	0.713012	0.713012	0.713012	0.713012	0.713012	0.713012	0.713012	0.713012	0.713012
	60%	0.865801	0.865801	0.865801	0.865801	0.865801	0.865801	0.865801	0.865801	0.865801	0.865801	0.865801
	70%	1.101928	1.101928	1.101928	1.101928	1.101928	1.101928	1.101928	1.101928	1.101928	1.101928	1.101928
	80%	1.515152	1.515152	1.515152	1.515152	1.515152	1.515152	1.515152	1.515152	1.515152	1.515152	1.515152
	90%	2.424242	2.424242	2.424242	2.424242	2.424242	2.424242	2.424242	2.424242	2.424242	2.424242	2.424242
	100%	6.060606	6.060606	6.060606	6.060606	6.060606	6.060606	6.060606	6.060606	6.060606	6.060606	6.060606

and less than 60%) and low SIMD and FP usage in the serial regions or (2) have medium parallel regions (over 60% and less than 80%) and virtually *any amount of* SIMD and FP, or (3) high parallel usage (90%) but less than 10% SIMD and FP usage in the serial region.

Finally, we also show in table 4.4 the behavior of the TUNE AMC when the application has a fixed parallel region ratio of 80%, increasing amount (left to right) of SIMD and FP inside the 20% serial region ratio, but also an increasing cost to offload these instruction (top to bottom). We increase the cost in a factor of 2. We see that, as long as the amount of SIMD and FP in the serial region is kept low (between 10% and 20%), the cost of executing these instructions can be 16x higher through the offloader than through the normal hardware. In fact, as we see in table 4.1, all the characterized applications are inside this range of SIMD and FP usage.

Nonetheless highlight that the aforementioned model makes several assumptions and has limitations. About the application and the underlying architecture, the model assumes (1) linear scalability of the parallel region, (2) the performance difference of the distinct cores will be bounded by Pollack’s rule and (3) the fixed 2X penalty for offload-

Table 4.4: Model variation heatmap for the TUNE AMC (1 A15 without NEON + 8 full A7) with a fixed parallel region ratio of 80% normalized by a full A15 single core. In the rows, cost of offloading instructions increases by a power of 2, while the columns increase the percentage of SIMD and FP operations in the serial region.

		SIMD and FP ratio in the Serial Region										
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Offloading Cost (x times more cycles)	2	2.155172	2.066116	1.984127	1.908397	1.838235	1.77305	1.712329	1.655629	1.602564	1.552795	1.506024
	4	2.155172	1.908397	1.712329	1.552795	1.420455	1.308901	1.213592	1.131222	1.059322	0.996016	0.93985
	8	2.155172	1.655629	1.344086	1.131222	0.976563	0.859107	0.766871	0.692521	0.631313	0.580046	0.536481
	16	2.155172	1.308901	0.93985	0.733138	0.600962	0.509165	0.441696	0.390016	0.349162	0.316056	0.288684
	32	2.155172	0.922509	0.586854	0.430293	0.339674	0.280584	0.239006	0.20816	0.184366	0.165453	0.15006
	64	2.155172	0.580046	0.335121	0.235627	0.181686	0.147842	0.124626	0.107712	0.094841	0.084717	0.076546
	1024	2.155172	0.047792	0.024164	0.01617	0.01215	0.009731	0.008115	0.00696	0.006092	0.005417	0.004877

ing the SIMD/FP operations. Therefore, there are static values that do not represent the real behaviour of the applications, and that are useful for this high-level and initial analysis. Although simplified, the performance model provides insights about the potential of TUNE architecture over a traditional AMC. Our evaluation in chapter 6 shows that the projections of the model are consistent with the results obtained from simulations.

5 SIMULATION METHODOLOGY

In this chapter, we briefly describe the architectural simulators used to evaluate this thesis. We describe two different simulators: gem5 and PHISA Simulator. The former is an open source, community developed, full system simulator, while the latter is a scheduling simulator for PHISA systems. We discuss each of the simulators and how they were integrated to produce this thesis results.

5.1 Gem5

Gem5 is a modular discrete event driven computer system simulator platform (LOWE-POWER, 2020). Its components can be rearranged, parameterized, extended or easily replaced to suit the designer's needs. It simulates the passing of time in the system as a series of discrete events and its intended use is to simulate one or more computer systems in various ways. Gem5 is more than just a simulator; it's a simulator platform that allows the designer to use as many of its premade components as needed to build up a custom simulation system.

Gem5 is written primarily in C++ and python and most components are provided under a BSD style license. It can simulate *a complete system with devices and an operating system* in full system mode (**FS mode**), or *user space only programs* where system services are provided directly by the simulator in syscall emulation mode (**SE mode**). There are varying levels of support for executing Alpha, ARM, MIPS, Power, SPARC, RISC-V, and 64 bit x86 binaries on different CPU models. Gem5 supports two simple single CPI models, an out of order model, and an in order pipelined model. A memory system can be flexibly built out of caches and crossbars or the Ruby simulator which provides even more flexible memory system modeling.

In this thesis, we have used both the System-call Emulation (SE) and Full System (FS) modes of gem5. To evaluate the PHISA system, which is aimed to run single-threaded workloads, we have used gem5 in the SE mode. The SE mode is preferable to use when one does not want to include OS overheads and is generally suggested for running single-threaded applications. As this mode does not include support for multiple core communication, we have also created a scheduling simulator to support PHISA. We further discuss this simulator in the following sections.

On the other hand, in the TUNEd PHISA, we aim to optimize multi-threaded

applications, which require OS support to create and manage threads. The TUNEd PHISA system was completely modeled using gem5 in the FS mode, and, as no scheduling of threads is required, no extra simulators or modifications were required.

5.2 McPAT

According to the HP Labs(HPLABS, 2009), McPAT (Multicore Power, Area, and Timing) (LI et al., 2009) is an integrated power, area, and timing modeling framework for multithreaded, multicore, and manycore architectures. It models power, area, and timing simultaneously and consistently and supports comprehensive early stage design space exploration for multicore and manycore processor configurations ranging from 90nm to 22nm and beyond. McPAT includes models for the components of a complete chip multiprocessor, including in-order and out-of-order processor cores, networks-on-chip, shared caches, and integrated memory controllers. McPAT models timing, area, and dynamic, short-circuit, and leakage power for each of the device types forecast in the ITRS roadmap including bulk CMOS, SOI, and double-gate transistors. McPAT has a flexible XML interface to facilitate its use with different performance simulators.

In this thesis, we have used McPAT version 1.3 to model designs of the A15 and A7 cores using a node technology of $28nm$. Our models consider the entire core (including MMU and instruction and data L1 caches) without L2 caches. Although McPAT models its components according to an A9 processor, we have used an approach similar to the one proposed in (ENDO; COUROUSSÉ; CHARLES, 2015) to model the A7 and A15. The authors use strategies such as Pollack’s rule(BORKAR, 2007) to scale the area and power of the components, and show that this approach results in models very close to the real processors.

Through the flexible XML interface, it is simple to use McPAT to model processors without SIMD and FP support. Configuring the XML file to implement these instructions triggers the exclusion of the FP instruction window, the FP Register File (RF), the FP register renaming structures, and the SIMD/FP execution lanes. When we evaluate the PHISA system, we remove all these components. However, in the TUNEd PHISA system, most of these structures **are still needed for decoding and offloading instructions**. Thus, we modify McPAT to only remove the SIMD/FP execution lanes in this case.

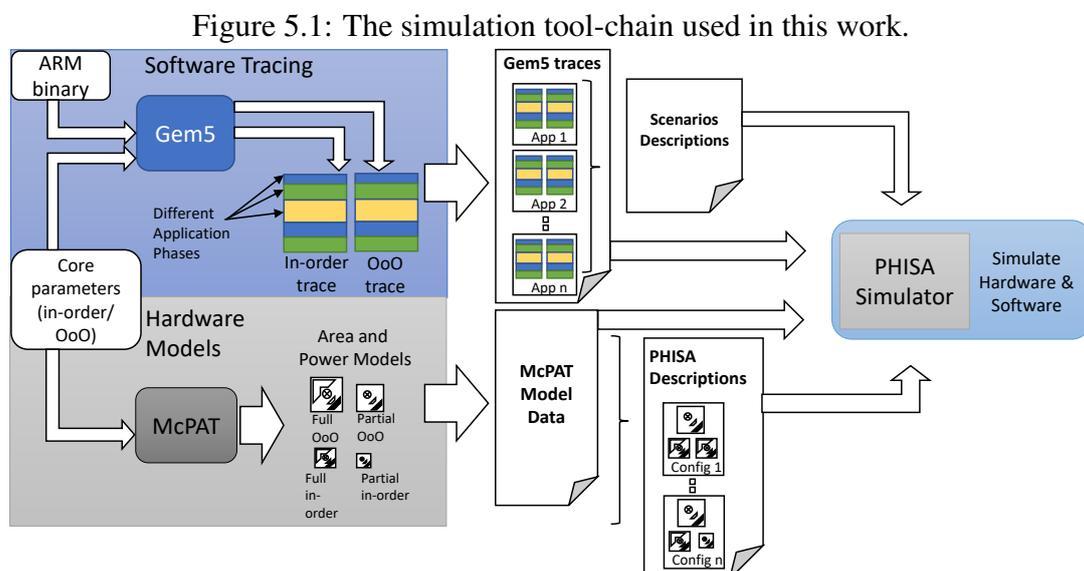
5.3 PHISA Simulator

Since the partial-ISA strategy used by PHISA requires modifications in both the hardware and the task management (e.g., OS) and we do not have such a real-life system, we perform our analysis in a simulated environment. In this section, we overview how the simulation environment is set up and how we combine data from different tools to evaluate our designs. The contents of this section have been adapted from (BECKER, 2019), which used the same simulator.

Figure 5.1 presents a high-level diagram of the simulation tool-chain we describe in this section. In the following subsections, we will refer to this diagram, explaining all the necessary steps to simulate the execution of multiple applications on a PHISA system. Namely, we detail how we collect workloads execution behavior, how we combine these with area and power data from McPAT to build our partial-ISA cores, and how we set up important configurations for experimentation.

5.3.1 Profiling and tracing workloads execution phases

As we discussed in Chapter 3, the PHISA system execution flow depends on a few hardware triggers for proper functioning. For example, the cores must notify the *Scheduler* when there are workload dependencies for an ISA-extension datapath (as a FP or SIMD operation). Thereby, it is mandatory to have those triggers information to simu-



Source: Adapted from (BECKER, 2019)

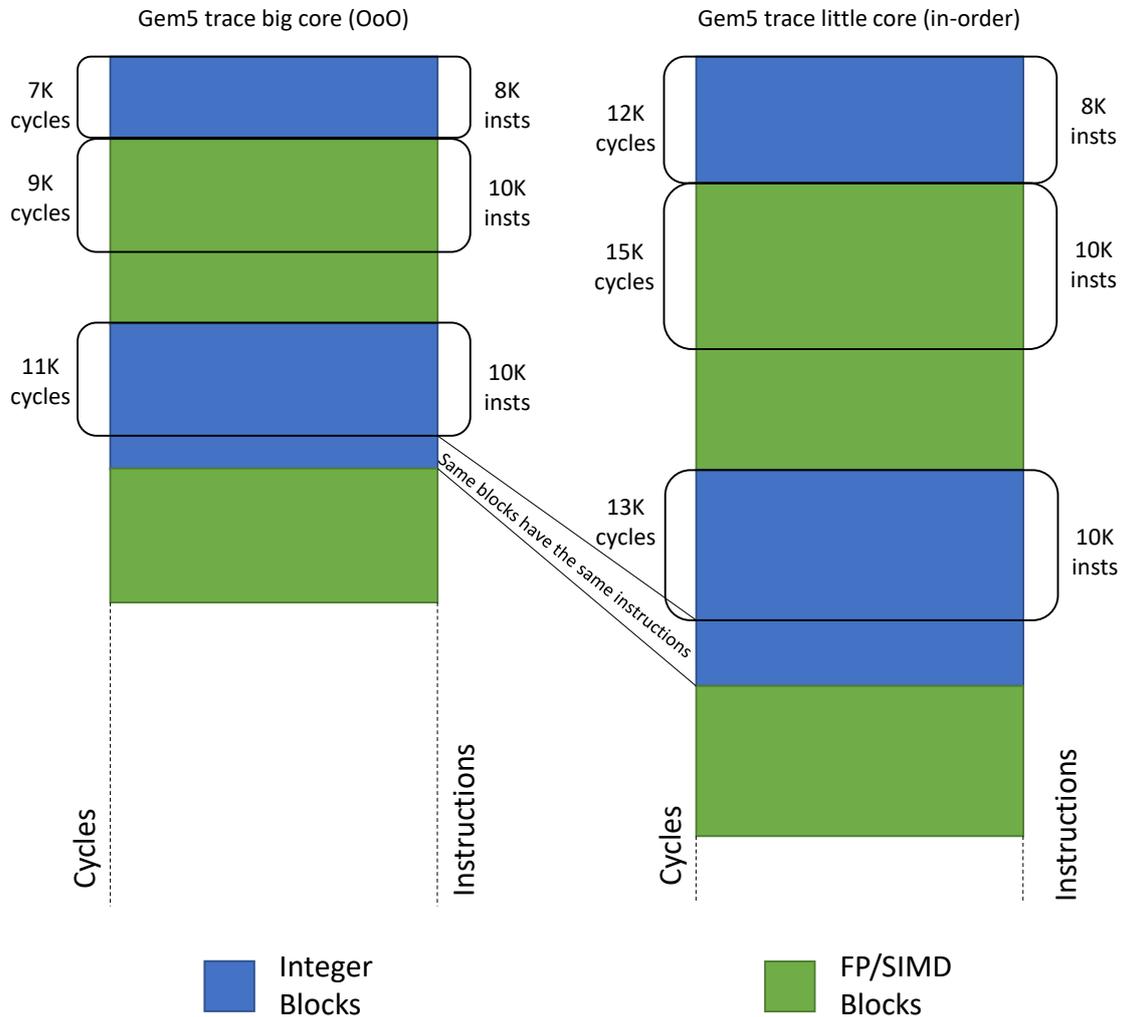
late task migrations in an accurate manner. Also, because the system has heterogeneous cores, and workloads can execute on any of them, our simulator must address the different performance that applications have in the different cores available. For example, a workload can execute quickly on an OoO but slowly on an in-order core. Therefore, these previous observations suggest for us to build our simulated environment using dynamic instrumentation.

To collect the behavior of the workloads under different host cores, we use the *gem5* cycle-accurate microarchitecture simulator, previously presented. The *gem5* is capable of executing binary code from different ISAs, including the ARM used in this work. It can be used as a tool for measuring stats of applications (like the execution time, the number of committed instructions, the L1 cache miss-rate) in different organizations. More importantly, its open-source code models cycle-accurate in-order and out-of-order cores and allows internal modifications for detailed on-the-fly execution profiling.

For our purposes, we have modified the *gem5* simulator so it traces the execution of ARM applications while executing workloads on both OoO (big) and in-order (little) cores. This appears on the upper part of Figure 5.1. As it appears in the Figure, the traces created during *gem5* profiling hold information regarding different phases of the workload's execution. Particularly, these phases will be consulted by our simulated *PHISA Simulator*, so it knows which portions of the program require extensions support (to mark applications as ISA-extension dependent during simulation). When executing the applications, *gem5* dynamically retrieves the executing instruction, and annotates when a unsupported operation is executed. Furthermore, we annotate a new phase whenever the program executes for longer than 10K cycles without extension instructions. This creates fine-grained phases that allow our custom simulator to schedule threads efficiently. With the aforementioned instrumentation, *gem5* traces contain dynamic information of the hardware triggers we expect from cores in our PHISA design.

To understand how the profiling traces are internally created and how we leverage their content information, Figure 5.2 depicts two traces for a given workload: one for its execution in a big core (OoO), and another for its execution in a little core (in-order). Each trace is composed of a set of blocks, representing the intervals of the application's execution. As the legend of the Figure describes, these blocks represent the intervals of the application with integer-only instructions and with ISA-extensions instructions (FP and SIMD in this work). Importantly, the blocks hold the number of cycles and instructions these execution intervals took to perform, depending on the host core.

Figure 5.2: Representation of the execution traces of a workload in both a big and a little core, generated with the gem5 simulator. The current host core of the workload determines which of the traces will be consumed for a given slice of the execution.



Source: Adapted from (BECKER, 2019)

To generate the traces, gem5 counts the number of committed instructions from the beginning to the end of an execution interval, on both OoO and in-order simulations. The blocks usually represent up to 10K instructions¹, and are successively reported covering the whole execution of the application. Blocks can be shortened, however, when an ISA extension is fetched, which closes an integer block and immediately starts a FP/SIMD block. This is depicted at the end of the first block of both traces in Figure 5.2. When gem5 counts more than 10K non-extension instructions, it closes the FP/SIMD block and starts a new integer block, emulating how a core would trigger the removal of the ISA-dependency from an application in our PHISA system.

¹The size of the blocks must be small for fine-grain representation of the execution phases, but not too small causing traces to be composed of many of them, turning the traces into big files (which also overloads the traces parsing in our simulator, as we explain in section 5.3.3).

Importantly, we guarantee the blocks represent the same portion of a program execution regardless of the host core, i.e., the number of total blocks and the instructions they represent are the same in both OoO and in-order traces. This appears in Figure 5.2, observing the block's amount of instructions, depicted on the right side of the blocks in the traces. Since both microarchitectures commit instructions in order, the N th committed instruction of a given application is the *same* for both cores. Since our modified gem5 counts instructions, its internal counters will be incremented symmetrically in both in-order and OoO simulations, for a given workload. Hence, the K th block in the big core trace is equivalent to the K th block in the little core trace. This is important because it assures that, at the end of each block, the workload is at the same point of execution in both traces. Thereby, our simulator can read the big core trace up to a point, and continue reading from that point ahead in the little core trace, which allows us to simulate migrations coherently, as we detail further (section 5.3.3).

What can (and generally will) vary is the amount of time it takes to execute a block depending on the host core. For example, big cores can achieve higher ILP exploitation to commit the instructions of a block faster than the little core. We also illustrate this in Figure 5.2, presenting the cycles for executing the blocks (varying with the core). This is used by our simulator to extract the performance difference among the different cores.

The final pairs of traces for each application (one for the in-order model, and the other for the OoO) are then grouped in different scenarios. In our evaluations, we use a set of applications to describe typical scenarios of user softwares, such as image processing and video decoding. These descriptions, along with the application traces themselves, will be used as inputs for our PHISA Simulator.

5.3.2 Modeling area and power using McPAT

We use McPAT to extract area and power data from our processors. There are six cores that are modeled in this thesis. First, the full A15 core, which is modeled as an embedded OoO core, and the full A7 core, which is an embedded in-order core. Then we have the partials A15 and A7 for PHISA, in which we remove all the components related to the NEON units (see section 5.2) from both cores. Finally, we have the partials A15 and A7 for the TUNEd PHISA, in which only the NEON execution lanes are removed, as the other components are still required for fetching, decoding and offloading these instructions.

McPAT produces an output with a breakdown on the area and power information of all the modeled components in the core. Therefore, it is possible to measure these metrics individually and analyze which components have a higher impact in the system. We use the area and power data to build our configurations according to some budget (either a maximum area in the layout or a maximum TDP). This is shown in figure 5.1 as the PHISA Descriptions, which are the different cores that compose each of the configurations we evaluate. The power data is also used by the PHISA simulator to determine how much energy an application consumes during execution. This is shown in figure 5.1 as the McPAT Model Data diagram, which is also fed to the PHISA Simulator.

5.3.3 Modeling the *PHISA Simulator* for a multi-task simulation

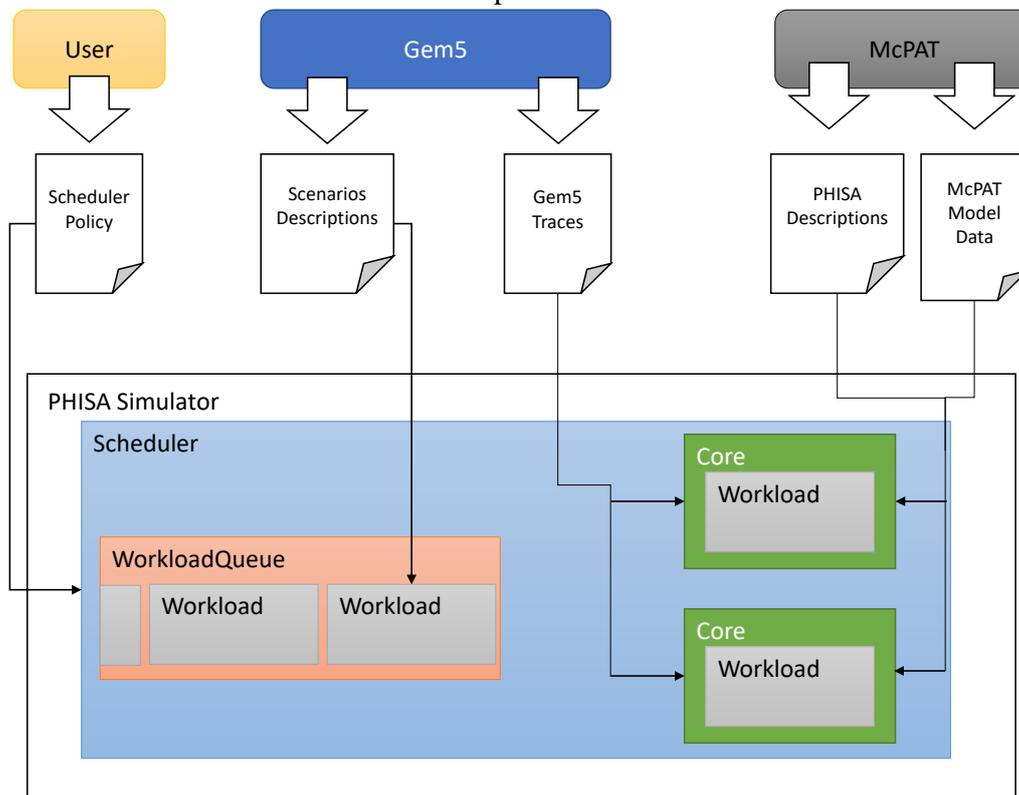
Although we execute every workload in gem5 to generate its execution traces, it is also necessary to consider its execution in a multi-task environment, where multiple workloads share the system resources. For that end, we developed an in-house simulator to model the *PHISA Simulator* scheduler policies (as described in section 3.2). Figure 5.1 presented a comprehensive overview of the tools and simulation flow. Through McPAT models of the OoO and in-order cores, we provide hardware-related data to the *PHISA Simulator*, so it has information regarding the power of the computing cores it is simulating. At the same time, through the steps described in Section 5.3.1, we provide software-related data from gem5 to the *PHISA Simulator*, so it has the executions traces of the applications to properly simulate the execution of the workloads under the available hardware. Internally, the *PHISA Simulator* simulates the OS *Scheduler* (with a policy chosen by the user), the system *Cores*, and all auxiliary modules to perform accurate simulation, as we describe next. With this, we can verify the impact of having partial-ISA cores, in heterogeneous designs.

Figure 5.3 presents an overview of the simulator components and inputs. As in the Figure, the *PHISA Simulator* module wraps the *Scheduler*. The *Scheduler*, in turn, has a list of workloads (in the WorkloadQueue) and the reference for available cores in the system. Note that workloads can be in the queue, or executing in a computing node. The mapping strategy to assign a workload from the queue to an idle core is implemented accordingly with the chosen scheduling policy, detailed in section 3.2.

The diagram illustrated in Figure 5.3 also presents (on the top) the necessary inputs for accurate experimentation. They are used in the following manner:

- **Scheduler Policy.** The policy is defined at the initialization of the simulator by the user. With this input, it is possible to choose between the naive, performance-oriented or the energy-oriented mapping strategies described in section 3.2. Also, the existence of this input allows the simulator to extend the number of policies while keeping an easy interface. We detail how the scheduler interacts with the remaining modules of our simulator further in this section.
- **Scenarios Descriptions.** This input contains the list of workloads we want to execute in the simulator, allowing us to create multi-tasking execution environments. The list of workloads is used to fill the *WorkloadQueue* when the simulator starts the execution. This input is simply a *json* file which holds the workloads names, and the reference for the trace files from gem5 (another input of the simulator as we detail further). With this interface, we can quickly create different scenarios for experimentation.
- **PHISA Descriptions.** This input is a *json* file containing each core in the system and its microarchitecture, since it is important to inform the simulator of the

Figure 5.3: A high-level view of the PHISA scheduler simulator. On the top, the set of inputs necessary for execution. Above, the different modules of the simulator and their interaction with each other and with the inputs.



Source: Adapted from (BECKER, 2019)

computing nodes available. Especially, the *Scheduler* may leverage this information when applying its current policy. The json file defines if the core is an OoO or an in-order core, and the ISA-extensions it supports (or does not support, if partial-ISA). Configuring the elements in the *json* file is all it takes for adjusting the composition of the PHISA system.

- **McPAT Model Data.** We use the McPAT outcome to feed our simulator with power information. With this, the simulator can have the mean power of each computing node at hand. We sum up the energy consumed by each core along with the execution of the workloads using data from the *PHISA Simulator*, which monitors whether or not the cores are executing.
- **gem5 Traces.** As previously explained, gem5 traces are used as a trustful representation of a workload executing in a core. In our environment, a workload can execute in either big or little cores (OoO or in-order), or accelerators. Because of this, for each workload we want to simulate, we need (and have) two gem5 traces. One contains the execution trace for the workload under an OoO core, and the other has the execution trace of the workload in an in-order core. Depending on the workloads' host core (decided on-the-fly by the *Scheduler* during our simulation), the appropriate trace will be consulted to advance the workloads' execution accordingly.

The execution of workloads in our simulation proceeds after we have these inputs at hand. With the *WorkloadQueue* filled with workloads, the *Scheduler* assigns tasks for the available cores, accordingly with the chosen scheduling policy, following the specifications in Chapter 3. For such, we carefully assure all the scheduler restrictions defined in Section 3.2 are respected in our implementation. A *Core* will execute a *Workload* block as soon as it is assigned. For this, the host *Core* module verifies its type (big or little). Based on that, the *Core* looks up in the corresponding trace (big core trace or little core trace, depending on its type) and gathers a block, checking its execution interval. This is used to append execution time in the total time of the workload's execution, and to add up the energy consumption for the block execution on that core. Also, it is used by the *Scheduler* to know what cores are occupied or idle at a given time. At the end of the block, the *Scheduler* verifies for how many cycles the workload has been executing on the core. If it surpassed the threshold of 160K (see section 3.2), the workload is preempted from the core and pushed to the *Workloads Queue* module.

If the workload is not preempted, the simulated *Core* looks up for the next block.

If the next block has different dependency than the previous block (e.g., it is marked as *ISA-extension dependent*), the core check if it is capable of executing it. If it is, the same process as above is repeated; if it is not capable of executing the block, the workload will either be preempted (removed from the core and pushed to the *Workloads Queue*), or emulated. If the core is able to emulate the requested instruction, the block will be executed in that core as usual, but paying a multiplicative cost of emulation. For instance, if a ISA-dependent block takes N cycles to execute in real hardware, the simulator will account $N * emulation_cost$ cycles in the emulation scenario (see section 3.2 for emulation cost estimation).

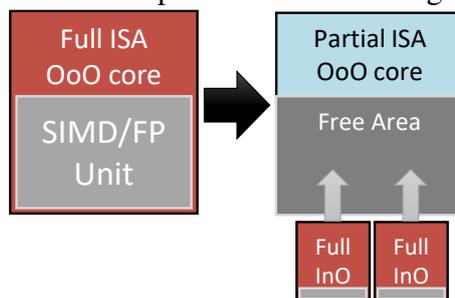
6 EVALUATION

In this chapter, we evaluate several PHISA systems designs to understand their benefits and limitations. We start by evaluating the system under single-threaded workloads, maintaining ISA compatibility between cores using thread migration. We first analyze the system using a naive proof-of-concept scheduler, and then we explore different scheduling policies. In the second section, we analyze the TUNEd PHISA system using multi-threaded applications. For both systems, we present the particularities in methodologies that we used for evaluation. Finally, we also compare both systems to other state-of-the-art solution.

6.1 Evaluation of PHISA with single-threaded workloads

In this section, we evaluate the PHISA system under single-threaded applications. In this system, we aim to remove support for SIMD and FP operation, to increase the number of cores in the system (see figure 6.1), and potentially increase its workload throughput. We first present the methodology used to model and simulate the system. Then, we discuss the benchmarks used and how we build scenarios using them. After that, we present the different system configurations that we have build, aiming to explore different design spaces. Later, we present the results for all experiments, along with discussions for each scenario, using the naive scheduling policy as a proof-of-concept. We then discuss results for specific policies optimizing performance and energy.

Figure 6.1: Example of a PHISA configuration.



6.1.1 Evaluation Methodology

Modeling and Simulation Parameters: We have used the gem5 simulator (BINKERT et al., 2011) to model the different versions of the ARM’s A7 and A15 processors and trace the applications’ execution. In this evaluation, our partial-ISA cores are configured to remove all components related to the NEON units, such as register files, issue queues and execution lanes. Details on how we use this data are shown in chapter 5.

Workload and Scheduling: Our workload set uses applications from different sources such as embedded applications from Mibench (GUTHAUS et al., 2001), media processing from Mediabench (FRITTS et al., 2009), linear-algebra, and data mining from Polybench (POUCHET, 2019) and IoT applications from Locus (TAN et al., 2017) as listed in Figure 1.1. We aim to simulate traditional but assorted scenarios, and mainly representative use case scenarios for edge computing. We assume scenarios in which the applications run either completely in parallel or in a pipeline manner - applications can output partial results to feed the input of the next benchmark. These scenarios are illustrated in table 6.1, in which the column ‘Task’ briefly describes the goal of the scenario, column ‘Workloads’ lists the benchmarks executed, column ‘Exec Mode’ specify if the scenario runs in pipeline or in parallel and ‘% NEON’ shows the percentage of dynamic NEON operations executed.

In scenario 1, we include a series of image filters and kernel operations that represent an image processing application. Scenario 2 and 3 include opensource libraries for encoding and decoding videos, along with kernels that represent data transmission (FFT and FFT-i), cryptography (AES), and redundancy and fault tolerance checks (CRC32)(ADEGBIJA et al., 2018). These latter kernels are also used in scenario 4 - a

Table 6.1: Workloads in each scenario.

	Task	Workloads	Exec Mode	% NEON
Scenario 1	Image processing	Susan (smooth, edges, corners); 2dconv; histogram; reg-detect; libpng; aes; CRC32; FFT	pipeline	0.5%
Scenario 2	Video encoding	FFT-i; libav-enc; aes; CRC32; FFT	pipeline	3.34%
Scenario 3	Video decoding	FFT-i; aes; h264-dec; CRC32; FFT	pipeline	0.04%
Scenario 4	Health app	FFT-i; ecg; libpng; aes; CRC32; FFT	pipeline	2.55%
Scenario 5	Voice synthesis	rsynth; aes; CRC32; FFT	pipeline	3.48%
Scenario 6	Multitasking	basicmath; bitcount; qsort; stringsearch; 3mm; atax; dynprog; correlation	parallel tasks	8.92%

health app that performs an ECG and uses the opensource libpng library to create an image from the data source -, and scenario 5, an app that uses rsynth to generate synthetic voice for user-device interaction. Finally, scenario 6 represents a multitasking environment in which the edge device is receiving tasks from multiple sources. For instance, 3mm and atax are matrix multiplication, transpose and vector multiplication kernels used in graphics processing, and dynamic programming (dynprog) and correlation are commonly used in data analytics.

Most of the chosen applications contain some degree of NEON usage. For instance, the selected kernels (correlation, 3mm, atax) are known for generating vectorized instructions, while the opensource libraries libav and libpng are optimized to use NEON operations. In our scenarios, from the 23 benchmarks used, only the applications bitcount, stringsearch, dynprog, h264-dec, CRC32 and aes do not present NEON instructions, representing common integer-only workloads. Although the selected applications - and their NEON usage - are representative for an assorted edge computing environment, a scalability study, in which we further increase the number of NEON operations executed, will be presented in the section 6.1.3.

Finally, to compile our workloads, we have used the gcc arm cross compiler arm-linux-gnueabi-gcc version 7.3.0 with -O3 optimization flag, which includes flags to generate vectorized instructions. The open-source libraries are also configured to use optimizations for NEON.

For the scheduler, there is a reallocation cost for each time a new workload is loaded from the queue. This migration cost considers the amount of time necessary to populate the L1 data cache, and was obtained through experimentation. The A15 processors need an average of 12K cycles to fill its data cache, while the A7 requires 17K cycles. This value may be improved, since we are not using any data prefetch technique when migrations are applied. Therefore, we are considering only the cache warm-up process as our migration cost, and we base this assumption on a previous work by Li et al. (2007), which claims cache overheads are dominant for task migration. However, it is important to notice that some extra overheads may apply, such as recovering the register file and core state and retraining the branch predictor.

Experiments: We have built several PHISA configurations using A7 and A15 processors with different ratios of full and partial cores. Table 6.2 shows all the tested configurations with their area and power characteristics. The configurations names are codified as Ax(yFzP) to express the cores used on it, being x the core type (A7 or A15), y

Table 6.2: Multicore configurations. *PHISA core using emulation

Configuration	A7		A15		Area (mm ²)	Power (W)
	Full	PHISA	Full	PHISA		
A15(4F0P)	0	0	4	0	14.12	2.76
A15(3F1P)	0	0	3	1	11.76	2.66
A15(2F2P)	0	0	2	2	9.41	2.56
A15(1F3P)	0	0	1	3	7.05	2.45
A15(1F0P)	0	0	1	0	3.53	0.69
A15(0F1P)A7(2F0P)	2	0	0	1	2.19	0.69
A15(0F1P)A7(1F1P)	1	1	0	1	2.07	0.69
A15(1F0P)A7(2F0P)	2	0	1	0	4.54	0.80
A15(0F1P)A7(2F0P)	2	0	0	1	2.19	0.69
A15(0F1P)A7(1F1P)	1	1	0	1	2.07	0.69
A15(0F1E)A7(2F0P)	2	0	0	1*	2.19	0.69
A15(0F1E)A7(1F1E)	1	1*	0	1*	2.07	0.69
A15(1F0P)A7(2F0P)	2	0	1	0	4.54	0.80
A15(0F1P)A7(4F0P)	4	0	0	1	3.20	0.80
A15(0F1E)A7(2F2E)	2	2*	0	1*	2.96	0.79

the number of (F)ull cores, and z the number of (P)artial cores. For instance, configuration A15(0F1P)A7(2F0P) is composed of 1 partial A15 core and 2 full A7 cores. We have also tested these configurations in three setups, aiming to understand the different behaviors of the PHISA system. We briefly describe all these setups shown in the section 6.1.2. These are also summarized in table 6.3.

In Setup 1 (6.1.2.1), we progressively replace full A15 cores by partial-ISA A15 cores to observe the impact of excluding the instruction extension datapaths. The first block of configurations in table 6.2 shows all the tested scenarios of this experiment, along with their extracted peak power and area, where the configuration names represent the type of cores they implement. For instance, the A15(3F1P) is a 4-Core processor with 3 Full cores and 1 PHISA.

We start by building PHISA configurations composed of A15 cores without NEON units (partial cores) and full A7 cores in Setup 2 (6.1.2.2). In this experiment, we compare the PHISA configurations that have the same peak power as a single-, full-ISA A15 core. For instance, as the table 6.2 shows in the second block of configurations, configuration A15(0F1P)A7(2FP) has approximately the same peak power as the A15(1F0P). We also extrapolate the partial cores usage - aiming to reduce energy further - replacing one of the A7 full cores with a partial A7. The goal of this setup is to show how a PHISA design can

Table 6.3: The experiments and their goals

	Description	Goal	Baseline	Configurations	Section
Setup 1	Homogeneous PHISA organization	Measure the impact of removing ISA extensions from a multicore processor.	A15(4F0P)	A15(4F0P); A15(3F1P); A15(2F2P); A15(1F3P)	6.1.2.1
Setup 2	Heterogeneous PHISA organization with same power budget of single core	Use the extra area and power of removing ISA extensions to create a heterogeneous system.	A15(1F0P)	A15(1F0P); A15(0F1P)A7(2F0P); A15(0F1P)A7(1F1P)	6.1.2.2
Setup 3	Heterogeneous PHISA organization vs DynamIQ-like configuration	Understand which gains are derived from the heterogeneous PHISA organization and which are from the use of big and little cores.	A15(1F0P)A7(2F0P)	A15(1F0P)A7(2F0P); A15(0F1P)A7(2F0P); A15(0F1P)A7(1F1P)	6.1.2.3
Setup 4	Heterogeneous PHISA organization with emulation vs DynamIQ-like configuration	Apply emulation to reduce migrations in the PHISA system and amortize the performance loses.	A15(1F0P)A7(2F0P)	A15(1F0P)A7(2F0P); A15(0F1E)A7(2F0P); A15(0F1E)A7(1F1E)	6.1.2.5
Setup 5	Heterogeneous PHISA organization with same power budget of DynamIQ-like	Reestablish the power budget to compare the DynamIQ-like system with the PHISA multicores.	A15(1F0P)A7(2F0P)	A15(1F0P)A7(2F0P); A15(0F1P)A7(4F0P); A15(0F1E)A7(2F2E)	6.1.2.6

improve energy and performance over a system with the same power budget.

We perform another experiment in Setup 3 (6.1.2.3) in which the power and area constraints are lifted to compare the PHISA system against a traditional single-ISA heterogeneous processor - reflecting an ARM DynamIQ configuration. The goal is to understand if the gains observed in Setup 1 were due to the PHISA configuration or because of the heterogeneous environment. The configurations in this setup are listed in the third block in table 6.2. This is represented as in configuration A7(2F0P)A15(1F0P) in table 6.2, which has the same cores as configuration A7(2F0P)A15(0F1P), but all full-ISA. We also discuss, in Setup 4 (6.1.2.5), how these configurations would perform using emulation in the partial-ISA cores before migrating to a full core, without considering any power or area constraints (fourth block in table 6.2). Note that the baseline processor in Setups 3 and 4 **do not** respect the power and area budgets and is much bigger than their PHISA counterparts.

Finally, we apply the power constraints back to show how a PHISA system of the same power as the DynamIQ configuration would perform (Setup 5, 6.1.2.6). We also allow the partial cores to emulate NEON instructions, aiming to reduce migrations in these cores. These are presented in the fifth block of configurations in table 6.2.

In the figures we present the results, **the performance is measured by the number of cycles, energy is the cycles x power, and EDP is the product of cycles and energy**. These are all presented normalized by the given baseline for each experiment (as shown in table 6.3). Therefore, all values below 1 are better than the baseline, while values above 1 are worse. Note that this also holds for what we call *performance* during this section, as performance here is the normalized number of cycles.

We also perform a final analysis in section 6.1.3, in which we estimate the behavior of PHISA in environments with high NEON usage. We have tested configurations similar to those from the previous experiments to analyze the behavior of Energy-Delay Product (EDP) when hypothetical applications with high usage of NEON instructions are executed.

6.1.2 Results

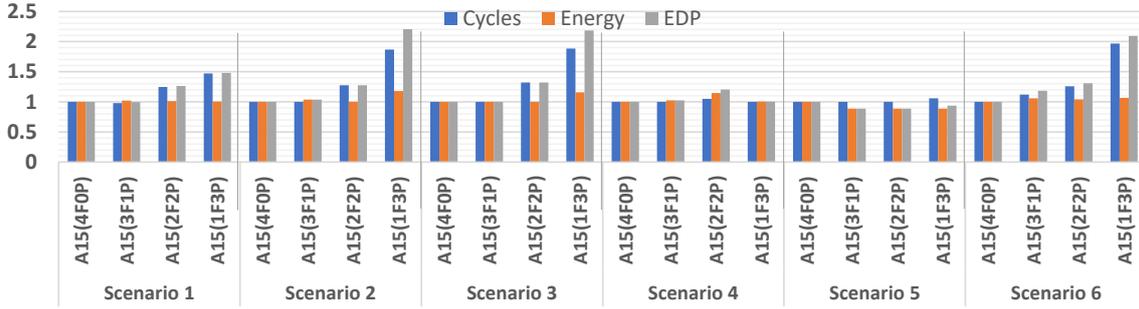
6.1.2.1 Impact of Partial ISA Cores

Experiment discussion: In this experiment, we measure the impact of implementing partial ISA cores in a multicore environment. The goal is to evaluate if removing cores capable of executing NEON operations would impact the performance and energy of the system, and in which ratio (full:partial) this impact would become relevant. This is done by progressively replacing full A15 ISA cores by partial A15 ISA ones in systems with 4 cores.

The first block of configurations in table 6.2 shows how area and power behave in the modeled systems of this experiment, with an expressive reduction in area and a smaller, but significant, decrease in power for configurations that comprise partial ISA cores. For example, when aggressively replacing full cores in the quad-core processor, the area is reduced by 50%, while power decreases by 11%.

Given the expected area and power decrease, we now analyze how they influence performance and energy consumption. We have executed all the six scenarios in table 6.1 in all systems from the first block in table 6.2 using the naive scheduler presented in section 3.2 to handle workload migration. Figure 6.2 shows results for this experiment. The x-axis contains the different A15 multicore versions, separated by each evaluated application scenario. The y-axis shows the normalized performance, energy, and EDP with respect to the A15(4F0P) configuration, which represents a traditional full-ISA multicore

Figure 6.2: A15 cores with progressive replacement of full with partial ISA cores. Performance, Energy and EDP are normalized to the A15(4F0P) configuration



processor. As a reminder, in the figures, **the performance is measured by the number of cycles, energy is the cycles x power, and EDP is the product of cycles and energy.** Therefore, for all the metrics, the lower the bar, the better. As the figure 6.2 shows, for most of the scenarios, the number of cycles increases as we include more partial cores, which is expected, as partial cores will need to migrate tasks that require NEON instructions. Energy, on the other hand, remains almost constant in most cases, due to the power reductions of the partial cores.

Observations from this experiment: Two observations should be highlighted at this point. (i) Although the cycle count increase is significant in the tested scenarios, this increase is much smaller when the proportion of full cores is high. For example, in the configuration with 75% of full ISA cores and 25% partial ISA cores (the A15(3F1P)), the increase in cycle counts is only relevant in scenario 6 (about 10%). In other words, *partial cores can be introduced in the system as long as we provide enough full cores for NEON execution.* (ii) As seen in Table 6.4, the full ISA A15 processor power is about 14x higher than the full A7 and occupies about 7x more area. A single partial A15 ISA core has 66% less area from a full A15 core and 15% less power. The freed area represents 4 times the area of a full A7, while the freed power is approximately the same as 2 full A7 cores. Thus, *extra A7 cores (which may be full or partial) can be introduced in the freed area of the system, while still respecting a power budget.* We explore next the trade-off between replacing full A15 by partial A15 cores - which consequently decreases performance - and adding A7 cores to recover some of the performance and increase energy efficiency.

Table 6.4: Area and power of full and partial A15 and A7 processors.

	A15		A7	
	Full	PHISA	Full	PHISA
Area(mm ²)	3.53	1.17	0.5	0.38
Power(W)	0.69	0.59	0.05	0.046

6.1.2.2 Full Core vs PHISA Multicore - Sharing a Power Budget

Let us consider that most IoT and mobile systems are battery-powered and that it is necessary to limit their designs to a particular peak power supplied by their batteries. Based on this reasoning, we establish a power budget for our system and use the extra area and power provided by removing NEON pipelines from A15 cores to add full A7 cores in the processor. Through this methodology, we create heterogeneous multicore configurations (in organization and ISA) that fit in the same area and power budgets of a traditional single-core processor. To exemplify this design, we have built the configurations A15(0F1P)A7(2F0P) and A15(0F1P)A7(1F1P), which have approximately the same peak power as the traditional single-core A15 processor (A15(1F0P), as shown in table 6.2).

Experiment discussion: Figure 6.3 shows the performance and energy consumption of the PHISA configurations A15(0F1P)A7(0F2P), A15(0F1P)A7(1F1P) and the traditional single-core A15(1F0P). PHISA multicores can significantly decrease energy consumption while also improving performance, as long as enough full cores are provided. A15(0F1P) A7(2F0P) reduces energy by 3.11x while improving performance by 1.94x in scenario 1 (Image Processing) when compared to the baseline. Similar results are observed in other scenarios. The performance increase is mainly attributed to the extra cores present in the system, which can execute more workloads in parallel. As the workloads are independent (apart from their pipelined behavior), the scheduler can easily distribute the applications between cores, so more cores result in more performance. On the other hand, energy is reduced by the use of much less power-hungry cores. Not only the partial ISA A15 cores have reduced power when compared to the traditional design, but the full A7 processors added to the system are also much more efficient. The exception is in scenario 6 (Multitasking), which is also the scenario that uses NEON operations the most. In this scenario, the pressure on the full cores (A7 cores) is much higher due to more usage of NEON instructions, thus performance improvements are smaller.

About the cores usage: Figure 6.4 shows the usage of the cores in configuration A15(0F1P) A7(2F0P) running scenario 6. In the figure, Core 0 is the partial A15 core, and the others are the full A7. The figure shows how the big cores are idle (low step) during a great part of the execution. The "solid bar" in Core0 are constant changes in the core state, which happen when the core has no more tasks to run (idle), receives a task (active), and the task fetches a NEON instruction and migrates again (idle). Such periods of inactivity are usually of 160K cycles, which is the period of migration in the full cores.

Figure 6.3: Evaluation PHISA multicores against a single-core baseline under a 700mW power budget. Performance, Energy and EDP are normalized to the A15(1F0P) configuration

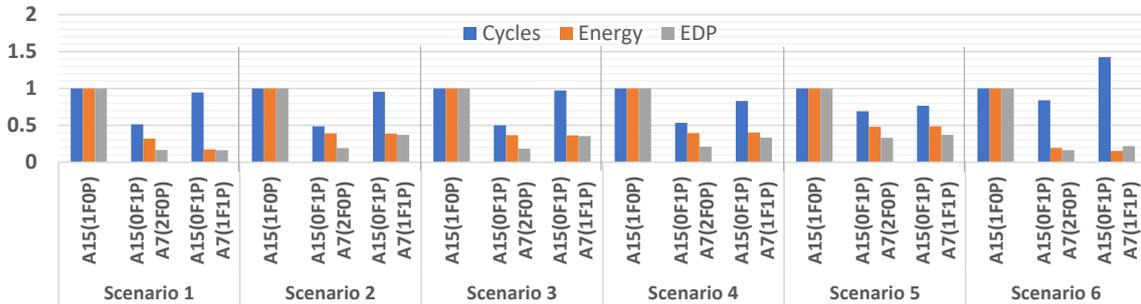
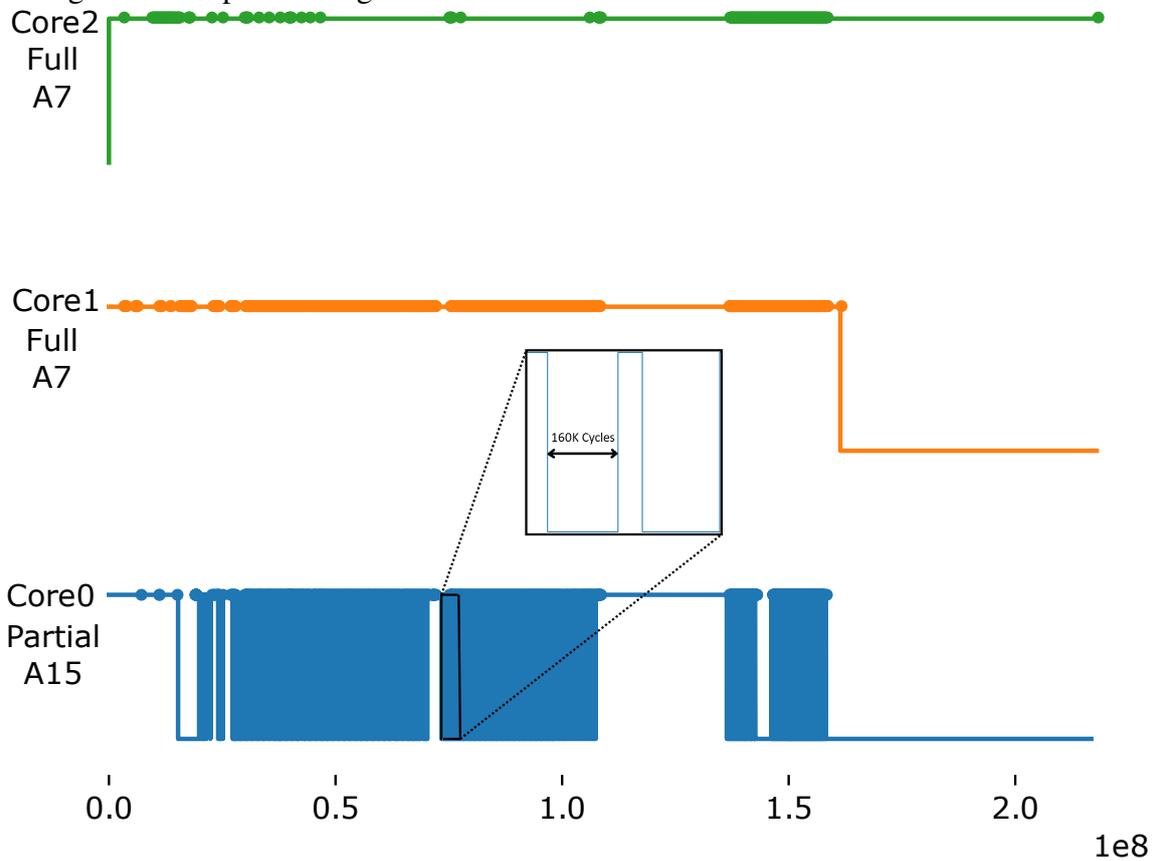


Figure 6.4: Core usage in configuration A15(0F1P)A7(2F0P) running scenario 6. High step means the core is in usage, low step is idle. Solid bars are constant idle-active changes. Dots represent migrations.



This smaller load in the big cores greatly affects the energy consumption, as the A7 cores have a peak power nearly 10x lower than the A15. Thus, although the maximum peak power of the PHISA is the same as the baseline, the dynamic peak power of the system is smaller because the workload is not fully concentrated in the big cores.

Introducing partial-ISA A7: Extrapolating further the reduction of power using partial cores, configuration A15(0F1P)A7(1F1P) - also in Figure 6.3 - replaces a full A7 core with its partial version, leaving the configuration with only one full core. The energy

consumption shows a small decrease in the scenarios, but the extra pressure in the only full core in the system causes a high increase in the number of cycles, which in turn, prevents higher energy reductions. In general, the trade-off between performance and energy consumption - the EDP - is worse in configuration A15(0F1P)A7(1F1P), mainly because of the performance of such a system. Scenarios 1 and 6 show a huge reduction in energy, and, controversially, a high increase in the number of cycles. This is because both of these scenarios include many applications that execute NEON instructions and compete for the only full core, an A7 core. As most of the execution happens in this A7, which is extremely energy efficient, the energy consumption falls, but the cycle count increase. Other scenarios have higher NEON usage than scenario 1, however, in these, the NEON operations are concentrated in fewer applications. For instance, in scenario 5 the NEON operations are only required by rsynth and FFT, which makes it simpler for the scheduler to manage the A7 resources. In general, the power reduction from replacing a full A7 core with a partial version is too small and does not show enough advantages, as is the case of the bigger A15 cores.

In this experiment, we have seen that it is possible to create heterogeneous systems with PHISA and have better energy consumption than power equivalent full processors. Nonetheless, if the power budget is not considered, this is as expected from all heterogeneous processors. In the next section, we isolate the gains provided by the PHISA system by comparing it with an equivalent full core processor.

6.1.2.3 PHISA vs Traditional Heterogeneous Systems (DynamIQ)

Experiment discussion: Heterogeneous processors naturally deliver better energy efficiency than homogeneous multicores. To understand which gains are derived from the usage of PHISA and which are simply from having additional cores, we now evaluate configuration A15(1F0P)A7(2F0P), which represents a DynamIQ heterogeneous processor in Figure 6.5. Nonetheless, it is important to highlight that, in this configuration, **the power and area budgets are completely ignored**. Configuration A15(1F0P)A7(2F0P) is much bigger (more than twice the size) and has higher power (about 14%) than the PHISA equivalents of the same core count. Thus, it is expected that the PHISA system will be worse in performance in this scenario, as the DynamIQ has many more resources to use. As can be seen in the Figure 6.5, the cycle count of the configurations with partial cores is higher than those from the DynamIQ configuration, which was expected, as the full core configuration does not require migrations to execute NEON instructions.

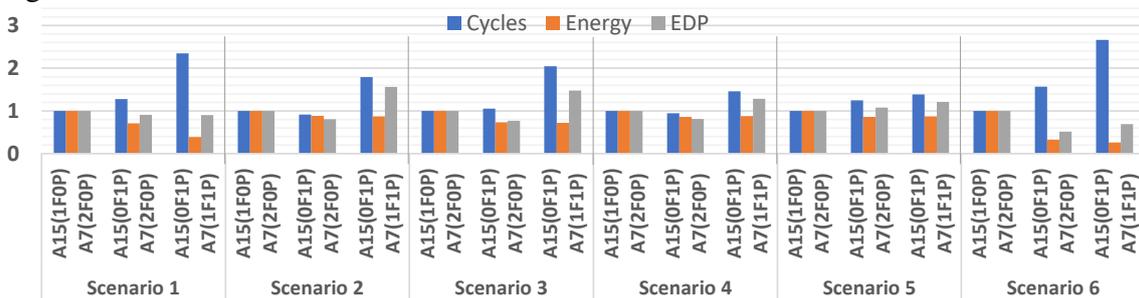
On the other hand, energy consumption in the PHISA systems is usually lower, showing that partial ISA cores are essential to decrease energy. In fact, the EDP of the PHISA system is usually lower, showing that the partial cores can deliver better trade-off between performance and energy consumption. Energy consumption in the PHISA configurations is reduced due to the partial A15 cores. The original A15 processor has a peak power 15% higher than its counterpart partial ISA version. Besides, since in the PHISA configuration the A15 cannot execute NEON instructions, the scheduler must migrate the workloads from the power-hungry A15s to the efficient A7s more frequently than in the traditional system. Effectively, these migrations increase the usage of the A7 cores, leaving the A15 idler and reducing energy consumption.

The DynamIQ configuration, on the other hand, tends to use the A15 core more often to increase performance, which comes at the price of energy. If the scheduler of the DynamIQ were to be changed to optimize energy - and use the A7 cores at the same ratio as PHISA - it would still consume more energy, as the full A15 core dissipates more power than the partial-ISA version. This balance is clearly seen with configuration A15(0F1P)A7(1F1P) in scenario 6, which frequently requires the NEON unit. There is only one full A7 in the system, and it has to execute all the NEON requests from the workloads. This pressure increases the time required to execute all applications, but also reduces energy consumption, as the A7 is much more efficient than the A15.

6.1.2.4 Task Migration or NEON Emulation

About task migration: Scheduling in PHISA multicores is tied to the usage of NEON instructions by a workload. To avoid constant migration, we use a scheduler policy of prioritizing NEON applications to full cores: if a full core migrates an application and

Figure 6.5: Evaluation of PHISA multicores against a DynamIQ baseline. The baseline has the same amount of cores as the PHISA configurations, thus there is no power/area budget. Performance, Energy and EDP are normalized to the A15(1F0P)A7(2F0P) configuration.



there are two possible targets (same size of workload queue), being one full and the other a partial core, it will schedule the application to the full core. This helps to avoid constant back and forth migrations from the partial cores, as they will be assigned more integer workloads. However, this does not completely remove this problem, as when the full cores are all busy, and the partial ones are free, workloads will be assigned to the partial cores, independently of the type of instruction they hold. These excessive migrations are also the reason for the high increases in the cycle counts of PHISA systems observed in the experiment of figure 6.5. To mitigate this problem and reduce the number of migrations, we use the minimalist policy scheduler with emulation described in section 3.2.2. In this policy, every time a NEON instruction is fetched by a partial core, the scheduler will decide whether the instruction should be emulated in software or migrate to a capable core: if the workload has already been executed for more than a certain threshold time in that core, it migrates. Otherwise, it emulates the instruction. For the sake of compatibility of the migration times, we have set this threshold to 160K cycles, the same time as the original migration event for full cores.

6.1.2.5 The impact of emulation

Figure 6.6 shows the results for configurations A15(0F1E)A7(2F0P) and A15(0F1E)A7(1F1E) compared to the DynamIQ-like configuration. In this processor, the 'E' in the name means a partial-ISA core that can emulate NEON instructions in software. As the figure shows, the emulation strategy can amortize some of the impacts in cycle counts caused by the partial cores. In some scenarios, such as 2, 4, and 5, the cycle count is even smaller than the baseline, due to the balancing of workloads in the cores. The energy, on the other hand, increases when compared to the non-emulation scenario, as now the partial A15 cores are used more frequently. When one considers the EDP, the PHISA systems are better than the baseline in almost all cases. From our experiments, emulation can potentially reduce the migration overhead from 10% to 0.5%. Nonetheless, it is important to remember that, in this case, the original DynamIQ-like configuration is 2x bigger and has 14% higher peak power than the PHISA configurations (table 6.2).

6.1.2.6 PHISA with emulation vs DynamIQ - Power Parity

When one considers a power budget parity between the DynamIQ and the PHISA configurations, it is possible to add two extra A7 cores in the PHISA system. This parity is

represented in configuration A15(0F1P)A7(4F0P) and a version with more partial cores, but that allows emulation, A15(0F1E)A7(2F2E) in table 6.2.

Figure 6.7 shows the results for running the scenarios in these configurations. In all the scenarios, the PHISA processors have better performance, energy, and, consequently, EDP than the completely full-ISA processor. The best performance improvement is observed in scenario 4 (Health app) with 32% reduction in cycle count. This scenario presents applications that contain high NEON usage (ECG), but that are very fast to execute, creating a perfect combination for the extra A7 cores.

The best energy consumption is observed in scenario 5 (Voice synthesis) - with 82% reduction -, which is composed of only two NEON applications that can execute in the two full cores of the system, while the other applications are executed in the (more energy efficient) partial-ISA cores. Emulation can reduce energy and execution time even further in some cases, especially when the applications show very sparse use of NEON, as it reduces the number of migrations in the entire system. However, in scenarios of high, and concentrated, NEON usage (such as the Multitasking), the emulation cost can be higher than the migration, increasing energy and execution time.

Furthermore, as shown in table 6.2, the PHISA configurations are still smaller, in area, than the DynamIQ-like processor. Thus, the PHISA designs present an opportunity to create systems that are smaller and more energy-efficient than the current industry trend.

Figure 6.6: Evaluation of PHISA multicores allowing emulation against a DynamIQ baseline. The baseline has the same amount of cores as the PHISA configurations, thus there is no power/area budget. Performance, Energy and EDP are normalized to the A15(1F0P)A7(2F0P) configuration

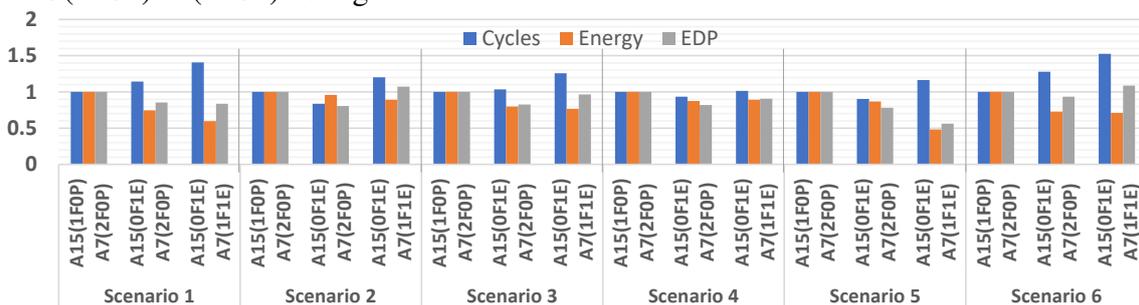
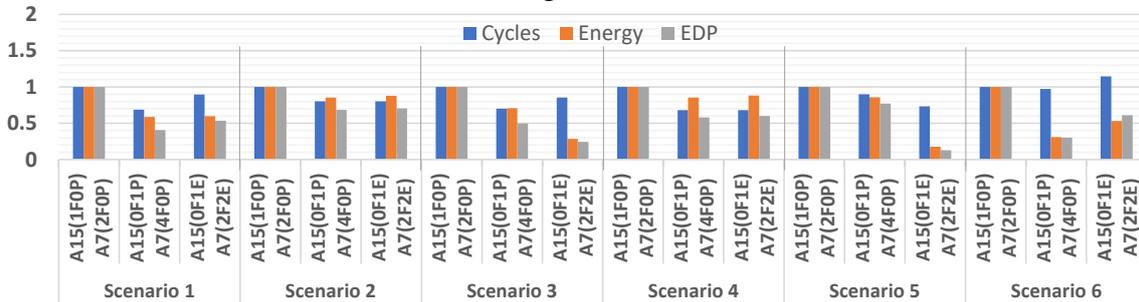


Figure 6.7: Evaluation of PHISA multicores with and without emulation against a DynamIQ baseline under a 800mW power budget. Performance, Energy and EDP are normalized to the A15(1F0P)A7(2F0P) configuration.



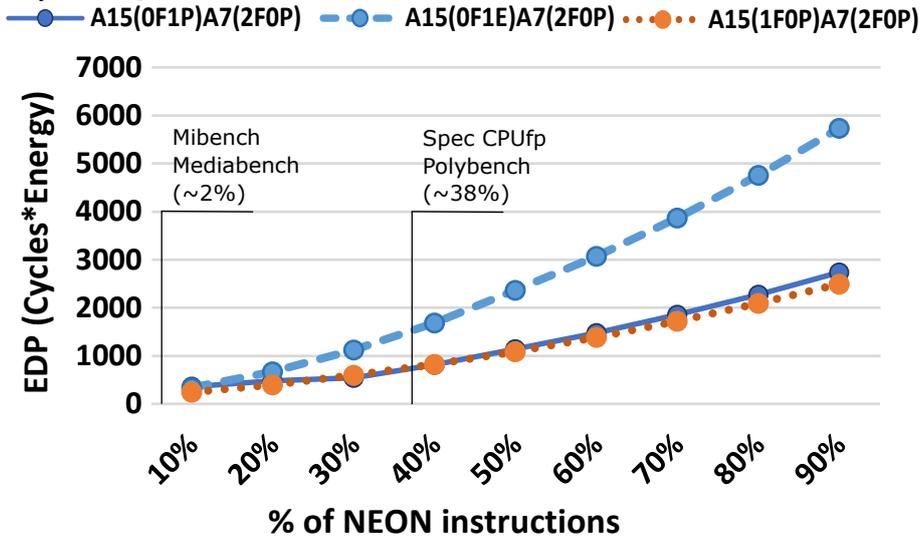
6.1.3 Analysis of PHISA on High NEON Usage

Our experiments have been using scenarios with some of the single-threaded workloads presented in Figure 1.1. Although the selected set of workloads covers a wide range of applications from the embedded system and IoT market, one may question the behavior of the system when exposed to higher amounts of NEON instructions. Considering that the number of instructions from removed extensions will directly influence the behavior of a PHISA multicore, we now use an analytic model of hypothetical applications, in which we can vary the number of issued NEON instructions, as shown in Figure 6.8. The goal is to observe how the different PHISA configurations scale with the number of NEON instructions compared to a traditional full-ISA system.

In this new environment, we assume configurations from the previous experiments, in which the **partial cores only execute integer operations, and the full cores execute both integer and NEON operations**. In the configuration A15(0F1E)A7(2F0P) the partial-ISA A15 core can also emulate NEON instructions. As a best-case comparison, we have selected configuration A15(1F0P)A7(2F0P), which represents a similar processor, but with all full-ISA cores.

We assume there is a migration cost (12k for the A15 and 17K for the A7, the same as in previous configurations) and that the number of migrations *increases proportionally to the ratio between NEON and integer instructions*: the higher is the ratio, the higher are chances of these instructions being interleaved, causing multiple migrations. This cost rises until 50% of NEON instructions, and from 60% forward, the cost decreases as the ratio inverts, and there is a lower chance of interleaved operations. For example, when the application has 10% NEON instructions, there will be nine integer instructions for one NEON instruction, which would cause one migration. For 50% NEON instructions, there

Figure 6.8: Behaviour on high NEON usage PHISA multicore (with and without emulation) and DynamIQ.



will be five integer instructions for each five NEON, which can be interleaved (1 int, 1 NEON, 1 int, 1 NEON...), and would cause five migrations. However, *this scenario will reverse* if there are more NEON instructions than integers. We also assume that a NEON instruction takes twice as many cycles to execute than an integer instruction in the A15 core and eight times more in the A7 core. These are average numbers estimated from simulations.

Figure 6.8 shows the behavior of the EDP of the configurations as the number of NEON instructions in the application increase. As the figure shows, configuration A15(0F1P)A7(2F0P) has good scalability, which is tied to its types of cores. As the number of NEON instruction increases, the partial A15 will become idle more often. Although this becomes a burden for the processor performance, migrating the load to the A7 processors greatly reduces the system dynamic peak power, which decreases the energy consumption. This allows the PHISA configuration to stay very close in EDP to its full-ISA counterpart. It is important to notice that the configuration with full cores has more than twice the area and higher peak power than the PHISA version.

On the other hand, the same PHISA configuration but with emulation capacity A15(0F1E)A7(2F0P) shows bad scalability in higher rates of NEON. When executing few NEON instructions, the emulation has good performance, however as the NEON instructions increase, the A15 processors will be assigned to emulate more of these instructions, which will incur in high-performance overhead. Furthermore, the A15 is a power-hungry core, which will also increase consumption.

This experiment demonstrates that the PHISA multicore has potential even when

the ratio of NEON instructions increases, as its expected EDP stays close to that of a full-ISA system. In fact, the difference in EDP from the full-ISA processor and the PHISA system in a modeled application with 90% of NEON instructions is of less than 10%. For both applications with low NEON usage (such as in the Mediabench and Mibench suites) and for high NEON usage applications (such as SPEC CPUfp and Polybench), the PHISA system can have similar scalability as the traditional DynamIQ-like heterogeneous processor. For low NEON usage, such as in Mibench applications, emulation is also a good choice to balance workloads between cores.

6.1.4 Scheduling Policies Impact

In this subsection, we discuss the impact of using different optimization policies in the PHISA scheduler. Details of the employed algorithms are presented in section 3.2.3.

6.1.4.1 Scheduling for Performance

As discussed in section 3.2.3, we have redesigned the naive scheduler to make a best effort to optimize the system for different goals. In the performance policy, the scheduler always gives priority to allocate tasks in the big OoO cores (A15), assuming that this core will execute the application faster than the small in-order cores (A7). Although this prioritization can improve system performance, it might not be the best strategy if one expects higher energy efficiency.

In this experiment, we have simulated the same scenarios from table 6.1, comparing the the configurations from setup 2 (heterogeneous PHISA against single-core baseline with same TDP) and setup 4 without emulation (heterogeneous PHISA against traditional DynamIQ-like system with same area TDP). Our goal is to evaluate how the performance policy affects the same scenarios already evaluated using the naive approach.

Figure 6.9 shows the performance, energy, and EDP of the PHISA configurations using the performance policy normalized by the single-core A15 processor. The figure shows that the PHISA configurations have better performance and energy consumption in all the scenarios. When compared to the naive scheduler, the performance is improved in every scenario for almost every configuration. The PHISA configurations with only one full core show even further improvements when compared to the naive scheduler. This is not only because of the policy itself but also because of the improvements in

Figure 6.9: Evaluation PHISA multicores against a single-core baseline under a 700mW power budget. Scheduling of tasks follows a performance optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the A15(1F0P) configuration.

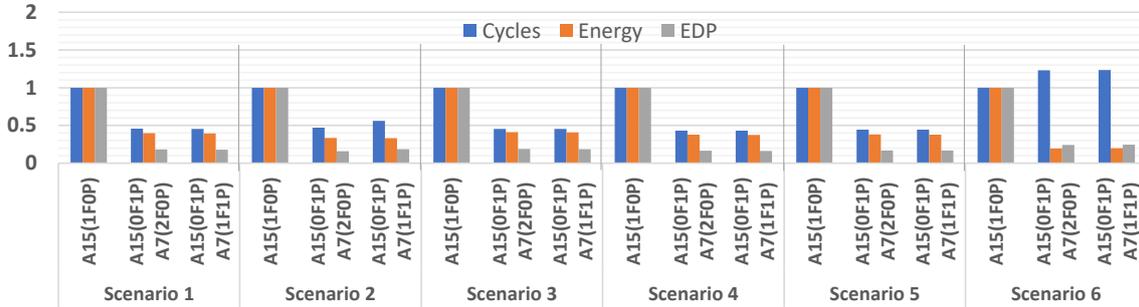
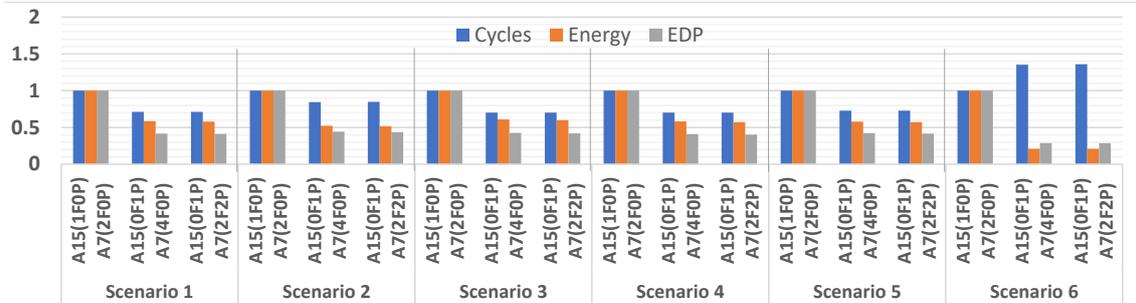


Figure 6.10: Evaluation of PHISA multicores against a DynamIQ baseline under a 800mW power budget. Scheduling of tasks follows a performance optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the A15(1F0P)A7(2F0P) configuration.

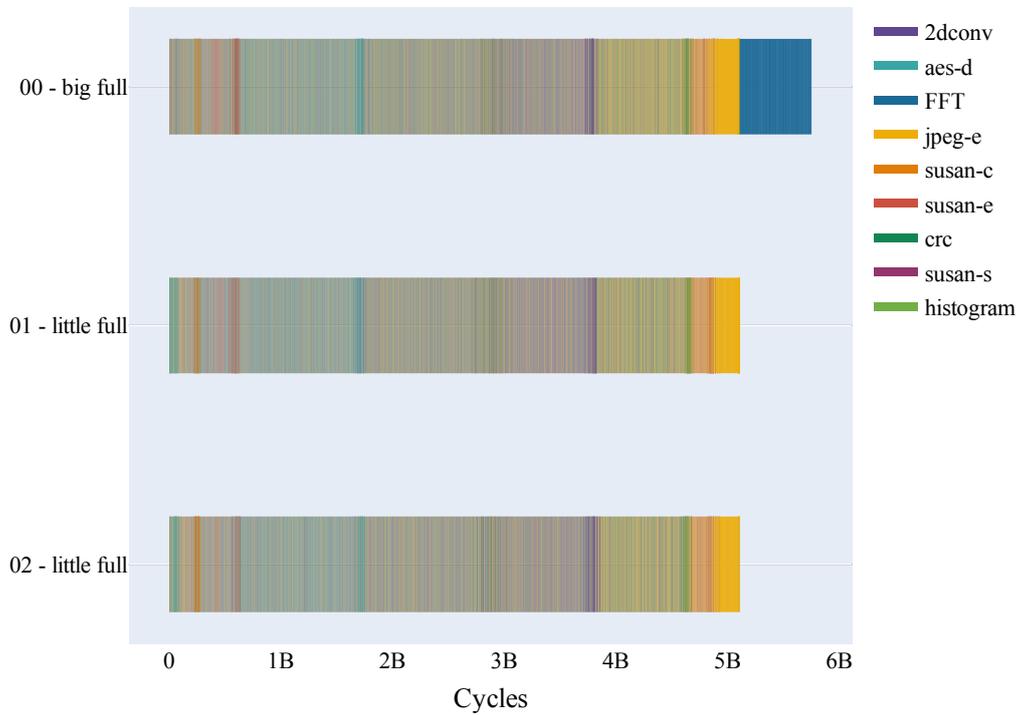


task preemption and workload queues that this new scheduler introduces - as discussed in section 3.2.3. Again, scenario 6 is the only one in which the PHISA systems show worse performance than the baseline, and this is because the scenario was built with high NEON usage in mind.

Figure 6.10 shows the performance, energy and EDP of the PHISA configurations using the performance policy normalized by a traditional DynamIQ-like configuration. Again, the PHISA configurations show both better performance and energy consumption in all scenarios. What is mostly interesting in this evaluation is that the differences between the results in the PHISA systems (with one or two full cores) are very small (close to 2% only).

Finally, figure 6.11 shows the scheduler behaviour of the traditional DynamIQ configuration A15(1F0P)A7(2F0P) while executing the applications in scenario 1. Figure 6.12 shows the behaviour in the same scenario for the PHISA A15(0F2P)A7(4F0P) configuration. As shown in the figure 6.12, the PHISA configuration has more cores to execute the multiple applications, increasing the throughput of the scenario. Migrations in the traditional DynamIQ happens only during preemption phases, while in the PHISA

Figure 6.11: Scheduler migrations using the performance policy in the traditional DynamIQ configuration A15(1F0P)A7(2F0P) during execution of scenario 1. Bars represent each application being run over time in each processor core.



the applications have to change cores whenever a non-implemented function is fetched in a partial core. This is better observed in the final execution of the *FFT* application, as in figure 6.11 it is completely run in the big full core, while in figure 6.12 it has to migrate from the big partial to the little full several times.

6.1.4.2 Scheduling for Energy

In this experiment, we prepare the same configurations and scenarios from subsection 6.1.4.1, but change the scheduler policy to optimize energy consumption. As discussed in the previous section 3.2.3, this policy prioritizes the allocation of tasks in the little cores, assuming that these will be more energy efficient. It is important to notice that in these scenarios, all the configurations use the energy consumption optimization policy, including the baseline.

Figure 6.13 shows the results for this experiments. The PHISA configurations using this policy are able to reduce the energy consumption further when compared to the results in subsection 6.1.4.1. Performance is also improved in relation to the baseline, as the baseline is also running the energy policy.

Figure 6.14 shows the same experiment for the PHISA configuration with same TDP as a traditional DynamIQ. Again, PHISA is able to further reduce the energy consumption, performance, and - consequently - their trade-off in the form of EDP.

Finally, figures 6.15 and 6.16 show the scheduling behaviour of configurations A15(1F0P)A7(2F0P) (DynamIQ) and A15(0F1P)A7(4F0P) (PHISA) using the energy policy respectively. Once more, the PHISA configuration can deliver more throughput

Figure 6.12: Scheduler migrations using the performance policy in the PHISA configuration A15(0F2P)A7(4F0P) during execution of scenario 1. Bars represent each application being run over time in each processor core.

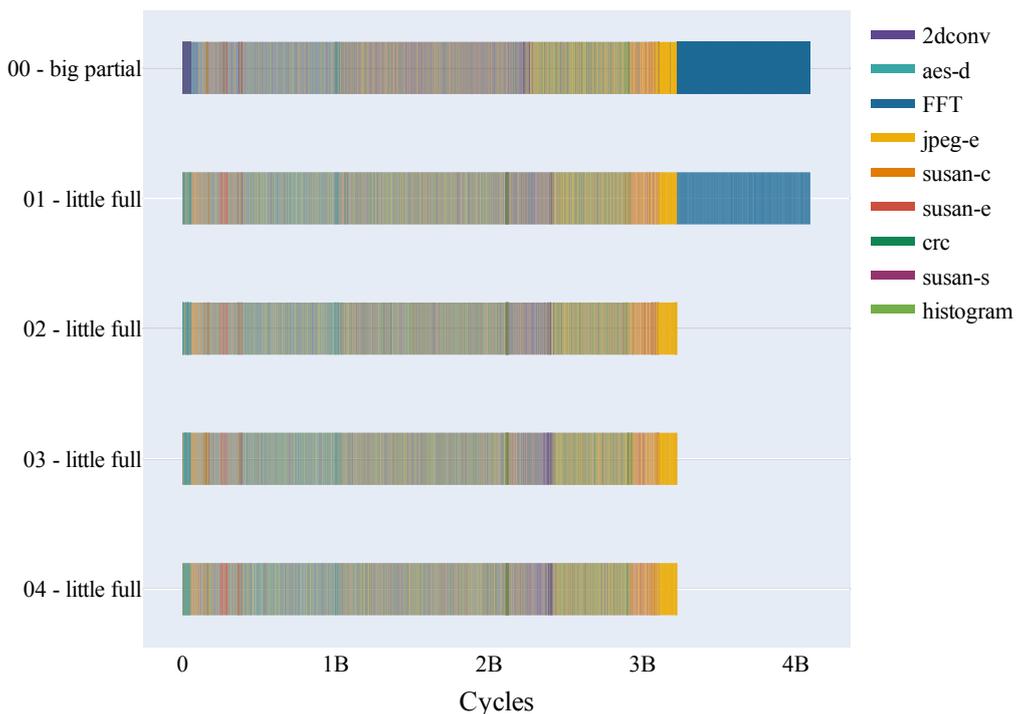
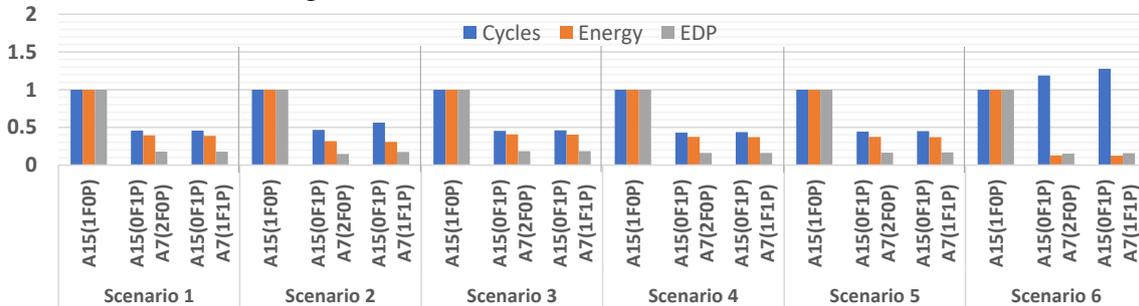


Figure 6.13: Evaluation PHISA multicores against a single-core baseline under a 700mW power budget. Scheduling of tasks follows a energy consumption optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the A15(1F0P) configuration.



than the DynamIQ. However, as the full little cores are prioritized due to their energy efficiency, the number of migrations is reduced. This is seen in the final execution of the *FFT* application, in which in both figures, it finishes executing in one of the full little cores (in contrast with the performance policy).

6.1.5 Summarizing the results

In this section we have analyzed the PHISA multicores for single-threaded applications under various scenarios, configurations and scheduling policies. We have shown that, in most scenarios, the strategy of removing resources from cores once destined to ISA extensions to invest in more GPPs provides better performance and energy consumption. The overhead of migrating workloads from partial- to full-ISA cores is usually compensated by the improvements provided by the extra workload throughput. Moreover, a system that provides efficient scheduling policies can further improve the overall performance and energy consumption of the processor.

Figure 6.14: Evaluation of PHISA multicores against a DynamIQ baseline under a 800mW power budget. Scheduling of tasks follows a energy consumption optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the A15(1F0P)A7(2F0P) configuration.

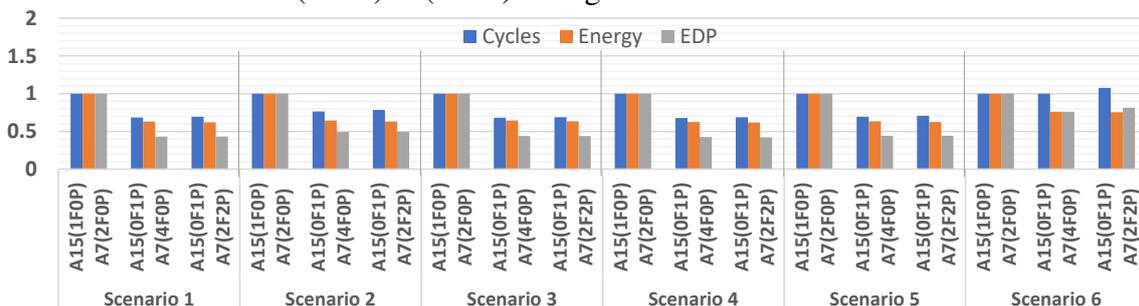


Figure 6.15: Scheduler migrations using the energy policy in the traditional DynamIQ configuration A15(1F0P)A7(2F0P) during execution of scenario 1. Bars represent each application being run over time in each processor core.

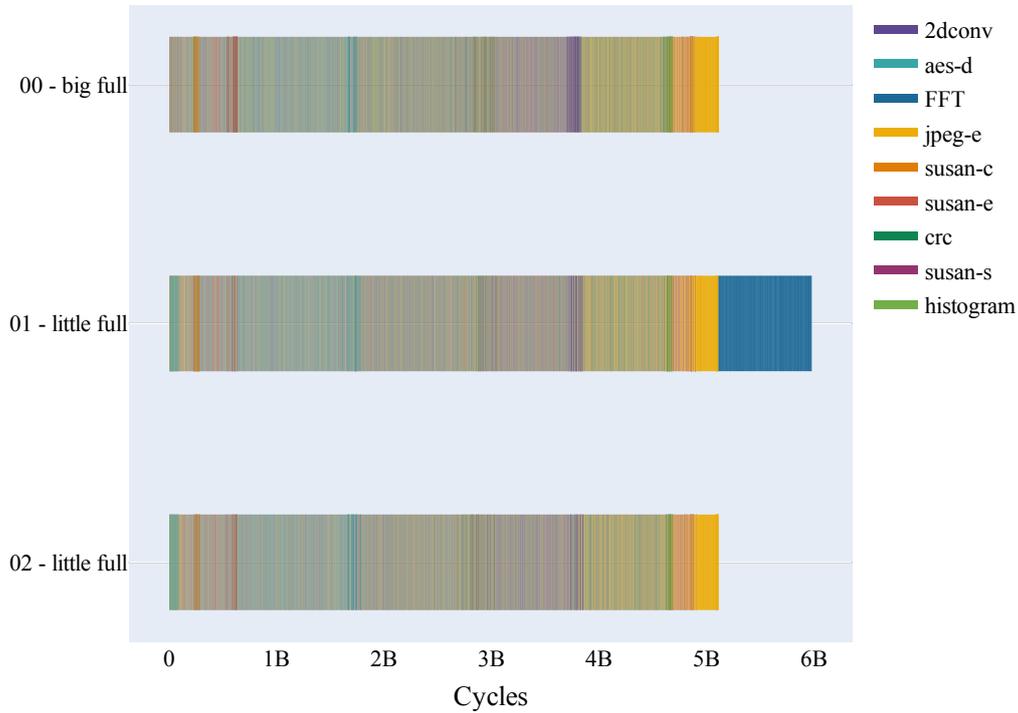
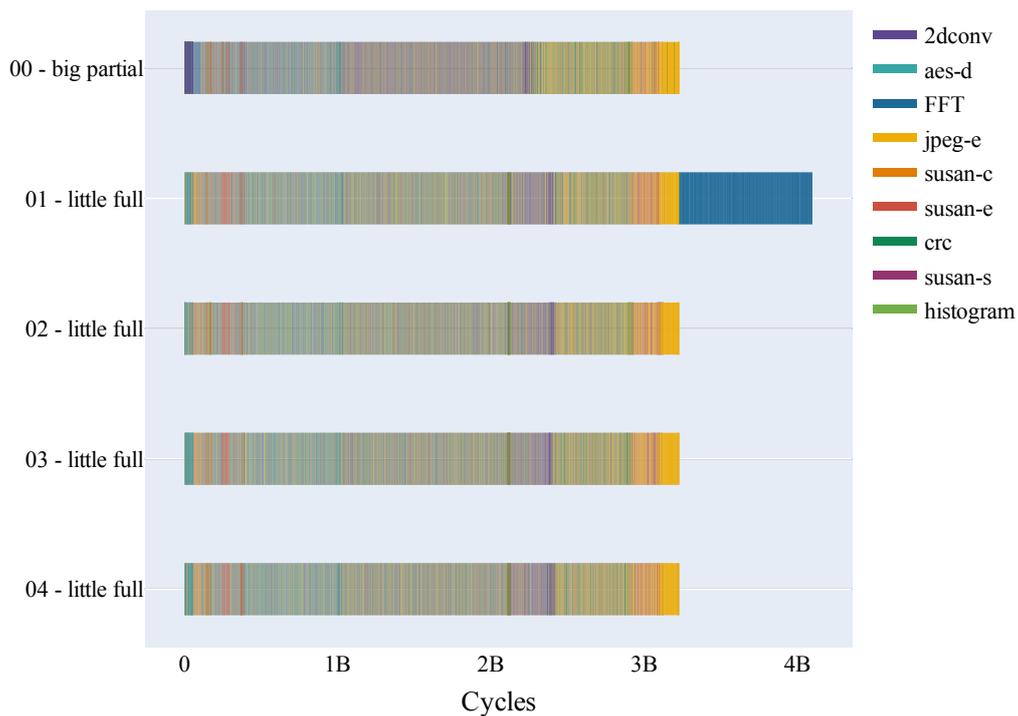


Figure 6.16: Scheduler migrations using the energy policy in the PHISA configuration A15(0F2P)A7(4F0P) during execution of scenario 1. Bars represent each application being run over time in each processor core.



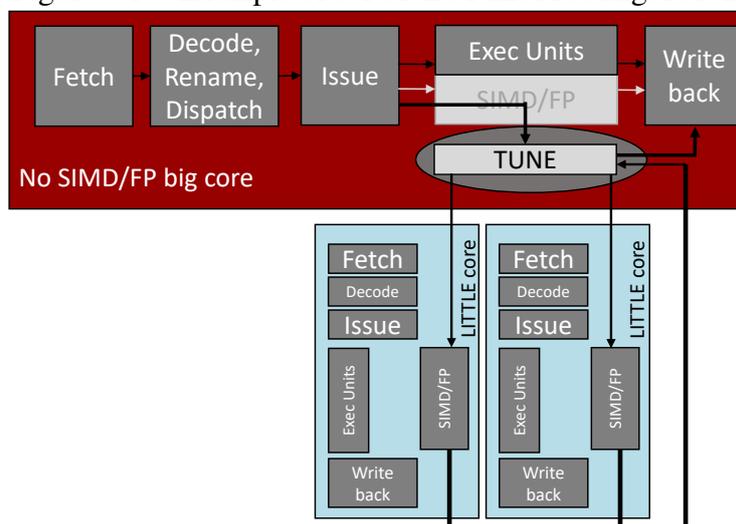
6.2 Evaluation of TUNEd PHISA with multi-threaded workloads

In this section, we evaluate the TUNEd PHISA, a system that combines the partial-ISA cores strategy with an instruction offloader to improve the performance of multi-threaded applications (see figure 6.17). We first present the methodology used to model and simulate the system. Then, we discuss the benchmarks used and how we build scenarios using them. Later, we present the results for all experiments, along with discussions for each scenario.

6.2.1 Evaluation Methodology

Modeling and Simulation Parameters: To extract area and power data, we also use McPAT in this evaluation. As mentioned in chapter 5, McPAT allows for configurations without FP and SIMD units (by simply setting the FP related tags in the template to zero), which also triggers the exclusion of the FP instruction window, the FP Register File (RF) and the FP register renaming structures. However, in the TUNE system, these structures **are still needed for decoding and offloading instructions**, thus we have modified McPAT to include such components in the model. To extract the performance of our system, we have used the gem5 simulator (BINKERT et al., 2011) in **Full System (FS) mode**. The FS mode emulates an entire platform, including the Operating System (OS). Thus, the simulations include all the typical overheads of parallel programming a real system would have. The offloader was also modeled to introduce the extra latency in

Figure 6.17: Example of a TUNEd PHISA configuration.



the execution of SIMD and FP instructions. In our experiments, we have considered *an extra ten cycles of latency for each SIMD and FP instruction*, which is approximately the same latency of the L2 cache. We model based in this latency as the L2 cache is also a shared resource between cores that, in the ARM processors, transfer the same amount of data per access (64 bytes) as the offloader (4x128bits).

Configurations: To evaluate a TUNEd PHISA system, we take as baseline a single-core, full-ISA, A15 processor. Then, we use an Asymmetric Multicore (AMC) traditionally employed to accelerate parallel applications, i.e., a processor with many in-order cores and one single OoO core. This is a configuration with one full-ISA A15 core along with four full-ISA A7 cores. To create our TUNEd PHISA configuration optimized for parallel applications, we remove the NEON units from the OoO core responsible for the serial regions (A15) and add four extra in-order A7 cores in their same area. Thus, the TUNEd PHISA configuration is composed of one partial-ISA A15 core along with eight full-ISA A7 cores.

Workloads: We have evaluated the TUNE mechanism using several benchmarks of different characteristics (table 6.5). The set includes applications of both high and medium ratios of parallelism and SIMD/FP usage. Thus, we can analyze the system in different scenarios. To compile our workloads, we have used the GCC arm cross compiler arm-linux-gnueabi-hf-gcc version 7.3.0 with -O3 optimization flag, which includes flags to generate vectorized and floating-point NEON instructions.

6.2.2 Performance results

We start our analysis with table 6.6, which shows two sets of results that help to understand the behavior of TUNE. The column $\frac{SingleA15}{SingleA7}$ shows how faster the A15 is compared to the A7 for each application. These results were obtained by simulating the single-threaded version of each application in both core types. Results show that the A7 slowdown varies from low 1.36x (*gemm*) to huge 5.70x (*gramschmidt*). Moreover, the column $\frac{8CoreA7}{SingleA7}$ shows the speedup achieved by an 8-core A7 processor when compared to a single A7, which represents how much performance the application can extract from parallel execution (8x means perfect parallel exploitation). Results show, as expected, that applications with smaller *PR* (from Table 6.5), such as *correlation* and *covariance*, can achieve lower parallel speedups. Nonetheless, some applications that previously showed high coverage of the *PR*, such as *bicg*, which region of interest is virtually 100% covered

Table 6.5: Application **region of interest** characterization in terms of parallel region size and SIMD/FP ratio in the serial region.

Benchmark	Parallel Ratio	Serial SIMD/FP Ratio
bicg	100.00%	0.00%
fdtd-apml	99.99%	0.00%
convolution-2d	99.99%	0.00%
gemm	99.97%	0.06%
symm	99.95%	0.00%
syrk	99.90%	0.04%
syr2k	99.89%	0.04%
atax	99.73%	0.06%
2mm	99.27%	1.03%
mvt	98.82%	1.11%
gesummv	98.76%	0.46%
3mm	98.68%	1.66%
doitgen	98.41%	0.83%
trmm	82.25%	6.74%
correlation	81.95%	10.10%
gramschmidt	78.70%	11.72%
covariance	77.40%	17.33%
lu	68.71%	0.09%

by the PR (Table 6.5), do not present the same expected speedup (only 4.86x). This is because many dynamic factors can influence the execution of the parallel region, such as shared memory accesses, data-synchronization, and bandwidth saturation (LORENZON et al., 2019). Table 6.6 also shows the overhead data for the simulation of a TUNE system in column *Offloading Overhead*. This data is obtained by measuring the cycles taken to execute each application serial region with and without using the offloading strategy. The data shows that the overhead is usually low, and is only significant in applications that have smaller parallel regions and larger amounts of SIMD/FP in their serial parts (Table 6.5).

Figure 6.18 shows the results of the different applications using the traditional AMC (A15 + 4 A7) and TUNEd PHISA (A15 (No NEON) + 8 A7) configurations. The figure shows both the speedups (left Y-Axis, represented by bars) of each configuration

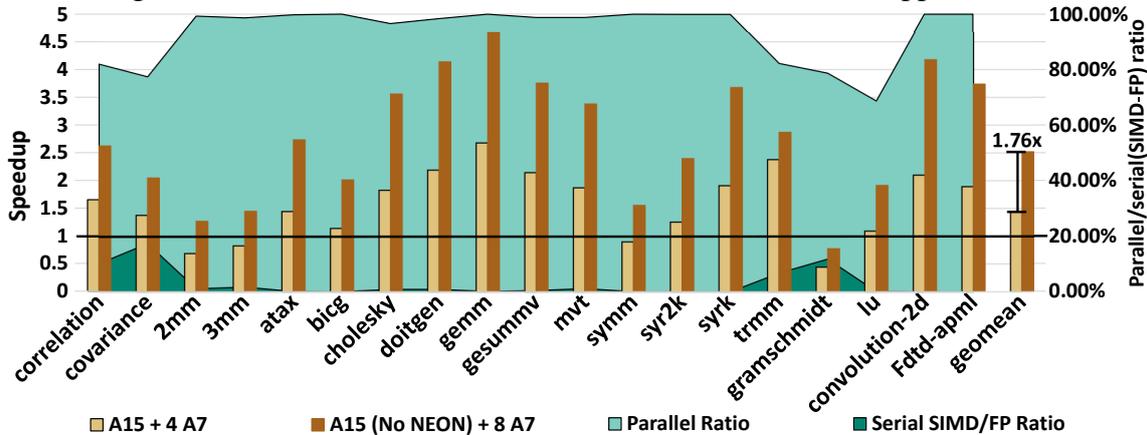
Table 6.6: Simulated application characteristics for Polybench applications. Columns show, in order: How many times a single A15 is faster than a single A7 (A7 slowdown); The parallel speedup of 8 A7 cores over a single A7; The overhead caused by offloading instructions in the TUNEd PHISA system.

Benchmark	$\frac{SingleA15}{SingleA7}$	$\frac{8CoreA7}{SingleA7}$	Offloading Overhead
correlation	1.51x	3.91x	14%
covariance	1.70x	3.38x	26%
2mm	5.04x	6.35x	2%
3mm	4.45x	6.38x	3%
atax	2.04x	5.58x	0%
bicg	2.41x	4.86x	0%
cholesky	1.51x	5.39x	1%
doitgen	1.79x	7.37x	1%
gemm	1.36x	6.36x	0%
gesummv	1.73x	6.50x	1%
mvt	1.88x	6.31x	1%
symm	4.07x	6.31x	0%
syr2k	3.11x	7.44x	0%
syrk	2.02x	7.44x	0%
trmm	1.99x	5.61x	12%
gramschmidt	5.70x	3.68x	15%
lu	1.84x	3.02x	0%
convolution-2d	1.87x	7.80x	0%
Fdtd-apml	2.04x	7.62x	0%

normalized by a single A15 core as a baseline and the PR and SRF parameters of each application (right Y-Axis, represented by the background areas). It is important to notice that we use the single-core A15 as a baseline to show the speedups of using both heterogeneous multicore systems. However, the parity of area exists only between the traditional configuration and the TUNEd PHISA system.

Some applications have a considerable amount of NEON instructions in their longer serial regions (e.g. *correlation*, *covariance* and *gramschmidt* have 10-17%, in table 6.5). These applications also have a smaller parallel speedup (as seen in table 6.6), which results in the worst-case scenarios for the TUNE configuration, as more instructions will require offloading and the extra A7 cores will not be optimally used. The lack of parallelism in these applications will affect both the speedup of the traditional system and TUNE. Still, with the extra cores, TUNE is able to extract more performance from the

Figure 6.18: Speedup according to the developed system simulation. Bars are the speedup over a single A15 core and areas are the % of *PR* and *SRF* of each application.



application. This behavior is more apparent when we analyze the application *lu*, which has the same long serial regions, *but almost no NEON operations*, as seen in table 6.5. In this case, with no offloading overheads to hold TUNE back, our system is nearly 2x faster than the traditional heterogeneous processor. The results show that in all these four applications, the TUNE system is faster than the traditional one, indicating that the extra cores can compensate for the slowdown of offloading instructions, even when the application is not highly parallel.

The application *gramschmidt* has similar parallel characteristics, however, both configurations show a slowdown in performance when compared to the single A15, due to the huge performance loss of the A7 cores (5.70x in table 6.6) being larger than the parallel speedup. One possible solution for the traditional configuration at this scenario would be to migrate all work to the A15 core, leaving the A7 cores idle and maintaining the higher baseline performance. The TUNE system could also adopt this policy, but in this case, it would need to offload every NEON instruction of the application to the A7 cores, adding a considerable overhead. We have simulated this particular scenario, running the single-threaded application in the A15 core with TUNE, and found that there is an increase in 30% of the execution time when compared to a full A15 core. This is virtually the same slowdown we have observed in the multithreaded execution of this application in figure 6.18.

Applications such as *2mm*, *3mm* and *symm*, have high parallel speedup (about 6x in table 6.6) and small ratios of NEON instructions in the serial region (about 1% in table 6.5). However, the slowdown of executing in the A7 cores is huge(4-5x in table 6.6), at the point that executing these applications in the traditional full core configuration is actually slower than in a single A15 core. This is seen in figure 6.18 as these three applications

are under the normalized performance of the baseline (constant black line). The TUNE system, on the other hand, can overcome the A7 slowdown where the traditional system could not, by exploiting more parallelism with the extra cores. In the case of the *2mm*, the traditional system performance is of 0.69x of a single A15, while the TUNE system achieves 1.27x speedup, as seen in figure 6.18.

The applications that benefit the most from the extra cores are those that show smaller A7 slowdown, a lower ratio of NEON instructions in the serial region, and higher parallel speedups, such as *doitgen*, *gemm*, and *convolution-2d*. These applications show high speedups in the traditional configuration of one A15 core and four A7 cores (up to 2.68x), but even higher in the TUNE configuration (up to 4.67x). As TUNE can provide more cores in the same area, it delivers more thread-level parallelism to these applications, without suffering from the performance loss of offloading instructions.

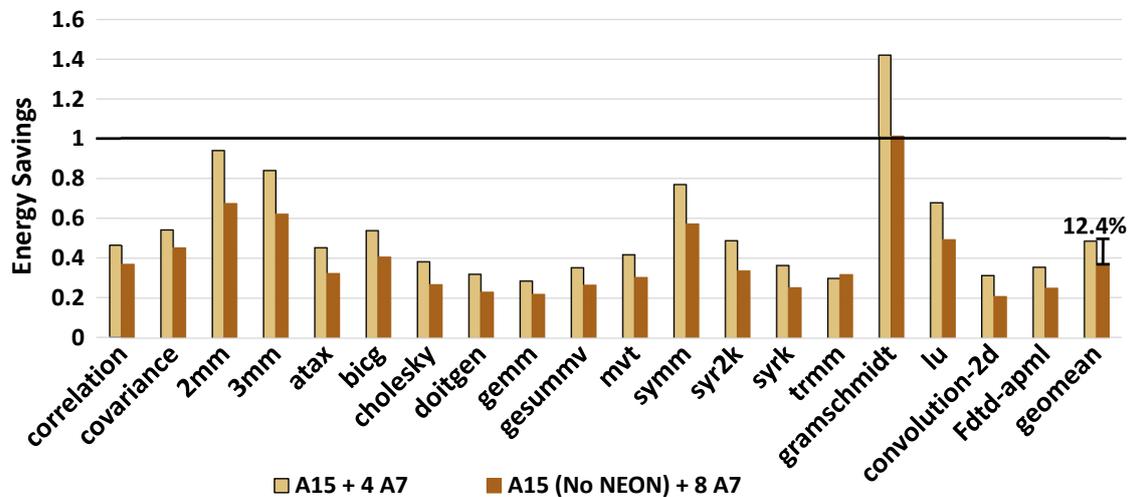
Finally, when one considers the geometric mean of all applications, the traditional system shows an average speedup of 1.43x, while the TUNE system is 2.52x faster than the baseline. Thus, TUNE is 1.76x faster than the traditional heterogeneous system. These average speedups are very close to the ones reported by our model in section 4.3 of the TUNE chapter. *This suggests that the model can be used as a guideline to estimate the performance of the system in a diverse application environment.*

6.2.3 Energy consumption results

Figure 6.19 shows the energy savings for both the traditional configuration and the system with TUNE normalized to a single A15 core. The bars show that the energy consumption in this scenario is proportional to the execution time of each application, as benchmarks with higher degrees of acceleration also show higher energy savings. This results in better energy savings for the TUNE system in almost all applications. Indeed, the only application in which the traditional configuration has lower energy consumption than the TUNE system is *trmm*, which is also the application with a smaller speedup difference between both systems. *gramschmidt* is the only application in which the TUNE system has the same energy consumption as the baseline (single A15), while the traditional configuration consumes more energy, which is directly caused by the bad performance of this configuration.

However, figure 6.19 also shows that energy savings in the TUNE system against the traditional configuration are not as expressive as the speedup in figure 6.18. This is

Figure 6.19: Energy savings for the traditional system and TUNE normalized by the energy the single A15 core. Bars below 1 means that the energy consumption was lower than the baseline.



because power dissipation can be broken into two components: static and dynamic power. While the dynamic power is related to the circuit activity, the static power is associated with the current leakage and is dissipated whenever the circuitry is powered on. As the TUNE system doubles the amount of A7 cores, it also doubles the amount of static power dissipated during the parallel region execution. Considering the energy for offloading, we assume that the NEON units in the A7 have their own power gating domain (as discussed in section 4.2) that allows turning this unit on and off individually, and adding negligible overhead when compared to the original full A15 core. On average, the TUNE system is still capable of delivering energy consumption 12.4% smaller than the traditional AMC system.

6.2.4 Summarizing the results

In this section, we have analyzed the TUNEd PHISA system running parallel applications. We show that maintain binary compatibility through instruction offloading can be an advantageous strategy for these applications. As in the characteristic of parallel applications, SIMD and FP instructions tend to be executed mostly in parallel regions, which reduces the overhead of offloading instructions. By using the extra cores provided by PHISA area reduction, it is also possible to further exploit TLP in these applications, increasing overall performance and decreasing energy consumption. Even in applications high longer serial regions, the overhead of offloading instructions is still smaller than the improvements provided by the extra cores, supporting the advantages of the TUNEd

PHISA system.

6.3 PHISA and TUNE vs the State-of-the-art

In the previous sections, we have analyzed the performance and energy consumption of PHISA, compared to baseline models that represent common processor configurations. In this section, we evaluate this thesis approaches against real processor models and other state-of-the-art proposed solutions.

6.3.1 PHISA vs State-of-the-art

In chapter 2, we have discussed different approaches to heterogeneous processors, that compare to the PHISA system approach. An example is the ARM big.LITTLE technology (ARM, 2016), which uses a mix of big OoO cores and little in-order cores to achieve energy-efficiency, just like PHISA. Differently from PHISA, the big.LITTLE strategy does not exploit functional heterogeneity, replicating all instructions in all cores of the processor. Another example is the reduced-ISA system proposed by Lee et al. (LEE et al., 2017). In this solution, instead of having a mix of big and little cores, the system provides full- and reduced-ISA cores with the same performance, apart that the reduced-ISA do not implement all instructions. Differently from PHISA, this approach assumes that either one of the cores (full or reduced) can be used at a time, and no performance asymmetry exists on the system.

To evaluate PHISA against these related solutions, we have faithfully modeled both the big.LITTLE system and the reduce-ISA approach. To give a competitive boost for the reduced-ISA, we allow it to run workloads on both the full and reduced cores concurrently. All three systems are designed to have the same area budget (a power budget against the reduced-ISA system would result in big.LITTLE and PHISA systems with

Table 6.7: PHISA vs State-of-the-art configurations.

Configuration	A7		A15		Area (mm ²)
	Full	Partial	Full	Partial	
big.LITTLE	2	0	1	0	4.55
SOA(Lee2017)	0	0	1	1	4.70
PHISA	6	0	0	1	4.22

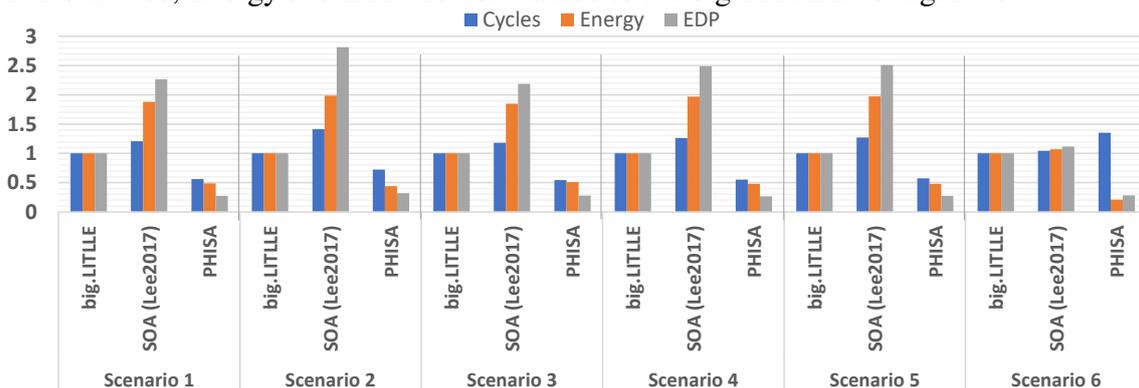
many more cores). Table 6.7 shows the configurations and the total area of each of the processors. All systems have been submitted to the same scenarios described by table 6.1, with both the performance and energy consumption optimization policies from section 3.2.3.

Figure 6.20 shows the results for all scenarios running in the three configurations using a performance optimization policy. The bars represent the number of cycles, energy consumption and EDP, all normalized by the big.LITTLE configuration. Therefore, bars above one represent worse results (more cycles to execute, more energy spent and higher EDP), while bars below one represent better results.

The figure 6.20 shows that the state-of-the-art approach proposed by (LEE et al., 2017) can be actually worse in performance, energy consumption and EDP than a big.LITTLE processor of same area. When this approach was proposed, the authors debated that the reduce-ISA core, coupled with a full core, could present better energy consumption than a single full core. Therefore, differently from our baseline, the authors were not considering an area budget to compare processors, but the amount of cores that can be active in the system concurrently. The big.LITTLE processor has more cores (3 against 2), that are actually more energy efficient (full little cores are more efficient than big reduced-ISA) than the reduced cores, providing better performance and energy consumption. The only scenario in which the SOA have similar performance to the baseline is in scenario 6. This is a scenario in which the little cores have a much lower performance than the big cores, thus running the workloads on more of them do not provide higher throughput.

On the other hand, figure 6.20 shows that PHISA can deliver better performance,

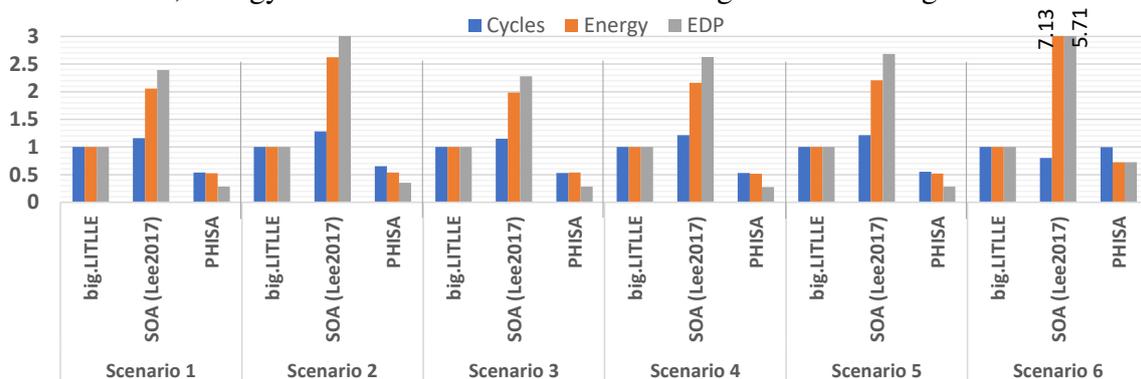
Figure 6.20: Evaluation of PHISA multicores and state-of-the-art (LEE et al., 2017) against a big.LITTLE baseline under a $4.7mm^2$ area budget. Scheduling of tasks follows a performance optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the big.LITTLE configuration.



energy consumption and EDP than both state-of-the-art approaches in almost every scenario. PHISA achieves better performance by the extra cores that the system can hold in the same area, along with the more energy-efficient OoO core (partial-ISA A15) present in this system. Scenario 6 is the only one in which PHISA shows worse performance than the compared state-of-the-art, as the high NEON usage and differences in performance of NEON execution in the A15 vs the A7 are more evident.

Figure 6.21 shows the results for the three configurations, but running under a energy consumption optimization policy. In this policy, small cores are given priority during the allocation phase over the big cores (to potentially reduce energy consumption). Nonetheless, the State-of-the-art (SOA) configuration does not have small cores, and will give priority to its reduced-ISA core. This is the reason why the performance of the SOA in this policy gets closer to the performance of the baseline. On the other hand, the lack of small cores directly affects the energy consumption of the SOA system, being much worse than the baseline. In scenario 6, this becomes extremely evident, as the energy consumption skyrockets. As the baseline executes most of its workloads in its small cores (due to policy), but the SOA can only execute in big cores, heavily increasing its energy consumption. Besides the poor results of the SOA, PHISA can still show better (or at least the same) performance and energy consumption when compared to the baseline, thanks to its ability to use both the advantages of the big.LITTLE system and the SOA.

Figure 6.21: Evaluation of PHISA multicores and state-of-the-art (LEE et al., 2017) against a big.LITTLE baseline under a $4.7mm^2$ area budget. Scheduling of tasks follows a energy consumption optimization policy for all configurations, including the baseline. Performance, Energy and EDP are normalized to the big.LITTLE configuration.



6.3.2 TUNEd PHISA vs State-of-the-art

The TUNE architecture approach is based in the idea that specific instruction set extensions do not have to be implemented in every core. Furthermore, some of this instructions, such as FP and SIMD can be offloaded from partial-ISA cores to one (or multiple) full-ISA cores. We have discussed in chapter 2 other processors that also share or offload instructions to external processing units. One of this works is the Niagara processors (UltraSparc T1 (SUN, 2019)), which is composed of 8 simple cores without FP capabilities and a single FP unit loosely coupled with them. In this section, we compare a faithfully modeled Niagara-like processor with a TUNEd PHISA system executing multi-threaded applications.

To model the Niagara-like processor, we use 8 in-order cores (modeled as A7 cores) that have increased latency to execute FP and SIMD operations. As all the 8 Niagara cores have to access a single FPU through a crossbar, we model this latency as 20 cycles, which is double the latency of the TUNE access. TUNE is tightly coupled and is designed to use dedicated wiring to connect the partial-ISA core to the units in the full cores. Furthermore, while in the TUNEd PHISA one partial A15 core can use up to 4 NEON units in the A7s, in the Niagara 8 cores have to share a single FP unit, which has a huge chance of causing resource contentions. Therefore, we believe that considering double the latency for the Niagara to access the FPU (compared to TUNE) is a fair assumption. Nonetheless, it is important to notice that the Niagara system works with a pool of threads, and that a new thread can be assigned to an integer core that has requested an FP operation. Unfortunately, our simulations can not model this behavior.

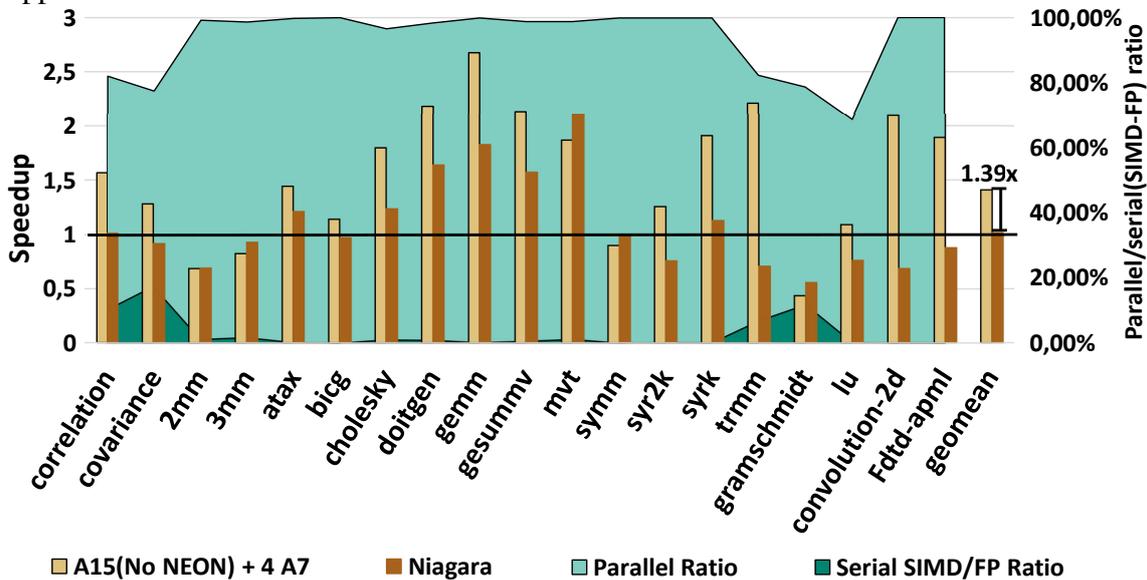
We compare this Niagara-like processor of 8 cores against TUNEd PHISA processor of same area budget. The Niagara area is modeled as 7 A7 cores without NEON units plus 1 A7 core with NEON unit, while the TUNEd PHISA is 1 A15 core without NEON along with 4 A7 cores with NEON. Table 6.8 shows the configurations of these systems.

Figure 6.22 shows the results for performance of both Niagara and TUNEd PHISA

Table 6.8: TUNEd PHISA vs Niagara configurations.

Configuration	A7		A15		Area (mm ²)
	Full	Partial	Full	Partial	
Baseline	0	0	1	0	3.52
Niagara	1	7	0	0	3.20
TUNEd PHISA	4	0	0	1	3.20

Figure 6.22: Speedup for the state-of-the-art Niagara system and TUNE normalized. Bars are the speedup over a single A15 core and areas are the % of *PR* and *SRF* of each application.



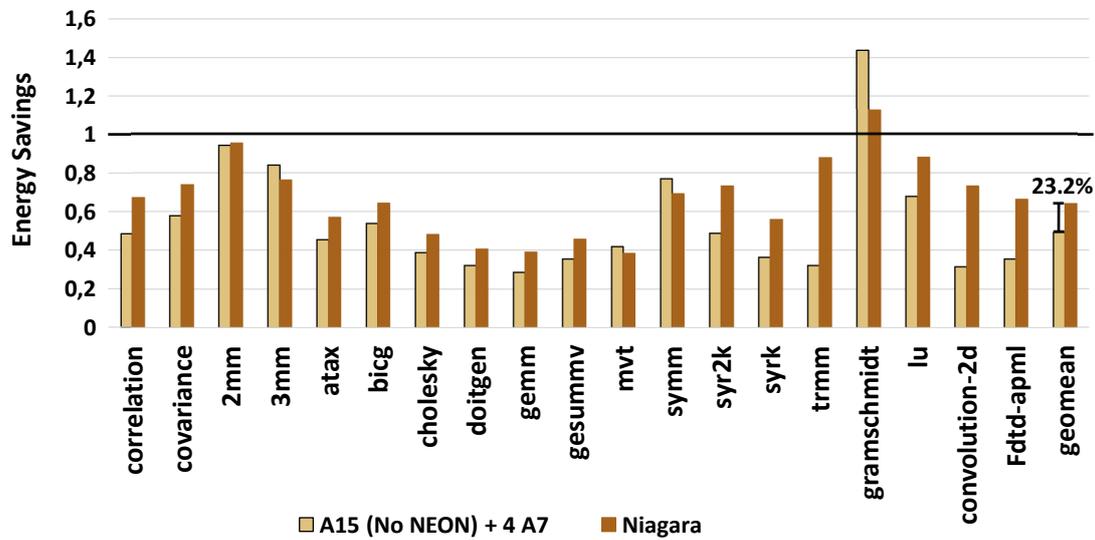
normalized by the single A15 processor. While the Niagara system can potentially exploit more TLP because of its extra cores, it also pays a heavy latency price in every FP and SIMD instruction executed. On the other hand, the TUNEd PHISA can still exploit TLP, but only suffers overhead in these instructions during serial regions, when they are only rarely used. The result is that the TUNEd PHISA has better performance than Niagara in 14 of the 19 tested benchmarks. The scenarios in which the Niagara performs better than TUNEd PHISA are either when the application has big parallel regions with low FP usage (such as in *3mm* and *mvt*) or when the serial region is too long and uses many FP operations (which causes high overhead in TUNE, such as in *gramschmidt*). In average, the TUNEd PHISA is still 1.39x faster than the Niagara system.

Figure 6.23 shows the energy savings of both the Niagara and TUNEd PHISA, normalized by the single A15 core. The TUNEd PHISA system shows better energy savings than the Niagara in 15 of the 19 benchmarks, showing that using a big core to accelerate the serial regions does not compromise the energy efficiency of the processor. In average, the energy savings of the TUNEd PHISA over Niagara are of 23%.

6.3.3 Summarizing the results

In this section, we have analyzed both the PHISA system for single-threaded applications and the TUNEd PHISA for multi-threaded applications against state-of-the-art

Figure 6.23: Energy savings for the state-of-the-art Niagara system and TUNE normalized by the energy the single A15 core. Bars below 1 means that the energy consumption was lower than the baseline.



systems. We have shown the advantages of our system designs and how they can deliver better performance and energy consumption under the same constraints. The PHISA system shows better performance and energy consumption than the big.LITTLE (ARM, 2016) and the reduced-ISA (LEE et al., 2017), as it can pack more efficient cores in the same area. On the other hand, the TUNEd PHISA shows better overall performance and energy consumption than the Niagara processor (SUN, 2019), as it can provide more support to ISA extensions, reducing resource contention.

7 CONCLUSIONS

In this section, we wrap-up all the conclusions provided by the systems developed in this thesis. Both our PHISA and TUNEd PHISA systems leverage the extra resources provided by the use of partial-ISA in some cores of a processor. However, each of them tackles the problem of maintaining binary compatibility using different but complementary strategies. Therefore, we split our conclusions not only focusing on the developed systems but also in their strategies for achieving transparent ISA compatibility. We then finish the section discussing some of the open challenges that this thesis introduces for future research.

7.1 On PHISA - binary support through migration

The PHISA multicores is a processor composed of cores that implement the full architectural ISA and other cores that implement such ISA partially. The partial cores are envisioned as cores that can deliver the same performance as the full cores, but are not able to execute a determined set of instructions. Therefore, to maintain the binary compatibility between all the cores in the processor, the system must be aware that cores can fail to execute some instructions - without this recurring into a kill signal - and must be able to migrate the faulting workload to cores that can execute these instructions. We have seen that this scheduling process of workloads is an essential part of the PHISA system. Depending on the decisions of the scheduler, the overhead of migrating threads can be highly reduced.

One of the strategies that we have explored to reduce migration overhead is the emulation of non-supported instructions through software. Emulation proved to be efficient in reducing the number of re-allocations, although it could introduce new overheads in the system. Naturally, emulation through software has a high cost in performance when compared to execution in hardware. Therefore, it should only be used in scenarios where the number of faulting instructions is small or highly scattered throughout the code. The system must be able to analyze the trade-off between the emulation cost and the migration cost and determine which one is lower to use emulation efficiently.

Further observations on the scheduler showed that by supporting binary compatibility through migration, it is possible to optimize the system for different non-functional requirements. In our experiments, we showed that scheduler policies could be introduced

in our system without a negative impact on overall performance. In other words, we can still optimize a PHISA multicore for - for instance - performance or energy consumption and still be better than traditional solutions optimized by the same requirements.

On another point, the instructions removed from partial-ISA cores must be in a set that is both normally not used by general applications, and that introduces a high impact in either power or area in the processor. This is essential not only to reduce the amount of migrations required in the system but also to create power and area efficient cores. If a common set of instructions is removed, or a set that uses few resources in the core (e.g., the DSP instructions in an ARM core), the partial implementation would hardly provide any benefits. Furthermore, as the PHISA system uses the extra area to increase the core count of the system, it becomes even more important that the impact in the area of the instruction set removed should be high.

In the current ISAs state, the sets of instructions that better fit these characteristics are the FP and SIMD (or vector) instructions. However, trends in applications suggest that other high resource-demanding - and highly specialized - instructions should appear as common options in future ISAs, such as instructions for Neural Network (NN) and Artificial Intelligence (AI) processing.

Furthermore, the same instruction extension can have a different impact on different processor organizations. This depends on various factors, such as the performance of the operation in the processor or its throughput capacity. For instance, the impact of the NEON instructions in the A15 core is proportionally much higher than in the A7 core. That is because the A15 core implements two-issue lanes of NEON (against one in the A7) and a twice wider SIMD operation (128-bits against 64-bits). These differences can become even more evident in the upcoming SVE instructions, which allow vector operations of up to 2048-bits wide operands. With that being said, we have seen that some cores have characteristics that make them better targets for removing support of instructions.

Overall, supporting compatibility through migration has shown an interesting level of flexibility and good efficiency for single-threaded workloads. As in this type of applications there are no direct dependencies¹, the overhead of preempting applications is low.

¹Some of our scenarios were modeled to execute in pipeline, which introduces some degree of dependency

7.2 On TUNEd PHISA - binary support through offloading

We argue that we can build a PHISA system just like traditional AMCs for parallel applications, where a single high-performance core is used to accelerate serial regions, and many small cores are used to execute parallel regions. As parallel regions (hopefully) represent larger sections of the application code, we use PHISA to create a partial-ISA big core, and with the extra area, we introduce more small cores for these regions.

Another characteristic of the parallel applications that we observed is that operations of FP and SIMD type are usually executed in the parallel regions. This also motivates removing these instructions from the big core, as they are only sporadically required in serial regions. However, we also discussed how the migration strategy could not be suitable for every type of parallel applications, as allocating threads to different cores could create unbalanced threads that would lead to delayed synchronization/join points.

Therefore, to use PHISA to optimize parallel applications, we have developed TUNE, which tackles the compatibility challenge differently from the traditional PHISA. TUNE is an instruction offloader that is implemented in place of the traditional FP/SIMD instructions. It is responsible for splitting the SIMD operations and forwarding them directly to the SIMD units of the small (full) cores of the system.

This solution proved efficient in the real implementation using ARM A15 and A7 cores. As one NEON unit of the A15 core has the same area of 4 A7 cores, we can highly increase the number of cores used to accelerate the parallel regions. Also, each lane of NEON in the A15 core works with operands of 128-bits, while the A7 works with operands of 64-bits. Therefore, it is possible to split each vector instruction of the A15 lane in two and directly assign (without any further transformation) these to two A7 cores. As the A15 cores have two NEON lanes, all the four added A7 cores could be used to offload the NEON instructions from the A15 core.

Another advantage of the TUNEd PHISA is that all modifications are made directly into the hardware. The OS only needs to implement a standard solution for AMCs of this type - i.e., that can identify serial regions to run in the big core and parallel regions to run in the small cores. Therefore, there is no need for a novel scheduler with faulting support as the migration strategy required.

The TUNEd PHISA showed good improvements when compared to other AMC, as it can deliver more cores in the same area. Even when compared to a many-core processor in the state-of-the-art that also uses instruction offloading, the TUNEd PHISA

showed average better performance. This is because, differently from other approaches that share a single FP/SIMD unit between many cores, our solution actually provides many of these units to a single core. The results are that with TUNE, there is no resource contention and that the instructions can be split into many units, which helps to reduce the overhead in offloading the instruction.

7.3 Limitations

We would like to acknowledge some limitations that come from the results of this thesis and that should be kept in mind in any future works related to it. First, all of our evaluation has been based in models and simulations. Although we have always tried to adopt the most commonly used tools in the field, and also striven for validating their data with public information, the results can not be taken as final numbers. Second, mixing results from different tools (McPAT, gem5, PHISA Simulator) will mostly certainly introduce inconsistencies in the final results. We rely in the high gains that we observe in most configurations and scenarios (in both PHISA and glsTUNEd PHISA) to assume that, when this inconsistencies are introduced to a real world system, it will still show improvements. Third and last, our analytical models have not been statistically analyzed. Although they show an average consistency with the simulated results, they could be tied only to the specific applications used in the analysis.

7.4 Open Challenges

In this section, we briefly describe some of the remaining open challenges introduced by this thesis, and that can be addressed in future works. The goal is not to provide solutions, but to discuss these problems in a high-level approach.

Dynamic analysis of emulation VS migration cost: In the current PHISA implementation, the scheduler can signal a faulting instruction to be emulated in a partial-ISA core. This is quite useful when the faulting instruction is not followed by many extra unsupported operations, as it avoids unnecessary workload migration. However, this decision is statically made: the application will always emulate the sequence of instructions for a fixed amount of cycles. If the unsupported instructions are still being issued after that time, the scheduler will migrate the workload to a full core. When this scenario happens,

the system is actually paying both the cost of emulating the instructions and migrating the workload, when the best decision would have been to migrate the workload directly (avoiding the emulation cost).

Therefore, one open challenge is to create a dynamic system, which is able to predict if the best decision is to migrate a workload or to emulate a faulting instruction. This could be done by profiling the application (either offline or during runtime) and learning the behaviors of its phases. Nonetheless, this profile could introduce overheads by itself, which should be considered by the researchers.

New applications and instruction mix: Although this thesis has striven for a varied and representative set of applications, newer benchmarks with a higher instruction mix could be analyzed. Benchmarks or real applications that stress more SIMD and FP operations, or even trending applications that do not use this instructions (which would run great in the PHISA system) could be added to the list. This would be interesting to analyze and verify if the current application set is not biased for our proposed system.

Impact in single-threaded applications using TUNE: The TUNEd PHISA AMC was explicitly designed in this work to accelerate parallel applications. Although this system should be able to run single-threaded applications, it definitely would experience performance losses on them. This is because the processor would either have to execute the applications in the full-ISA small (and low performance) cores, or in the partial-ISA big cores, which have an extra overhead for executing FP/SIMD operations.

One interesting research possibility would be to find a solution to mask the offloading overhead in the big cores of a TUNEd PHISA. This would be interesting not only for the execution of the serial regions in parallel applications but also to improve the execution of single-thread applications. The current TUNEd PHISA design keeps all SIMD related components in the big core, to reduce the complexity of the offloading. This includes data in the caches and register files. If this data was directly available in the A7 cores, some offloading costs could be amortized. For instance, if the FP register file of the A15 are the register file from the A7 themselves, the amount of data transferred would be smaller.

Another possibility would be to improve vector operations performance by using length agnostic instructions such as SVE. A designer could create a morphcore-like architecture that can fuse or split NEON units from different A7 cores. This could also be a dynamic operation, which could provide varying support for wider vectors VS many cores support.

Impact of the latency: This thesis has assumed a fixed latency of 10 cycles to execute the offloaded instructions. However, we did not analyze the real impact in this extra latency. How much can this latency increase and TUNE still be a viable option for AMCs?

Allowing the use of big cores in the parallel region: In this work we have restricted the usage of big cores in TUNE to accelerate serial regions. However, this extra core could be used to run the parallel region too, increasing the TLP of the system. This would be very interesting in applications that have unbalanced threads, in which higher demanding threads can take too long to execute and become a bottle neck of the application. These threads could be accelerated in the big core too. A future work could analyze this scenario, but it would involve the development of a scheduler capable of dealing with this type of threads and also the offloading problem: during parallel regions, the A7s are busy. How to offload operation to them in this situation?

Merging PHISA and TUNEd PHISA designs: In this work, PHISA for single-threaded workloads and TUNEd PHISA for parallel applications were presented as different designs. However, they are actually complementary solutions. A TUNEd PHISA is still a PHISA system, but with the TUNE offloader.

It is possible to use both the migration and offloading strategies in the PHISA system. However, some extra modifications would be required. In the TUNEd PHISA, the partial cores still implement the decoding logic for the unsupported operations. Therefore, an extra verification would be necessary for the core to throw an exception for the scheduler if it is running a single-thread application. Furthermore, a big core might require to offload some serial region (of a parallel application) instruction while the small cores are being used to execute other single-threaded workloads. In this case, either a scheduling decision to halt these workloads should be issued, or some hardware interruption required so that the NEON units in the small cores are made available for offloading.

8 PUBLICATIONS

The results of this thesis have been published in three different articles:

- Souza, J. D. and Beck, A. C. (2019). Trimming the ISA to Optimize Area and EDP in Heterogeneous CMPs. 2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 17–24. (SOUZA; FILHO, 2019) - This publication introduced the PHISA system for single-threaded workloads. It contains most of the results presented in section 6.1. This article has received **the best paper award** of the conference.
- Souza, J. D.; Manivannan, M.; Pericàs, M. and Beck, A. C. S. (2020). Enhancing Multithreaded Performance of Asymmetric Multicores with SIMD Offloading. In: 23rd Design, Automation and Test in Europe Conference (DATE'20) (SOUZA et al., 2020a) - The article introduces the TUNEd PHISA and most of the performance results showed in section 6.2.2.
- Souza, J. D.; Manivannan, M.; Pericàs, M. and Beck, A. C. S. (2020). Enhancing Thread-Level Parallelism in Asymmetric Multicores using Transparent Instruction Offloading. In: 57th ACM/EDAC/IEEE Design Automation Conference (DAC'20) (SOUZA et al., 2020b)¹ - This publication presents all the concepts of the TUNED PHISA for multi-threaded applications. It includes the performance model (section 4.3), and the results from the simulated environments, both for performance and energy (section 6.2.2).

As an indirect result of (but related to) this thesis, the Ph.D. has collaborated in the following works:

- Becker, P. H. E.; Souza, J. D.; Beck, A. C. S. (2019). Increasing MPSoCs designspace with partial-ISA processors. In: 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS). (BECKER; SOUZA; BECK, 2019) - This publication evaluates a scenario in which ISA extensions give space to different accelerators in an MPSoC.
- Becker, P. H. E.; Souza, J. D.; Beck, A. C. S. (2020) Tuning the ISA for increased heterogeneous computation in MPSoCs. In: 23rd Design, Automation and Test in Europe Conference (DATE20). (BECKER; SOUZA; BECK, 2020) - The article extends the previous work by analyzing different scheduling policies and EDP scal-

¹This paper was accepted for publication and will be presented during the conference in July 2020

ability.

Not directly related to this thesis, the Ph.D. has published the following works:

- Souza, J. D.; Carro, L.; Rutzig, M. B. and Beck, A. C. S. (2014) Towards a Dynamic and Reconfigurable Multicore Heterogeneous System. In: Brazilian Symposium on Computing Systems Engineering, p. 73–78 (SOUZA et al., 2014) - A performance analysis of a multicore heterogeneous system comprised of reconfigurable hardware, using a static scheduler.
- Souza, J. D.; Cachola, J. V. G.; Carro, L.; Rutzig, M. B. and Beck, A. C. S. (2016). Evaluating schedulers in a reconfigurable multicore heterogeneous system. In: International Symposium on Applied Reconfigurable Computing (SOUZA et al., 2016) - An evaluation of different scheduling methods in the reconfigurable multicore heterogeneous system.
- Souza, J. D.; Carro, L.; Rutzig, M. B.; and Beck, A. C. S. (2016). A Reconfigurable Heterogeneous Multicore with a Homogeneous ISA. Proceedings of the 2016 Conference on Design, Automation & Test in Europe (DATE'16), 1598–1603. (SOUZA et al., 2016) - Full performance and energy analysis of the reconfigurable multicore heterogeneous system using an Oracle scheduler for measuring the system potential.
- Souza, J. D.; Sartor, A. L.; Carro, L.; Rutzig, M. B.; Wong, S. and Beck, A. C. S. (2018). DIM-VEX: Exploiting Design Time Configurability and Runtime Reconfigurability. In: International Symposium on Applied Reconfigurable Computing (SOUZA et al., 2018) - An evaluation of performance and energy in a system with reconfigurable hardware coupled to configurable processors.

Moreover, the Ph.D. has also published the following works in project cooperations:

- (LORENZON; Dellagostin Souza; BECK, 2017) - A library for automatic optimization of thread count in OpenMP applications.
- (LORENZON et al., 2018) - An OpenMP extension for seamless and dynamic optimization of thread count in parallel applications.
- (SILVEIRA et al., 2016) - Image processing acceleration using approximation and function reuse.
- (BRANDALERO et al., 2017) - Acceleration of error-tolerant applications using

memoization and input value hashing.

- (ERICHSEN et al., 2018) - Exploiting big.LITTLE-like processors to provide Diversity Triple Modular Redundancy (TMR) and fault tolerance.
- (De Moura et al., 2016) - Extends the previous works in reconfigurable multicore heterogeneous processors to evaluate unified context caches.
- (SFREDDO et al., 2017) - A framework to create efficient heterogeneous configurations for reconfigurable hardware.

REFERENCES

ADEGBIJA, T. et al. Microprocessor Optimizations for the Internet of Things: A Survey. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 37, n. 1, p. 7–20, jan 2018. ISSN 0278-0070. Available from Internet: <<http://ieeexplore.ieee.org/document/7954016/>>.

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: ACM. **Proceedings of the spring joint computer conference**. [S.l.], 1967.

AMINOT, A. et al. FPU Speedup Estimation for Task Placement Optimization on Asymmetric Multicore Designs. **2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip**, p. 81–87, 2015. Available from Internet: <<http://ieeexplore.ieee.org/document/7328190/>>.

ANNAMALAI, A. et al. An Opportunistic Prediction-Based Thread Scheduling to Maximize Throughput/Watt in AMPs. In: **Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques**. [S.l.]: IEEE Press, 2013. (PACT '13), p. 63–72. ISBN 9781479910212.

ARM. **Cortex-A9 NEON Media Processing Engine - Instruction Timings**. [S.l.], 2010. Available from Internet: <<http://www.arm.com>>.

ARM. **Cortex-A7 – Arm Developer**. 2011. Available from Internet: <<https://developer.arm.com/ip-products/processors/cortex-a/cortex-a7>>.

ARM. **big.LITTLE Technology**. 2016. Available from Internet: <<https://developer.arm.com/technologies/big-little>>.

ARM. **Technologies | DynamIQ – Arm Developer**. 2018. Available from Internet: <<https://developer.arm.com/technologies/dynamiq>>.

ARM Ltd. White Paper: big. LITTLE Technology : The Future of Mobile. p. 12, 2013.

BECK, A. C. S.; Lang Lisbôa, C. A.; CARRO, L. **Adaptable Embedded Systems**. 1. ed. New York, NY: Springer New York, 2013. ISBN 978-1-4614-1745-3. Available from Internet: <<http://link.springer.com/10.1007/978-1-4614-1746-0>>.

BECKER, P. **Selectively supporting ISA-extensions to enhance heterogeneous MPSoC designs under power and area constraints**. Dissertation (Master) — Universidade Federal do Rio Grande do Sul, Brazil, 2019.

BECKER, P. H. E.; SOUZA, J. D.; BECK, A. C. S. Increasing MPSoCs design space with partial-ISA processors. In: **2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)**. [S.l.]: Institute of Electrical and Electronics Engineers (IEEE), 2019. p. 678–681. ISBN 9781728109961.

BECKER, P. H. E.; SOUZA, J. D.; BECK, A. C. S. Tuning the ISA for increased heterogeneous computation in MPSoCs. In: **23rd Design, Automation and Test in Europe Conference**. [S.l.: s.n.], 2020.

BIENIA, C. **Benchmarking Modern Multiprocessors**. Thesis (PhD) — Princeton University, 2011.

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>>.

BLEM, E.; MENON, J.; SANKARALINGAM, K. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In: **2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)**. IEEE, 2013. p. 1–12. ISBN 978-1-4673-5587-2. Available from Internet: <<http://ieeexplore.ieee.org/document/6522302/>>.

BLEM, E. et al. ISA Wars: Understanding the Relevance of ISA being RISC or CISC to Performance, Power, and Energy on Modern Architectures. **ACM Transactions on Computer Systems**, v. 33, n. 1, p. 1–34, 2015. ISSN 07342071. Available from Internet: <<http://dl.acm.org/citation.cfm?id=2745713.2699682>>.

BORIN, E.; WU, Y. Characterization of dbt overhead. In: **Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009**. [S.l.: s.n.], 2009. p. 178–187. ISBN 9781424451562.

BORKAR, S. Thousand Core ChipsA Technology Perspective. In: **2007 44th ACM/IEEE Design Automation Conference**. IEEE, 2007. p. 746–749. ISBN 978-1-59593-627-1. ISSN 0738-100X. Available from Internet: <http://www.crossref.org/deleted{_}DOI.h>.

BRANDALERO, M. et al. Accelerating error-tolerant applications with approximate function reuse. **Science of Computer Programming**, 2017. ISSN 01676423.

BROWN, J. A.; PORTER, L.; TULLSEN, D. M. Fast thread migration via cache working set prediction. In: **2011 IEEE 17th International Symposium on High Performance Computer Architecture**. IEEE, 2011. p. 193–204. ISBN 978-1-4244-9432-3. Available from Internet: <<http://ieeexplore.ieee.org/document/5749728/>>.

BULLDOZER, W. A. **AMD Bulldozer block diagram (CPU core block) - Bulldozer (microarchitecture) - Wikipedia**. 2011. Available from Internet: <[https://en.wikipedia.org/wiki/Bulldozer{_}\(microarchitecture\){\#}/media/File:AMD{_}Bulldozer{_}block{_}diagram{_}\(C\)](https://en.wikipedia.org/wiki/Bulldozer{_}(microarchitecture){\#}/media/File:AMD{_}Bulldozer{_}block{_}diagram{_}(C))>.

CEBRIAN, J. M.; JAHRE, M.; NATVIG, L. ParVec: vectorizing the PARSEC benchmark suite. **Computing**, Springer-Verlag Wien, v. 97, n. 11, p. 1077–1100, nov 2015. ISSN 0010485X.

CONG, J.; YUAN, B. Energy-efficient scheduling on heterogeneous multi-core architectures. In: **Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design - ISLPED '12**. New York, New York, USA: ACM Press, 2012. p. 345. ISBN 9781450312493. Available from Internet: <<http://dl.acm.org/citation.cfm?doid=2333660.2333737>>.

CONSTANTINOU, T. et al. Performance implications of single thread migration on a chip multi-core. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, 2005.

De Moura, R. et al. The impact of heterogeneity on a reconfigurable multicore system. In: **Proceedings of IEEE Computer Society Annual Symposium on VLSI, ISVLSI**. [S.l.: s.n.], 2016. v. 2016-Sept. ISBN 9781467390385. ISSN 21593477.

ENDO, F. A.; COUROUSSÉ, D.; CHARLES, H. P. Micro-architectural simulation of embedded core heterogeneity with gem5 and McPAT. **ACM International Conference Proceeding Series**, v. 19-21-Janu, p. 1–6, 2015. Available from Internet: <<http://dl.acm.org/citation.cfm?doid=2693433.2693440>>.

ERICHSEN, A. G. et al. ISA-DTMR: Selective Protection in Configurable Heterogeneous Multicores. In: . Springer, Cham, 2018. p. 231–242. Available from Internet: <<http://link.springer.com/10.1007/978-3-319-78890-6>>.

FOG, A. **The microarchitecture of Intel, AMD and VIA CPUs An optimization guide for assembly programmers and compiler makers**. [S.l.], 1996. Available from Internet: <<https://www.agner.org/optimize/microarchitecture.pdf>>.

FRITTS, J. E. et al. MediaBench II video: Expediting the next generation of video systems research. **Microprocessors and Microsystems**, Elsevier, v. 33, n. 4, p. 301–318, jun 2009. ISSN 0141-9331. Available from Internet: <<https://www.sciencedirect.com/science/article/abs/pii/S014193310900026X>>.

GNU. **Auto-vectorization in GCC - GNU Project - Free Software Foundation (FSF)**. 2020. <<https://gcc.gnu.org/projects/tree-ssa/vectorization.html>>.

GRAUER-GRAY, S. et al. Auto-tuning a high-level language targeted to GPU codes. In: **2012 Innovative Parallel Computing (InPar)**. IEEE, 2012. p. 1–10. ISBN 978-1-4673-2633-9. Available from Internet: <<http://ieeexplore.ieee.org/document/6339595/>>.

GUTHAUS, M. et al. MiBench: A free, commercially representative embedded benchmark suite. In: **Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)**. IEEE, 2001. p. 3–14. ISBN 0-7803-7315-4. Available from Internet: <<http://ieeexplore.ieee.org/document/990739/>>.

GUTIERREZ, A.; DRESLINSKI, R. G.; MUDGE, T. Evaluating private vs. shared last-level caches for energy efficiency in asymmetric multi-cores. In: **2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)**. IEEE, 2014. p. 191–198. ISBN 978-1-4799-3770-7. Available from Internet: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6893211>>.

HILL, M. D.; MARTY, M. R. Amdahl's Law in the Multicore Era. **Computer**, v. 41, n. 7, p. 33–38, jul 2008. ISSN 0018-9162. Available from Internet: <<http://ieeexplore.ieee.org/document/4563876/>>.

HPLABS. **HP Labs : McPAT**. 2009. Available from Internet: <<https://www.hpl.hp.com/research/mcpat/>>.

HRUBY, T.; BOS, H.; TANENBAUM, A. S. When Slower is Faster: On Heterogeneous Multicores for Reliable Systems. In: **Proceedings of the 2013 USENIX Conference on Annual Technical Conference**. Berkeley, CA, USA:

USENIX Association, 2013. (USENIX ATC'13), p. 255–266. Available from Internet: <<http://dl.acm.org/citation.cfm?id=2535461.2535493>>.

HRUSKA, J. **ARM Cortex-A15 explained: Intel's Atom is down, but not out - ExtremeTech**. 2012. Available from Internet: <<http://www.extremetech.com/computing/139393-arm-cortex-a15-explained-intels-atom-is-down-but-not-out>>.

IEEE-754. **IEEE 754-2008 - IEEE Standard for Floating-Point Arithmetic**. 2008. Available from Internet: <<https://standards.ieee.org/standard/754-2008.html>>.

IPEK, E. et al. Core fusion: accommodating software diversity in chip multiprocessors. In: **Proceedings of the 34th annual international symposium on Computer architecture - ISCA '07**. New York, New York, USA: ACM Press, 2007. v. 35, n. 2, p. 186. ISBN 9781595937063. ISSN 0163-5964. Available from Internet: <<http://portal.acm.org/citation.cfm?doid=1250662.1250686>>.

JEFF, B. big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling. p. 1–13, 2013.

KHAN, O.; KUNDU, S. A self-adaptive scheduler for asymmetric multi-cores. In: **Proceedings of the 20th symposium on Great lakes symposium on VLSI - GLSVLSI '10**. New York, New York, USA: ACM Press, 2010. p. 397. ISBN 9781450300124. Available from Internet: <<http://portal.acm.org/citation.cfm?doid=1785481.1785573>>.

KHUBAIB et al. MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. In: **2012 45th Annual IEEE/ACM International Symposium on Microarchitecture**. IEEE, 2012. p. 305–316. ISBN 978-1-4673-4819-5. Available from Internet: <<http://ieeexplore.ieee.org/document/6493629/>>.

KNAUERHASE, R.; BRETT, P. Kinship : Efficient Resource Management for Performance and Functionally Asymmetric Platforms Categories and Subject Descriptors. n. 1, 2013.

KOPPANALIL, J. et al. A 1.6 GHz dual-core ARM Cortex A9 implementation on a low power high-K metal gate 32nm process. In: **Proceedings of 2011 International Symposium on VLSI Design, Automation and Test, VLSI-DAT 2011**. [S.l.: s.n.], 2011. p. 239–242. ISBN 9781424484997.

KUMAR, R. et al. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. **Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.**, 2003.

KUMAR, R.; JOUPPI, N. P.; TULLSEN, D. M. **Conjoined-core Chip Multiprocessing**. [S.l.], 2004.

LEE, W. et al. Exploring Heterogeneous-ISA Core Architectures for High-Performance and Energy-Efficient Mobile SoCs. **Proceedings of the on Great Lakes Symposium on VLSI 2017 - GLSVLSI '17**, p. 419–422, 2017. Available from Internet: <<http://dl.acm.org/citation.cfm?doid=3060403.3060408>>.

LI, S. et al. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In: **MICRO 42: Proceedings of the 42nd**

Annual IEEE/ACM International Symposium on Microarchitecture. [S.l.: s.n.], 2009. p. 469–480.

LI, T. et al. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In: **ACM/IEEE Conference on Supercomputing.** [S.l.: s.n.], 2007.

LI, T.; BRETT, P.; KNAUERHASE, R. Operating system support for overlapping-ISA heterogeneous multi-core architectures. **Proceedings - International Symposium on High-Performance Computer Architecture (HPCA)**, p. 1–12, 2010. ISSN 1530-0897. Available from Internet: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5416660>{\%}5Cnhttp://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=5>.

LIY, Y. et al. CMP Design Space Exploration Subject to Physical Constraints. In: **The Twelfth International Symposium on High-Performance Computer Architecture, 2006.** IEEE, 2006. p. 15–26. ISBN 0-7803-9368-6. Available from Internet: <<http://ieeexplore.ieee.org/document/1598109/>>.

LOPES, B. C. et al. Shrink. **Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15**, p. 311–322, 2015. ISSN 0163-5964. Available from Internet: <<http://dl.acm.org/citation.cfm?id=2749469.2750391>>.

LORENZON, A. F. et al. Aurora: Seamless Optimization of OpenMP Applications. **IEEE Transactions on Parallel and Distributed Systems**, p. 1–1, 2018. ISSN 1045-9219. Available from Internet: <<https://ieeexplore.ieee.org/document/8477128/>>.

LORENZON, A. F.; Dellagostin Souza, J.; BECK, A. C. S. LAANT: A library to automatically optimize EDP for OpenMP applications. In: **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017.** IEEE, 2017. p. 1229–1232. ISBN 978-3-9815370-8-6. Available from Internet: <<http://ieeexplore.ieee.org/document/7927176/>>.

LORENZON, A. F. et al. Aurora: Seamless Optimization of OpenMP Applications. **IEEE Transactions on Parallel and Distributed Systems**, v. 30, n. 5, p. 1007–1021, may 2019. ISSN 1045-9219. Available from Internet: <<https://ieeexplore.ieee.org/document/8477128/>>.

LOWE-POWER, J. **gem5: Learning gem5.** 2020. Available from Internet: <https://www.gem5.org/documentation/learning{_}gem5/introducti>.

LUKEFAHR, A. et al. Composite Cores: Pushing Heterogeneity Into a Core. In: **2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.** IEEE, 2012. p. 317–328. ISBN 978-1-4673-4819-5. Available from Internet: <<http://ieeexplore.ieee.org/document/6493630/>>.

MALLIA, L. **ARM Developers Conference 2007 Qualcomm High Performance Processor Core and Platform for Mobile Applications.** [S.l.], 2007. Available from Internet: <http://rtcgroup.com/arm/2007/presentations/253-ARM{_}DevCon{_}2007{_}Snapdragon{_}FINAL{_}.>

MICROSYSTEMS, S. UltraSPARC T1™ Supplement. n. 8, 2007.

MITTAL, S. A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors. **ACM Computing Surveys**, v. 48, n. 3, p. 1–38, 2016. ISSN 03600300. Available from Internet: <<http://dl.acm.org/citation.cfm?doid=2856149.2856125>>.

MOGUL, J. C. et al. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. **IEEE Micro**, v. 28, n. 3, p. 26–41, may 2008. ISSN 0272-1732. Available from Internet: <<http://ieeexplore.ieee.org/document/4550858/>>.

MONCHIERO, M.; CANAL, R.; GONZALEZ, A. Power/Performance/Thermal Design-Space Exploration for Multicore Architectures. **IEEE Transactions on Parallel and Distributed Systems**, v. 19, n. 5, p. 666–681, may 2008. ISSN 1045-9219. Available from Internet: <<http://ieeexplore.ieee.org/document/4359440/>>.

NAVADA, S. et al. A Unified View of Non-monotonic Core Selection and Application Steering in Heterogeneous Chip Multiprocessors. In: **Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques**. Piscataway, NJ, USA: IEEE Press, 2013. (PACT '13), p. 133–144. ISBN 978-1-4799-1021-2. Available from Internet: <<http://dl.acm.org/citation.cfm?id=2523721.2523743>>.

PADMANABHA, S. et al. DynaMOS: Dynamic schedule migration for heterogeneous cores. In: **Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48**. New York, New York, USA: ACM Press, 2015. p. 322–333. ISBN 9781450340342. Available from Internet: <<http://dl.acm.org/citation.cfm?doid=2830772.2830791>>.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design, Fifth Edition: The Hardware/Software Interface**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 0124077269, 9780124077263.

POUCHET, L.-N. **PolyBench/C – The Polyhedral Benchmark suite**. 2019. Available from Internet: <<http://web.cse.ohio-state.edu/~pouchet.2/software/polybenc>>.

REDDY, D. et al. Bridging functional heterogeneity in multicore architectures. **ACM SIGOPS Operating Systems Review**, v. 45, n. 1, p. 21, 2011. ISSN 01635980.

SFREDDO, J. et al. A framework to automatically generate heterogeneous organization reconfigurable multiprocessing. In: **2017 IEEE International Symposium on Circuits and Systems (ISCAS)**. IEEE, 2017. p. 1–4. ISBN 978-1-4673-6853-7. Available from Internet: <<http://ieeexplore.ieee.org/document/8050438/>>.

SHIMPI, A. L. **The Bulldozer Review: AMD FX-8150 Tested**. 2011. Available from Internet: <<https://www.anandtech.com/show/4955/the-bulldozer-review-amd-fx8150-tested>>.

SILVEIRA, L. A. da et al. The Potential of Accelerating Image-Processing Applications by Using Approximate Function Reuse. In: IEEE. **Computing Systems Engineering (SBESC), 2016 VI Brazilian Symposium on**. [S.l.], 2016. p. 122–127.

SMITH, J.; SOHI, G. The microarchitecture of superscalar processors. **Proceedings of the IEEE**, v. 83, n. 12, p. 1609–1624, 1995. ISSN 00189219. Available from Internet: <<http://ieeexplore.ieee.org/document/476078/>>.

SONDAG, T.; RAJAN, H. Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors. In: **2009 ICSE Workshop on Multicore Software Engineering**. IEEE, 2009. p. 73–80. ISBN 978-1-4244-3718-4. Available from Internet: <<http://ieeexplore.ieee.org/document/5071386/>>.

SOUZA, J. et al. **Evaluating schedulers in a reconfigurable multicore heterogeneous system**. [S.l.: s.n.], 2016. 261–272 p. ISSN 16113349 03029743. ISBN 9783319304809.

SOUZA, J. D. et al. Towards a Dynamic and Reconfigurable Multicore Heterogeneous System. **2014 Brazilian Symposium on Computing Systems Engineering**, p. 73–78, 2014. Available from Internet: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7091169>>.

SOUZA, J. D. et al. A Reconfigurable Heterogeneous Multicore with a Homogeneous ISA. In: **Proceedings of the 2016 Conference on Design, Automation & Test in Europe**. San Jose, CA, USA: EDA Consortium, 2016. (DATE '16), p. 1598–1603. ISBN 978-3-9815370-6-2. Available from Internet: <<http://dl.acm.org/citation.cfm?id=2971808.2972181>>.

SOUZA, J. D.; FILHO, A. C. S. B. Trimming the ISA to Optimize Area and EDP in Heterogeneous CMPs. In: **2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. IEEE, 2019. p. 17–24. ISBN 978-1-7281-4194-7. Available from Internet: <<https://ieeexplore.ieee.org/document/8924189/>>.

SOUZA, J. D. et al. Enhancing Multithreaded Performance of Asymmetric Multicores with SIMD Offloading. In: **23rd Design, Automation and Test in Europe Conference**. [S.l.: s.n.], 2020.

SOUZA, J. D. et al. Enhancing Thread-Level Parallelism in Asymmetric Multicores using Transparent Instruction Offloading. In: **57th ACM/EDAC/IEEE Design Automation Conference (DAC'20)**. [S.l.: s.n.], 2020.

SOUZA, J. D. et al. DIM-VEX: Exploiting Design Time Configurability and Runtime Reconfigurability. 2018. Available from Internet: <[https://doi.org/10.1007/978-3-319-78890-6{_}.>](https://doi.org/10.1007/978-3-319-78890-6{_}.)

SRINIVASAN, S. et al. Exploring Heterogeneity within a Core for Improved Power Efficiency. **IEEE Transactions on Parallel and Distributed Systems**, v. 27, n. 4, p. 1057–1069, apr 2016. ISSN 1045-9219. Available from Internet: <<http://ieeexplore.ieee.org/document/7103357/>>.

SULEMAN, M. A. et al. Accelerating critical section execution with asymmetric multi-core architectures. In: **Proceeding of the 14th international conference on Architectural support for programming languages and operating systems - ASPLOS '09**. New York, New York, USA: ACM Press, 2009. v. 44, n. 3, p. 253. ISBN 9781605584065. ISSN 0362-1340. Available from Internet: <<http://portal.acm.org/citation.cfm?doid=1508244.1508274>>.

SUN. **OpenSPARC T1**. 2019. Available from Internet: <<https://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html>>.

TAN, C. et al. Locus: Low-power customizable many-core architecture for wearables. **ACM Trans. Embed. Comput. Syst.**, ACM, New York, NY, USA, v. 17, n. 1, nov. 2017.

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)**. IEEE, 2019. p. 42–55. ISBN 978-1-7281-1444-6. Available from Internet: <<https://ieeexplore.ieee.org/document/8675215/>>.

VENKAT, A.; TULLSEN, D. M. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. **Proceedings - International Symposium on Computer Architecture**, p. 121–132, 2014. ISSN 10636897.

WOO, S. C. et al. The SPLASH-2 programs. **ACM SIGARCH Computer Architecture News**, Association for Computing Machinery (ACM), v. 23, n. 2, p. 24–36, may 1995. ISSN 01635964.

Appendices

Appendix A

A.1 Processor configuration files from gem5

A.1.1 A15 core

```
1 # Copyright (c) 2012 The Regents of The University of Michigan
# Copyright (c) 2016 Centre National de la Recherche Scientifique
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
6 # modification, are permitted provided that the following conditions
    are
# met: redistributions of source code must retain the above copyright
# notice, this list of conditions and the following disclaimer;
# redistributions in binary form must reproduce the above copyright
# notice, this list of conditions and the following disclaimer in the
11 # documentation and/or other materials provided with the distribution;
# neither the name of the copyright holders nor the names of its
# contributors may be used to endorse or promote products derived from
# this software without specific prior written permission.
#
16 # THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
# OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
21 # SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
26 # OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
# Authors: Ron Dreslinski
#          Anastasiia Butko
#          Louisa Bessad
31
from m5.objects import *
```

```

#
-----

#           ex5 big core (based on the ARM Cortex-A15)
36 #
-----

# Simple ALU Instructions have a latency of 1
class ex5_big_Simple_Int (FUDESC):
    opList = [ OpDesc(opClass='IntALU', opLat=1) ]
41     count = 2

# Complex ALU instructions have a variable latencies
class ex5_big_Complex_Int (FUDESC):
    opList = [ OpDesc(opClass='IntMult', opLat=4, pipelined=True),
46             OpDesc(opClass='IntDiv', opLat=11, pipelined=False),
             OpDesc(opClass='IprAccess', opLat=3, pipelined=True) ]
    count = 1

# Floating point and SIMD instructions
51 class ex5_big_FP (FUDESC):
    opList = [ OpDesc(opClass='SimdAdd', opLat=3),
              OpDesc(opClass='SimdAddAcc', opLat=4),
              OpDesc(opClass='SimdAlu', opLat=4),
              OpDesc(opClass='SimdCmp', opLat=4),
56             OpDesc(opClass='SimdCvt', opLat=3),
              OpDesc(opClass='SimdMisc', opLat=3),
              OpDesc(opClass='SimdMult', opLat=6),
              OpDesc(opClass='SimdMultAcc', opLat=5),
              OpDesc(opClass='SimdShift', opLat=3),
61             OpDesc(opClass='SimdShiftAcc', opLat=3),
              OpDesc(opClass='SimdSqrt', opLat=9),
              OpDesc(opClass='SimdFloatAdd', opLat=6),
              OpDesc(opClass='SimdFloatAlu', opLat=5),
              OpDesc(opClass='SimdFloatCmp', opLat=3),
66             OpDesc(opClass='SimdFloatCvt', opLat=3),
              OpDesc(opClass='SimdFloatDiv', opLat=21),
              OpDesc(opClass='SimdFloatMisc', opLat=3),
              OpDesc(opClass='SimdFloatMult', opLat=6),
              OpDesc(opClass='SimdFloatMultAcc', opLat=1),

```

```

71         OpDesc (opClass=' SimdFloatSqrt' , opLat=9),
           OpDesc (opClass=' FloatAdd' , opLat=6),
           OpDesc (opClass=' FloatCmp' , opLat=5),
           OpDesc (opClass=' FloatCvt' , opLat=5),
           OpDesc (opClass=' FloatDiv' , opLat=12, pipelined=False),
76         OpDesc (opClass=' FloatSqrt' , opLat=33, pipelined=False),
           OpDesc (opClass=' FloatMult' , opLat=8) ]

    count = 2

81 # Load/Store Units
    class ex5_big_Load (FUDesc) :
        opList = [ OpDesc (opClass=' MemRead' , opLat=2) ]
        count = 1

86 class ex5_big_Store (FUDesc) :
        opList = [ OpDesc (opClass=' MemWrite' , opLat=2) ]
        count = 1

    # Functional Units for this CPU
91 class ex5_big_FUP (FUPool) :
        FUList = [ex5_big_Simple_Int (), ex5_big_Complex_Int (),
                  ex5_big_Load (), ex5_big_Store (), ex5_big_FP ()]

    # Bi-Mode Branch Predictor
96 class ex5_big_BP (BiModeBP) :
        globalPredictorSize = 4096
        globalCtrBits = 2
        choicePredictorSize = 1024
        choiceCtrBits = 3
101 BTBEntries = 4096
        BTBTagSize = 18
        RASSize = 48
        instShiftAmt = 2

106 class ex5_big (DerivO3CPU) :
        LQEntries = 16
        SQEntries = 16
        LSQDepCheckShift = 0
        LFSTSize = 1024
111 SSITSize = 1024

```

```
decodeToFetchDelay = 1
renameToFetchDelay = 1
iewToFetchDelay = 1
commitToFetchDelay = 1
116 renameToDecodeDelay = 1
    iewToDecodeDelay = 1
    commitToDecodeDelay = 1
    iewToRenameDelay = 1
    commitToRenameDelay = 1
121 commitToIEWDelay = 1
    fetchWidth = 3
    fetchBufferSize = 16
    fetchToDecodeDelay = 3
    decodeWidth = 3
126 decodeToRenameDelay = 2
    renameWidth = 3
    renameToIEWDelay = 1
    issueToExecuteDelay = 1
    dispatchWidth = 6
131 issueWidth = 8
    wbWidth = 8
    fuPool = ex5_big_FUP()
    iewToCommitDelay = 1
    renameToROBDelay = 1
136 commitWidth = 8
    squashWidth = 8
    trapLatency = 13
    backComSize = 5
    forwardComSize = 5
141 numPhysIntRegs = 90
    numPhysFloatRegs = 256
    numIQEntries = 48
    numROBEntries = 60

146 switched_out = False
    branchPred = ex5_big_BP()

class L1Cache(Cache):
    tag_latency = 2
151 data_latency = 2
    response_latency = 2
```

```
    tgts_per_mshr = 8
    # Consider the L2 a victim cache also for clean lines
    writeback_clean = True
156
# Instruction Cache
class L1I(L1Cache):
    mshrs = 2
    size = '32kB'
161    assoc = 2
    is_read_only = True

# Data Cache
class L1D(L1Cache):
166    mshrs = 6
    size = '32kB'
    assoc = 2
    write_buffers = 16

171 # TLB Cache
# Use a cache as a L2 TLB
class WalkCache(Cache):
    tag_latency = 4
    data_latency = 4
176    response_latency = 4
    mshrs = 6
    tgts_per_mshr = 8
    size = '1kB'
    assoc = 8
181    write_buffers = 16
    is_read_only = True
    # Writeback clean lines as well
    writeback_clean = True

186 # L2 Cache
class L2(Cache):
    tag_latency = 15
    data_latency = 15
    response_latency = 15
191    mshrs = 16
    tgts_per_mshr = 8
    size = '2MB'
```

```

assoc = 16
write_buffers = 8
196 prefetch_on_access = True
clusivity = 'mostly_excl'
# Simple stride prefetcher
prefetcher = StridePrefetcher(degree=8, latency = 1)
tags = RandomRepl()

```

A.1.2 A7 core

```

# Copyright (c) 2012 The Regents of The University of Michigan
# Copyright (c) 2016 Centre National de la Recherche Scientifique
# All rights reserved.
#
5 # Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are
# met: redistributions of source code must retain the above copyright
# notice, this list of conditions and the following disclaimer;
# redistributions in binary form must reproduce the above copyright
10 # notice, this list of conditions and the following disclaimer in the
# documentation and/or other materials provided with the distribution;
# neither the name of the copyright holders nor the names of its
# contributors may be used to endorse or promote products derived from
# this software without specific prior written permission.
15 #
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
20 # OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
25 # (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#

```

```

# Authors: Ron Dreslinski
#           Anastasiia Butko
30 #           Louisa Bessad

from m5.objects import *

#
-----

35 #           ex5 LITTLE core (based on the ARM Cortex-A7)
#
-----

# Simple ALU Instructions have a latency of 3
class ex5_LITTLE_Simple_Int (MinorDefaultIntFU) :
40     opList = [ OpDesc (opClass='IntALU', opLat=4) ]

# Complex ALU instructions have a variable latencies
class ex5_LITTLE_Complex_IntMul (MinorDefaultIntMulFU) :
     opList = [ OpDesc (opClass='IntMult', opLat=7) ]
45
class ex5_LITTLE_Complex_IntDiv (MinorDefaultIntDivFU) :
     opList = [ OpDesc (opClass='IntDiv', opLat=9) ]

# Floating point and SIMD instructions
50 class ex5_LITTLE_FP (MinorDefaultFloatSimdFU) :
     opList = [ OpDesc (opClass='SimdAdd', opLat=6),
                OpDesc (opClass='SimdAddAcc', opLat=4),
                OpDesc (opClass='SimdAlu', opLat=4),
                OpDesc (opClass='SimdCmp', opLat=1),
55     OpDesc (opClass='SimdCvt', opLat=3),
                OpDesc (opClass='SimdMisc', opLat=3),
                OpDesc (opClass='SimdMult', opLat=4),
                OpDesc (opClass='SimdMultAcc', opLat=5),
                OpDesc (opClass='SimdShift', opLat=3),
60     OpDesc (opClass='SimdShiftAcc', opLat=3),
                OpDesc (opClass='SimdSqrt', opLat=9),
                OpDesc (opClass='SimdFloatAdd', opLat=8),
                OpDesc (opClass='SimdFloatAlu', opLat=6),
                OpDesc (opClass='SimdFloatCmp', opLat=6),

```

```

65         OpDesc (opClass=' SimdFloatCvt' , opLat=6),
           OpDesc (opClass=' SimdFloatDiv' , opLat=20, pipelined=False
                   ),
           OpDesc (opClass=' SimdFloatMisc' , opLat=6),
           OpDesc (opClass=' SimdFloatMult' , opLat=15),
           OpDesc (opClass=' SimdFloatMultAcc' , opLat=6),
70         OpDesc (opClass=' SimdFloatSqrt' , opLat=17),
           OpDesc (opClass=' FloatAdd' , opLat=8),
           OpDesc (opClass=' FloatCmp' , opLat=6),
           OpDesc (opClass=' FloatCvt' , opLat=6),
           OpDesc (opClass=' FloatDiv' , opLat=15, pipelined=False),
75         OpDesc (opClass=' FloatSqrt' , opLat=33),
           OpDesc (opClass=' FloatMult' , opLat=6) ]

# Load/Store Units
class ex5_LITTLE_MemFU (MinorDefaultMemFU) :
80     opList = [ OpDesc (opClass=' MemRead' , opLat=1),
                 OpDesc (opClass=' MemWrite' , opLat=1) ]

# Misc Unit
class ex5_LITTLE_MiscFU (MinorDefaultMiscFU) :
85     opList = [ OpDesc (opClass=' IprAccess' , opLat=1),
                 OpDesc (opClass=' InstPrefetch' , opLat=1) ]

# Functional Units for this CPU
class ex5_LITTLE_FUP (MinorFUPool) :
90     funcUnits = [ex5_LITTLE_Simple_Int (), ex5_LITTLE_Simple_Int (),
                  ex5_LITTLE_Complex_IntMul (), ex5_LITTLE_Complex_IntDiv (),
                  ex5_LITTLE_FP (), ex5_LITTLE_MemFU (),
                  ex5_LITTLE_MiscFU () ]

95 class ex5_LITTLE (MinorCPU) :
    executeFuncUnits = ex5_LITTLE_FUP ()

class L1Cache (Cache) :
    tag_latency = 2
100    data_latency = 2
        response_latency = 2
        tgts_per_mshr = 8
        # Consider the L2 a victim cache also for clean lines
        writeback_clean = True

```

```
105
class L1I(L1Cache):
    mshrs = 2
    size = '32kB'
    assoc = 2
110    is_read_only = True
    tgts_per_mshr = 20

class L1D(L1Cache):
    mshrs = 4
115    size = '32kB'
    assoc = 4
    write_buffers = 4

# TLB Cache
120 # Use a cache as a L2 TLB
class WalkCache(Cache):
    tag_latency = 2
    data_latency = 2
    response_latency = 2
125    mshrs = 6
    tgts_per_mshr = 8
    size = '1kB'
    assoc = 2
    write_buffers = 16
130    is_read_only = True
    # Writeback clean lines as well
    writeback_clean = True

# L2 Cache
135 #class L2(Cache):
#    tag_latency = 9
#    data_latency = 9
#    response_latency = 9
#    mshrs = 8
140 #    tgts_per_mshr = 12
#    size = '512kB'
#    assoc = 8
#    write_buffers = 16
#    prefetch_on_access = True
145 #    clusivity = 'mostly_excl'
```

```

# # Simple stride prefetcher
# prefetcher = StridePrefetcher(degree=1, latency = 1)
# tags = RandomRepl()

150 # L2 Cache big core
class L2(Cache):
    tag_latency = 15
    data_latency = 15
    response_latency = 15
155 mshrs = 16
    tgts_per_mshr = 8
    size = '2MB'
    assoc = 16
    write_buffers = 8
160 prefetch_on_access = True
    clusivity = 'mostly_excl'
    # Simple stride prefetcher
    prefetcher = StridePrefetcher(degree=1, latency = 1)
    tags = RandomRepl()
165 #repl_policy = RandomRP()

```

A.1.3 A15 core with extra SIMD/FP latency (for TUNE)

```

# Copyright (c) 2012 The Regents of The University of Michigan
# Copyright (c) 2016 Centre National de la Recherche Scientifique
# All rights reserved.
#
5 # Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are
# met: redistributions of source code must retain the above copyright
# notice, this list of conditions and the following disclaimer;
# redistributions in binary form must reproduce the above copyright
10 # notice, this list of conditions and the following disclaimer in the
# documentation and/or other materials provided with the distribution;
# neither the name of the copyright holders nor the names of its
# contributors may be used to endorse or promote products derived from
# this software without specific prior written permission.

```

```

15 #
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
20 # OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
25 # (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
# Authors: Ron Dreslinski
#          Anastasiia Butko
30 #          Louisa Bessad

from m5.objects import *

#
-----
35 #          ex5 big core (based on the ARM Cortex-A15)
#
-----

# Simple ALU Instructions have a latency of 1
class ex5_big_Simple_Int (FUDesc) :
40     opList = [ OpDesc(opClass='IntALU', opLat=1) ]
     count = 2

# Complex ALU instructions have a variable latencies
class ex5_big_Complex_Int (FUDesc) :
45     opList = [ OpDesc(opClass='IntMult', opLat=4, pipelined=True),
                OpDesc(opClass='IntDiv', opLat=11, pipelined=False),
                OpDesc(opClass='IprAccess', opLat=3, pipelined=True) ]
     count = 1

50 # Floating point and SIMD instructions
class ex5_big_FP (FUDesc) :

```

```

opList = [ OpDesc(opClass=' SimdAdd' , opLat=3),
           OpDesc(opClass=' SimdAddAcc' , opLat=4),
           OpDesc(opClass=' SimdAlu' , opLat=4),
55         OpDesc(opClass=' SimdCmp' , opLat=4),
           OpDesc(opClass=' SimdCvt' , opLat=3),
           OpDesc(opClass=' SimdMisc' , opLat=3),
           OpDesc(opClass=' SimdMult' , opLat=6),
           OpDesc(opClass=' SimdMultAcc' , opLat=5),
60         OpDesc(opClass=' SimdShift' , opLat=3),
           OpDesc(opClass=' SimdShiftAcc' , opLat=3),
           OpDesc(opClass=' SimdSqrt' , opLat=9),
           OpDesc(opClass=' SimdFloatAdd' , opLat=6),
           OpDesc(opClass=' SimdFloatAlu' , opLat=5),
65         OpDesc(opClass=' SimdFloatCmp' , opLat=3),
           OpDesc(opClass=' SimdFloatCvt' , opLat=3),
           OpDesc(opClass=' SimdFloatDiv' , opLat=21),
           OpDesc(opClass=' SimdFloatMisc' , opLat=3),
           OpDesc(opClass=' SimdFloatMult' , opLat=6),
70         OpDesc(opClass=' SimdFloatMultAcc' , opLat=1),
           OpDesc(opClass=' SimdFloatSqrt' , opLat=9),
           OpDesc(opClass=' FloatAdd' , opLat=6),
           OpDesc(opClass=' FloatCmp' , opLat=5),
           OpDesc(opClass=' FloatCvt' , opLat=5),
75         OpDesc(opClass=' FloatDiv' , opLat=12, pipelined=False),
           OpDesc(opClass=' FloatSqrt' , opLat=33, pipelined=False),
           OpDesc(opClass=' FloatMult' , opLat=8) ]

timings = [MinorFUTiming(description=' FloatSimd' ,
                          extraCommitLat=10, srcRegsRelativeLats=[2])]
80 count = 2

# Load/Store Units
class ex5_big_Load(FUDesc):
85     opList = [ OpDesc(opClass=' MemRead' , opLat=2) ]
    count = 1

class ex5_big_Store(FUDesc):
    opList = [OpDesc(opClass=' MemWrite' , opLat=2) ]
90     count = 1

# Functional Units for this CPU

```

```

class ex5_big_FUP(FUPool):
    FUList = [ex5_big_Simple_Int(), ex5_big_Complex_Int(),
95             ex5_big_Load(), ex5_big_Store(), ex5_big_FP()]

# Bi-Mode Branch Predictor
class ex5_big_BP(BiModeBP):
    globalPredictorSize = 4096
    globalCtrBits = 2
100    choicePredictorSize = 1024
    choiceCtrBits = 3
    BTBEntries = 4096
    BTBTagSize = 18
105    RASSize = 48
    instShiftAmt = 2

class ex5_big(DerivO3CPU):
    LQEntries = 16
110    SQEntries = 16
    LSQDepCheckShift = 0
    LFSTSize = 1024
    SSITSize = 1024
    decodeToFetchDelay = 1
115    renameToFetchDelay = 1
    iewToFetchDelay = 1
    commitToFetchDelay = 1
    renameToDecodeDelay = 1
    iewToDecodeDelay = 1
120    commitToDecodeDelay = 1
    iewToRenameDelay = 1
    commitToRenameDelay = 1
    commitToIEWDelay = 1
    fetchWidth = 3
125    fetchBufferSize = 16
    fetchToDecodeDelay = 3
    decodeWidth = 3
    decodeToRenameDelay = 2
    renameWidth = 3
130    renameToIEWDelay = 1
    issueToExecuteDelay = 1
    dispatchWidth = 6
    issueWidth = 8

```

```

wbWidth = 8
135 fuPool = ex5_big_FUP()
    iewToCommitDelay = 1
    renameToROBDelay = 1
    commitWidth = 8
    squashWidth = 8
140 trapLatency = 13
    backComSize = 5
    forwardComSize = 5
    numPhysIntRegs = 90
    numPhysFloatRegs = 256
145 numIQEntries = 48
    numROBEntries = 60

    switched_out = False
    branchPred = ex5_big_BP()
150
class L1Cache(Cache):
    tag_latency = 2
    data_latency = 2
    response_latency = 2
155 tgts_per_mshr = 8
    # Consider the L2 a victim cache also for clean lines
    writeback_clean = True

# Instruction Cache
160 class L1I(L1Cache):
    mshrs = 2
    size = '32kB'
    assoc = 2
    is_read_only = True

165
# Data Cache
class L1D(L1Cache):
    mshrs = 6
    size = '32kB'
170 assoc = 2
    write_buffers = 16

# TLB Cache
# Use a cache as a L2 TLB

```

```

175 class WalkCache(Cache):
    tag_latency = 4
    data_latency = 4
    response_latency = 4
    mshrs = 6
180 tgts_per_mshr = 8
    size = '1kB'
    assoc = 8
    write_buffers = 16
    is_read_only = True
185 # Writeback clean lines as well
    writeback_clean = True

# L2 Cache
class L2(Cache):
190 tag_latency = 15
    data_latency = 15
    response_latency = 15
    mshrs = 16
    tgts_per_mshr = 8
195 size = '2MB'
    assoc = 16
    write_buffers = 8
    prefetch_on_access = True
    clusivity = 'mostly_excl'
200 # Simple stride prefetcher
    prefetcher = StridePrefetcher(degree=8, latency = 1)
    tags = RandomRepl()

```

A.1.4 A7 core with extra SIMD/FP latency (for Niagara)

```

# Copyright (c) 2012 The Regents of The University of Michigan
2 # Copyright (c) 2016 Centre National de la Recherche Scientifique
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
    are

```

```

7 # met: redistributions of source code must retain the above copyright
# notice, this list of conditions and the following disclaimer;
# redistributions in binary form must reproduce the above copyright
# notice, this list of conditions and the following disclaimer in the
# documentation and/or other materials provided with the distribution;
12 # neither the name of the copyright holders nor the names of its
# contributors may be used to endorse or promote products derived from
# this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
17 # "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
# OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
22 # LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
27 #
# Authors: Ron Dreslinski
#          Anastasiia Butko
#          Louisa Bessad
32 from m5.objects import *
#
#          -----
#          ex5 LITTLE core (based on the ARM Cortex-A7)
#          -----
37
# Simple ALU Instructions have a latency of 3
class ex5_LITTLE_Simple_Int (MinorDefaultIntFU):
    opList = [ OpDesc(opClass='IntALU', opLat=4) ]
42 # Complex ALU instructions have a variable latencies
class ex5_LITTLE_Complex_IntMul (MinorDefaultIntMulFU):

```

```

opList = [ OpDesc(opClass='IntMult', opLat=7) ]

class ex5_LITTLE_Complex_IntDiv(MinorDefaultIntDivFU):
47  opList = [ OpDesc(opClass='IntDiv', opLat=9) ]

# Floating point and SIMD instructions
class ex5_LITTLE_FP(MinorDefaultFloatSimdFU):
    opList = [ OpDesc(opClass='SimdAdd', opLat=6),
52             OpDesc(opClass='SimdAddAcc', opLat=4),
             OpDesc(opClass='SimdAlu', opLat=4),
             OpDesc(opClass='SimdCmp', opLat=1),
             OpDesc(opClass='SimdCvt', opLat=3),
             OpDesc(opClass='SimdMisc', opLat=3),
57             OpDesc(opClass='SimdMult', opLat=4),
             OpDesc(opClass='SimdMultAcc', opLat=5),
             OpDesc(opClass='SimdShift', opLat=3),
             OpDesc(opClass='SimdShiftAcc', opLat=3),
             OpDesc(opClass='SimdSqrt', opLat=9),
62             OpDesc(opClass='SimdFloatAdd', opLat=8),
             OpDesc(opClass='SimdFloatAlu', opLat=6),
             OpDesc(opClass='SimdFloatCmp', opLat=6),
             OpDesc(opClass='SimdFloatCvt', opLat=6),
             OpDesc(opClass='SimdFloatDiv', opLat=20, pipelined=False
67             ),
             OpDesc(opClass='SimdFloatMisc', opLat=6),
             OpDesc(opClass='SimdFloatMult', opLat=15),
             OpDesc(opClass='SimdFloatMultAcc', opLat=6),
             OpDesc(opClass='SimdFloatSqrt', opLat=17),
             OpDesc(opClass='FloatAdd', opLat=8),
72             OpDesc(opClass='FloatCmp', opLat=6),
             OpDesc(opClass='FloatCvt', opLat=6),
             OpDesc(opClass='FloatDiv', opLat=15, pipelined=False),
             OpDesc(opClass='FloatSqrt', opLat=33),
             OpDesc(opClass='FloatMult', opLat=6) ]
77  timings = [MinorFUTiming(description='FloatSimd',
                             extraCommitLat=20, srcRegsRelativeLats=[2])]

# Load/Store Units
class ex5_LITTLE_MemFU(MinorDefaultMemFU):
82  opList = [ OpDesc(opClass='MemRead', opLat=1),
             OpDesc(opClass='MemWrite', opLat=1) ]

```

```

# Misc Unit
class ex5_LITTLE_MiscFU(MinorDefaultMiscFU):
87     opList = [ OpDesc(opClass='IprAccess', opLat=1),
                OpDesc(opClass='InstPrefetch', opLat=1) ]

# Functional Units for this CPU
class ex5_LITTLE_FUP(MinorFUPool):
92     funcUnits = [ex5_LITTLE_Simple_Int(), ex5_LITTLE_Simple_Int(),
                  ex5_LITTLE_Complex_IntMul(), ex5_LITTLE_Complex_IntDiv(),
                  ex5_LITTLE_FP(), ex5_LITTLE_MemFU(),
                  ex5_LITTLE_MiscFU()]

97 class ex5_LITTLE_BUFFERED(MinorCPU):
    executeFuncUnits = ex5_LITTLE_FUP()

class L1Cache(Cache):
    tag_latency = 2
102    data_latency = 2
    response_latency = 2
    tgts_per_mshr = 8
    # Consider the L2 a victim cache also for clean lines
    writeback_clean = True

107 class L1I(L1Cache):
    mshrs = 2
    size = '32kB'
    assoc = 2
112    is_read_only = True
    tgts_per_mshr = 20

class L1D(L1Cache):
    mshrs = 4
117    size = '32kB'
    assoc = 4
    write_buffers = 4

# TLB Cache
122 # Use a cache as a L2 TLB
class WalkCache(Cache):
    tag_latency = 2

```

```
data_latency = 2
response_latency = 2
127 mshrs = 6
    tgts_per_mshr = 8
    size = '1kB'
    assoc = 2
    write_buffers = 16
132 is_read_only = True
    # Writeback clean lines as well
    writeback_clean = True

# L2 Cache
137 class L2(Cache):
    tag_latency = 9
    data_latency = 9
    response_latency = 9
    mshrs = 8
142 tgts_per_mshr = 12
    size = '512kB'
    assoc = 8
    write_buffers = 16
    prefetch_on_access = True
147 clusivity = 'mostly_excl'
    # Simple stride prefetcher
    prefetcher = StridePrefetcher(degree=1, latency = 1)
    tags = RandomRepl()
```

Appendix B

B.1 PHISA Multicores raw results values

Table B.1: Setup 1

		Cycles	Energy	EDP
Scenario 1	A15(4F0P)	132403130	228411710.1	3.02424E+16
	A15(3F1P)	129398986	232916623	3.01392E+16
	A15(2F2P)	165050854	231135711.2	3.81491E+16
	A15(1F3P)	194616321	229742773.1	4.47117E+16
Scenario 2	A15(4F0P)	129320412	153334138.4	1.98292E+16
	A15(3F1P)	129320916	158853153.5	2.0543E+16
	A15(2F2P)	164778015	153306314.9	2.52615E+16
	A15(1F3P)	241607457	180868375.8	4.36991E+16
Scenario 3	A15(4F0P)	129320412	155685349.1	2.01333E+16
	A15(3F1P)	129320152	155654513.1	2.01293E+16
	A15(2F2P)	171050052	155619207.8	2.66187E+16
	A15(1F3P)	243658974	180464051.4	4.39717E+16
Scenario 4	A15(4F0P)	129320152	170347914.1	2.20294E+16
	A15(3F1P)	129320412	174329318.1	2.25443E+16
	A15(2F2P)	135897330	195183071	2.65249E+16
	A15(1F3P)	129320411	171198590.8	2.21395E+16
Scenario 5	A15(4F0P)	129259607	119214415.5	1.54096E+16
	A15(3F1P)	129438513	105820894.8	1.36973E+16
	A15(2F2P)	129438513	105807568	1.36956E+16
	A15(1F3P)	136587841	105892254.1	1.44636E+16
Scenario 6	A15(4F0P)	60233946	83814001.45	5.04845E+15
	A15(3F1P)	67386822	88680156.07	5.97587E+15
	A15(2F2P)	75752735	87187148.6	6.60466E+15
	A15(1F3P)	118433300	89277017.71	1.05734E+16

Table B.2: Setup 2

		Cycles	Energy	EDP
Scenario 1	A15(1F0P)	588703904	406498279.6	2.39307E+17
	A15(0F1P) A7(2F0P)	303050639	130446422.2	3.95319E+16
	A15(0F1P) A7(1F1P)	555568230	70797202.72	3.93327E+16
Scenario 2	A15(1F0P)	359523965	248250219.3	8.92519E+16
	A15(0F1P) A7(2F0P)	175264300	97251648.64	1.70447E+16
	A15(0F1P) A7(1F1P)	343728371	96028393.28	3.30077E+16
Scenario 3	A15(1F0P)	387663281	267680332.5	1.0377E+17
	A15(0F1P) A7(2F0P)	194318684	98341070.65	1.91095E+16
	A15(0F1P) A7(1F1P)	377419153	97311312.2	3.67272E+16
Scenario 4	A15(1F0P)	336271085	232194175.4	7.80802E+16
	A15(0F1P) A7(2F0P)	180024569	91697885.4	1.65079E+16
	A15(0F1P) A7(1F1P)	278596562	93509554.29	2.60514E+16
Scenario 5	A15(1F0P)	261329038	180446916.8	4.7156E+16
	A15(0F1P) A7(2F0P)	180376029	86397254.96	1.5584E+16
	A15(0F1P) A7(1F1P)	200060633	87654140.12	1.75361E+16
Scenario 6	A15(1F0P)	266420882	183962819.8	4.90115E+16
	A15(0F1P) A7(2F0P)	223772236	35791482.33	8.00914E+15
	A15(0F1P) A7(1F1P)	379894324	28232956.96	1.07255E+16

Table B.3: Setup 3

		Cycles	Energy	EDP
Scenario 1	A15(1F0P) A7(2F0P)	236865977	183618620.9	4.3493E+16
	A15(0F1P) A7(2F0P)	303050639	130446422.2	3.95319E+16
	A15(0F1P) A7(1F1P)	555568230	70797202.72	3.93327E+16
Scenario 2	A15(1F0P) A7(2F0P)	192086097	110105107.3	2.11497E+16
	A15(0F1P) A7(2F0P)	175264300	97251648.64	1.70447E+16
	A15(0F1P) A7(1F1P)	343728371	96028393.28	3.30077E+16
Scenario 3	A15(1F0P) A7(2F0P)	184747330	134636906.6	2.48738E+16
	A15(0F1P) A7(2F0P)	194318684	98341070.65	1.91095E+16
	A15(0F1P) A7(1F1P)	377419153	97311312.2	3.67272E+16
Scenario 4	A15(1F0P) A7(2F0P)	190762441	106589860	2.03333E+16
	A15(0F1P) A7(2F0P)	180024569	91697885.4	1.65079E+16
	A15(0F1P) A7(1F1P)	278596562	93509554.29	2.60514E+16
Scenario 5	A15(1F0P) A7(2F0P)	144463496	100359609.2	1.44983E+16
	A15(0F1P) A7(2F0P)	180376029	86397254.96	1.5584E+16
	A15(0F1P) A7(1F1P)	200060633	87654140.12	1.75361E+16
Scenario 6	A15(1F0P) A7(2F0P)	142633018	109207844.1	1.55766E+16
	A15(0F1P) A7(2F0P)	223772236	35791482.33	8.00914E+15
	A15(0F1P) A7(1F1P)	379894324	28232956.96	1.07255E+16

Table B.4: Setup 4

		Cycles	Energy	EDP
Scenario 1	A15(1F0P) A7(2F0P)	236865977	183618620.9	4.3493E+16
	A15(0F1E) A7(2F0P)	271012060	136850767.6	3.70882E+16
	A15(0F1E) A7(1F1E)	332966322	109374882.9	3.64182E+16
Scenario 2	A15(1F0P) A7(2F0P)	192086097	110105107.3	2.11497E+16
	A15(0F1E) A7(2F0P)	160879354	105671406.6	1.70003E+16
	A15(0F1E) A7(1F1E)	230945570	98128058.41	2.26622E+16
Scenario 3	A15(1F0P) A7(2F0P)	184747330	134636906.6	2.48738E+16
	A15(0F1E) A7(2F0P)	191107188	107638211.2	2.05704E+16
	A15(0F1E) A7(1F1E)	232030445	103327719.7	2.39752E+16
Scenario 4	A15(1F0P) A7(2F0P)	190762441	106589860	2.03333E+16
	A15(0F1E) A7(2F0P)	178351640	93356221.17	1.66502E+16
	A15(0F1E) A7(1F1E)	193350780	95141159	1.83956E+16
Scenario 5	A15(1F0P) A7(2F0P)	144463496	100359609.2	1.44983E+16
	A15(0F1E) A7(2F0P)	130201717	87051862.84	1.13343E+16
	A15(0F1E) A7(1F1E)	168158964	48509813.35	8.15736E+15
Scenario 6	A15(1F0P) A7(2F0P)	142633018	109207844.1	1.55766E+16
	A15(0F1E) A7(2F0P)	182503509	79756290.18	1.45558E+16
	A15(0F1E) A7(1F1E)	217668864	77873942.84	1.69507E+16

Table B.5: Setup 5

		Cycles	Energy	EDP
Scenario 1	A15(1F0P) A7(2F0P)	236865977	183618620.9	4.3493E+16
	A15(0F1P) A7(4F0P)	163221669	107853436.1	1.7604E+16
	A15(0F1E) A7(2F2E)	212356040	109735279.9	2.33029E+16
Scenario 2	A15(1F0P) A7(2F0P)	192086097	110105107.3	2.11497E+16
	A15(0F1P) A7(4F0P)	153831474	93956368.95	1.44534E+16
	A15(0F1E) A7(2F2E)	153804646	96867511.41	1.48987E+16
Scenario 3	A15(1F0P) A7(2F0P)	184747330	134636906.6	2.48738E+16
	A15(0F1P) A7(4F0P)	129906779	95336677.03	1.23849E+16
	A15(0F1E) A7(2F2E)	157831878	38297295.36	6.04453E+15
Scenario 4	A15(1F0P) A7(2F0P)	190762441	106589860	2.03333E+16
	A15(0F1P) A7(4F0P)	129881605	91021591.94	1.1822E+16
	A15(0F1E) A7(2F2E)	129797737	94033651.82	1.22054E+16
Scenario 5	A15(1F0P) A7(2F0P)	144463496	100359609.2	1.44983E+16
	A15(0F1P) A7(4F0P)	129980829	86192117.78	1.12033E+16
	A15(0F1E) A7(2F2E)	105660883	17681726.58	1.86827E+15
Scenario 6	A15(1F0P) A7(2F0P)	142633018	109207844.1	1.55766E+16
	A15(0F1P) A7(4F0P)	138777334	33822276.5	4.69377E+15
	A15(0F1E) A7(2F2E)	163317574	58154467.62	9.49765E+15

Table B.6: Performance Policy - Single A15

		Cycles	Energy	EDP
Scenario 1	A15_1F0P	13177354359	8514224604	1.12195E+20
	A15_0F1P_A7_2F0P	6007625560	3390851060	2.0371E+19
	A15_0F1P_A7_1F1P	5986478472	3349345798	2.00508E+19
Scenario 2	A15_1F0P	6924669448	4454499341	3.08459E+19
	A15_0F1P_A7_2F0P	3251247362	1490785935	4.84691E+18
	A15_0F1P_A7_1F1P	3876275528	1471918929	5.70556E+18
Scenario 3	A15_1F0P	16525345054	10660315998	1.76165E+20
	A15_0F1P_A7_2F0P	7506222463	4389370940	3.29476E+19
	A15_0F1P_A7_1F1P	7525206119	4360639694	3.28147E+19
Scenario 4	A15_1F0P	13990996018	9004252136	1.25978E+20
	A15_0F1P_A7_2F0P	6043463415	3413880345	2.06317E+19
	A15_0F1P_A7_1F1P	6045428254	3386277049	2.04715E+19
Scenario 5	A15_1F0P	11641130291	7484630004	8.71296E+19
	A15_0F1P_A7_2F0P	5175624520	2849736967	1.47492E+19
	A15_0F1P_A7_1F1P	5179434615	2826356146	1.46389E+19
Scenario 6	A15_1F0P	1238808634	797731621.9	9.88237E+17
	A15_0F1P_A7_2F0P	1525867958	156510738	2.38815E+17
	A15_0F1P_A7_1F1P	1531257208	157272536.4	2.40825E+17

Table B.7: Performance Policy - big.LITTLE

		Cycles	Energy	EDP
Scenario 1	A15_1F0P_A7_2F0P	5747917148	4208095612	2.41878E+19
	A15_0F1P_A7_4F0P	4091508039	2469127653	1.01025E+19
	A15_0F1P_A7_2F2P	4092869321	2435882602	9.96975E+18
Scenario 2	A15_1F0P_A7_2F0P	2898605187	2102978947	6.09571E+18
	A15_0F1P_A7_4F0P	2453669344	1102096931	2.70418E+18
	A15_0F1P_A7_2F2P	2461573996	1081719262	2.66273E+18
Scenario 3	A15_1F0P_A7_2F0P	7278351340	5346748909	3.89155E+19
	A15_0F1P_A7_4F0P	5101111319	3254198174	1.66E+19
	A15_0F1P_A7_2F2P	5105560860	3208180094	1.63796E+19
Scenario 4	A15_1F0P_A7_2F0P	5810715154	4238043220	2.46261E+19
	A15_0F1P_A7_4F0P	4086903406	2466706520	1.00812E+19
	A15_0F1P_A7_2F2P	4088289510	2431598278	9.94108E+18
Scenario 5	A15_1F0P_A7_2F0P	4845718233	3521400047	1.70637E+19
	A15_0F1P_A7_4F0P	3524093515	2044406888	7.20468E+18
	A15_0F1P_A7_2F2P	3527913783	2015572252	7.11077E+18
Scenario 6	A15_1F0P_A7_2F0P	1118059732	727589988.5	8.13489E+17
	A15_0F1P_A7_4F0P	1514098814	154082253.3	2.33296E+17
	A15_0F1P_A7_2F2P	1519983033	153519867.2	2.33348E+17

Table B.8: Energy Policy - Single A15

		Cycles	Energy	EDP
Scenario 1	A15(1F0P)	13177354359	8514224604	1.12195E+20
	A15(0F1P) A7(2F0P)	6007300820	3345325735	2.00964E+19
	A15(0F1P) A7(1F1P)	6043015207	3303619701	1.99638E+19
Scenario 2	A15(1F0P)	6924669448	4454499341	3.08459E+19
	A15(0F1P) A7(2F0P)	3238406199	1415874511	4.58518E+18
	A15(0F1P) A7(1F1P)	3893742580	1377241962	5.36263E+18
Scenario 3	A15(1F0P)	16525345054	10660315998	1.76165E+20
	A15(0F1P) A7(2F0P)	7528199329	4359621305	3.28201E+19
	A15(0F1P) A7(1F1P)	7583891955	4317211857	3.27413E+19
Scenario 4	A15(1F0P)	13990996018	9004252136	1.25978E+20
	A15(0F1P) A7(2F0P)	6042126529	3367523271	2.0347E+19
	A15(0F1P) A7(1F1P)	6102805033	3341149851	2.03904E+19
Scenario 5	A15(1F0P)	11641130291	7484630004	8.71296E+19
	A15(0F1P) A7(2F0P)	5173721630	2802200532	1.44978E+19
	A15(0F1P) A7(1F1P)	5236319529	2779694015	1.45554E+19
Scenario 6	A15(1F0P)	1238808634	797731621.9	9.88237E+17
	A15(0F1P) A7(2F0P)	1470766463	101124842.7	1.48731E+17
	A15(0F1P) A7(1F1P)	1580867976	99594957.43	1.57446E+17

Table B.9: Energy Policy - big.LITTLE

		Cycles	Energy	EDP
Scenario 1	A15_1F0P_A7_2F0P	5972597162	3836175259	2.29119E+19
	A15_0F1P_A7_4F0P	4094381053	2425775690	9.93205E+18
	A15_0F1P_A7_2F2P	4148396094	2389713794	9.91348E+18
Scenario 2	A15_1F0P_A7_2F0P	3199721514	1591768870	5.09322E+18
	A15_0F1P_A7_4F0P	2442099725	1027610947	2.50953E+18
	A15_0F1P_A7_2F2P	2506962538	1007184437	2.52497E+18
Scenario 3	A15_1F0P_A7_2F0P	7483573288	4982718550	3.72885E+19
	A15_0F1P_A7_4F0P	5102810596	3211485967	1.63876E+19
	A15_0F1P_A7_2F2P	5158603861	3163329344	1.63184E+19
Scenario 4	A15_1F0P_A7_2F0P	6034435265	3866209661	2.33304E+19
	A15_0F1P_A7_4F0P	4087231585	2421090767	9.89556E+18
	A15_0F1P_A7_2F2P	4144104950	2386185555	9.8886E+18
Scenario 5	A15_1F0P_A7_2F0P	5072463496	3145328307	1.59546E+19
	A15_0F1P_A7_4F0P	3524821369	1998384765	7.04395E+18
	A15_0F1P_A7_2F2P	3582725308	1967466158	7.04889E+18
Scenario 6	A15_1F0P_A7_2F0P	1459278035	108952978.5	1.58993E+17
	A15_0F1P_A7_4F0P	1458512482	82797203.31	1.20761E+17
	A15_0F1P_A7_2F2P	1570625813	82276376.95	1.29225E+17

Table B.10: PHISA vs SOA - Performance Policy

		Cycles	Energy	EDP
Scenario 1	big.LITLLE	5747917148	4208095612	2.41878E+19
	SOA (Lee2017)	6934683157	7900232951	5.47856E+19
	PHISA	3221960451	2055324015	6.62217E+18
Scenario 2	big.LITLLE	2898605187	2102978947	6.09571E+18
	SOA (Lee2017)	4100960526	4179633420	1.71405E+19
	PHISA	2103543518	930027531.4	1.95635E+18
Scenario 3	big.LITLLE	7278351340	5346748909	3.89155E+19
	SOA (Lee2017)	8607884454	9883725757	8.5078E+19
	PHISA	3971745240	2720653973	1.08057E+19
Scenario 4	big.LITLLE	5810715154	4238043220	2.46261E+19
	SOA (Lee2017)	7338408284	8350770323	6.12814E+19
	PHISA	3205967320	2041150697	6.54386E+18
Scenario 5	big.LITLLE	4845718233	3521400047	1.70637E+19
	SOA (Lee2017)	6165633346	6945246381	4.28218E+19
	PHISA	2790100675	1686068936	4.7043E+18
Scenario 6	big.LITLLE	1118059732	727589988.5	8.13489E+17
	SOA (Lee2017)	1167770977	777473120.1	9.07911E+17
	PHISA	1512459457	153105390.9	2.31566E+17

Table B.11: PHISA vs SOA - Energy Policy

		Cycles	Energy	EDP
Scenario 1	big.LITLLE	5972597162	3836175259	2.29119E+19
	SOA (Lee2017)	6934683157	7900232951	5.47856E+19
	PHISA	3221647605	2009276098	6.47318E+18
Scenario 2	big.LITLLE	3199721514	1591768870	5.09322E+18
	SOA (Lee2017)	4100960526	4179633420	1.71405E+19
	PHISA	2091726782	856704568.7	1.79199E+18
Scenario 3	big.LITLLE	7483573288	4982718550	3.72885E+19
	SOA (Lee2017)	8607884454	9883725757	8.5078E+19
	PHISA	3972088769	2676809468	1.06325E+19
Scenario 4	big.LITLLE	6034435265	3866209661	2.33304E+19
	SOA (Lee2017)	7338408284	8350770323	6.12814E+19
	PHISA	3206043770	1995631068	6.39808E+18
Scenario 5	big.LITLLE	5072463496	3145328307	1.59546E+19
	SOA (Lee2017)	6165633346	6945246381	4.28218E+19
	PHISA	2790422276	1640027989	4.57637E+18
Scenario 6	big.LITLLE	1459278035	108952978.5	1.58993E+17
	SOA (Lee2017)	1167770977	777473120.1	9.07911E+17
	PHISA	1456673686	79137193.59	1.15277E+17

B.2 TUNEd PHISA raw results values

Table B.12: Performance

	A15	A15+4A7	A15(P)+8A7
correlation	168013	101745	64052.2
covariance	151261	110075	73860.4
2mm	2.7E+07	3.9E+07	2.1E+07
3mm	4.3E+07	5.2E+07	3E+07
atax	3315352	2295791	1211861
bicg	2553661	2242909	1268350
cholesky	2265741	1239833	636702
doitgen	7293761	3331253	1761612
gemm	1.1E+08	4.2E+07	2.4E+07
gesummv	1.3E+08	6E+07	3.4E+07
mvt	6714679	3587764	1986457
symm	1.5E+07	1.6E+07	9531864
syr2k	2.6E+07	2.1E+07	1.1E+07
syrk	1.9E+07	1E+07	5240825
trmm	9861433	4143704	3436830
gramschmidt	1.7E+07	3.9E+07	2.2E+07
lu	6390694	5856566	3343264
convolution-2d	4E+07	1.9E+07	9602851
Fdtd-apml	2.4E+07	1.2E+07	6288539

Table B.13: Energy

	1_0	1_4	1(b)_8
correlation	4.1E-05	1.9E-05	1.5E-05
covariance	3.8E-05	2.1E-05	1.7E-05
2mm	0.00731	0.00687	0.00492
3mm	0.01117	0.00937	0.00694
atax	0.00091	0.00041	0.0003
bicg	0.00075	0.0004	0.0003
cholesky	0.00061	0.00023	0.00016
doitgen	0.00195	0.00062	0.00045
gemm	0.02602	0.00743	0.00569
gesummv	0.03034	0.01069	0.00804
mvt	0.0015	0.00063	0.00046
symm	0.00364	0.0028	0.00208
syr2k	0.00739	0.0036	0.00249
syrk	0.00493	0.00179	0.00124
trmm	0.0025	0.00075	0.00079
gramschmidt	0.00436	0.0062	0.00442
lu	0.00163	0.0011	0.0008
convolution-2d	0.01037	0.00325	0.00216
Fdtd-apml	0.00617	0.00219	0.00154

Table B.14: Performance vs Niagara-like

	A15	A15(P)+4A7	Niagara
correlation	168013	106977	166574
covariance	151261	117966	165461
2mm	2.7E+07	3.9E+07	38558751
3mm	4.3E+07	5.3E+07	46688703
atax	3315352	2296942	2740877
bicg	2553661	2242915	2637490
cholesky	2265741	1259940	1832253
doitgen	7293761	3347108	4444602
gemm	1.1E+08	4.2E+07	61070058
gesummv	1.3E+08	6E+07	81921199
mvt	6714679	3591286	3190544
symm	1.5E+07	1.6E+07	15010840
syr2k	2.6E+07	2.1E+07	34239923
syrk	1.9E+07	1E+07	17082189
trmm	9861433	4460487	13989463
gramschmidt	1.7E+07	3.9E+07	30683078
lu	6390694	5856543	8409925
convolution-2d	4E+07	1.9E+07	58609912
Fdtd-apml	2.4E+07	1.2E+07	26814534

Table B.15: Energy vs Niagara-like

	1_0	1(b)_4	8(b)
correlation	4.1E-05	1.98E-05	2.73E-05
covariance	3.8E-05	2.20E-05	2.81E-05
2mm	0.00731	6.88E-03	6.97E-03
3mm	0.01117	9.38E-03	8.50E-03
atax	0.00091	4.13E-04	5.20E-04
bicg	0.00075	4.02E-04	4.80E-04
cholesky	0.00061	2.36E-04	2.92E-04
doitgen	0.00195	6.24E-04	7.87E-04
gemm	0.02602	7.43E-03	1.01E-02
gesummv	0.03034	1.08E-02	1.38E-02
mvt	0.0015	6.29E-04	5.76E-04
symm	0.00364	2.80E-03	2.52E-03
syr2k	0.00739	3.61E-03	5.40E-03
syrk	0.00493	1.79E-03	2.75E-03
trmm	0.0025	8.00E-04	2.19E-03
gramschmidt	0.00436	6.27E-03	4.90E-03
lu	0.00163	1.10E-03	1.43E-03
convolution-2d	0.01037	3.25E-03	7.58E-03
Fdtd-apml	0.00617	2.19E-03	4.09E-03

Appendix C - RESUMO EM PORTUGUÊS

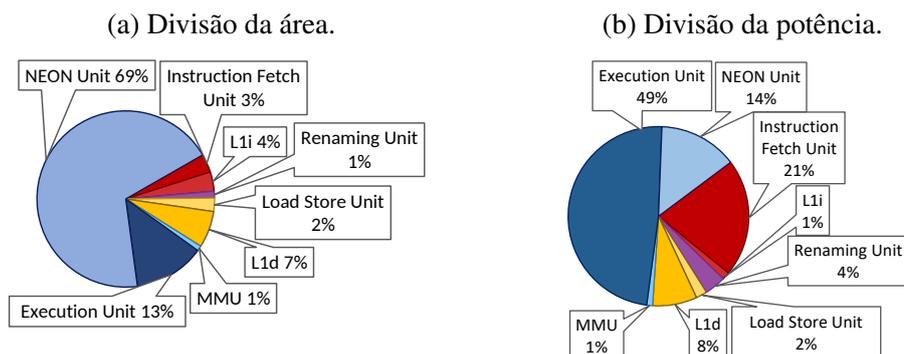
C.1 Introdução

Por gerações, os processadores de propósito geral implementam instruções especializadas na forma de extensões de conjuntos de instruções Instruction Set Architecture (ISA) com o objetivo de aumentar o desempenho de aplicações emergentes. Contudo, tais extensões impõe um custo significativo na área e potência do processador. Um exemplo está nas instruções do tipo instrução única, múltiplos dados Single Instruction Multiple Data (SIMD) e de ponto flutuante Floating-Point (FP), cujas *pipelines* podem representar mais da metade da área total de um núcleo do processador. A figura C.1 mostra a divisão de área e potência do processador A15 da ARM e destaca que a área dos componentes responsáveis pelas instruções NEON (FP e SIMD na arquitetura ARM) ocupa 69% da área do processador.

Além disso, a figura C.2 mostra que a quantidade de instruções NEON usadas em aplicações de *thread* única (*single-thread*) é muito baixa. Além disso, a tabela C.1 mostra que em aplicações paralelas, a quantidade de instruções do tipo NEON é muito pequena nas regiões seriais, mesmo em aplicações em que essas regiões são maiores.

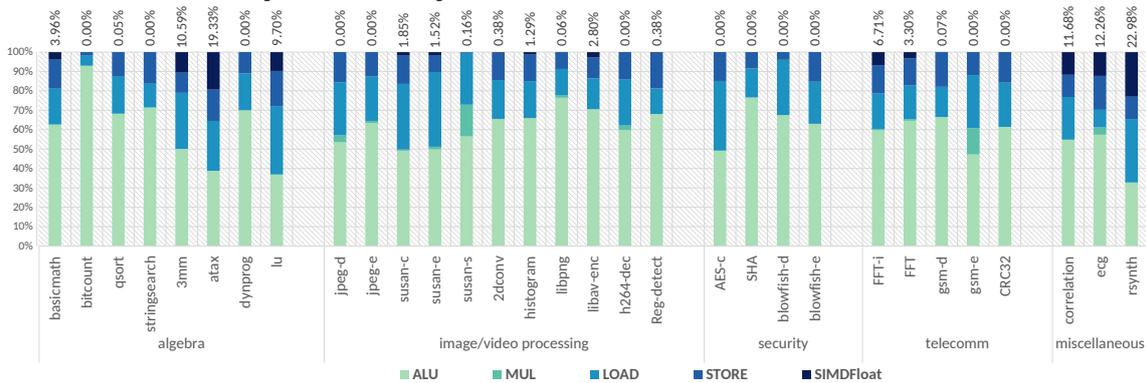
Nesta tese, são propostas soluções para reduzir a quantidade de suporte que é dado a extensões de instruções em processadores multinúcleo assimétricos Asymmetric Multicore (AMC) (sistemas que usualmente implementam núcleos de alto desempenho - núcleos grandes - e alta eficiência de área/energia - núcleos pequenos), aprimorando sua eficiência em área e energia. Inicialmente, é introduzido o sistema multinúcleo de ISA parcialmente heterogênea (Partially Heterogeneous ISA (PHISA)). PHISA é composto

Figure C.1: Divisão da área e potência por componente de um processador ARM A15 de acordo com o McPAT.



Fonte: O Autor

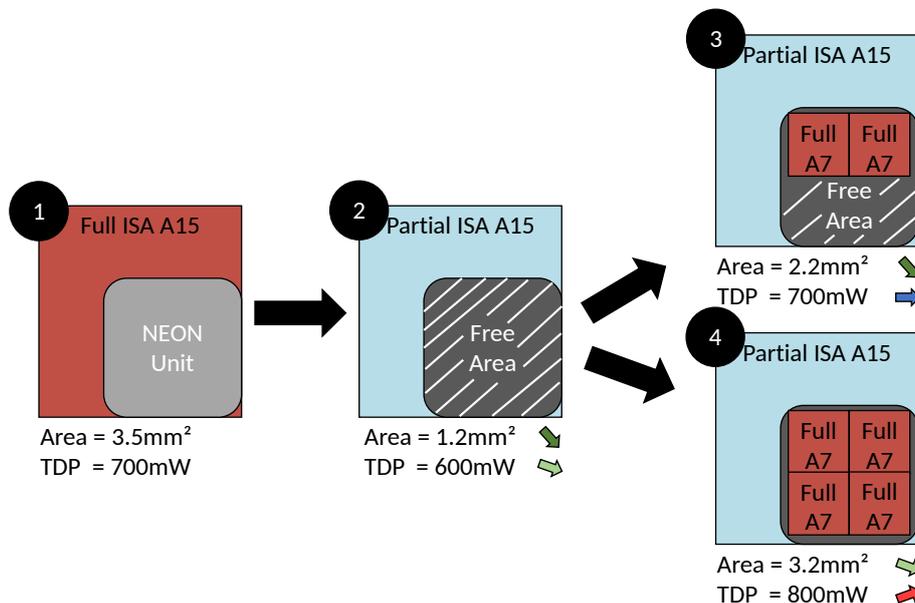
Figure C.2: Divisão do tipo de instruções em diversas aplicações *single-thread*. São destacadas as instruções de extensões NEON.



Fonte: O Autor

de núcleos heterogêneos com uma única ISA base, mas funcionalidades diferentes. Em outras palavras, alguns dos núcleos deste sistema heterogêneo não implementam completamente as caras extensões de ISA, mas ainda assim todos compartilham uma ISA base mútua. Desta forma, ao substituir núcleos de ISA completa por núcleos de ISA parcial e migrando tarefas sempre que necessário, é possível liberar recursos valiosos de área e potência do projeto do processador, sem abrir mão completamente do suporte as extensões de ISA. A figura C.3 mostra como esse sistema é pensado. Partindo de um núcleo A15 completo (1), remove-se as instruções NEON (2) e na mesma potência (3) ou área

Figure C.3: (1) Núcleo A15 tradicional. (2) Núcleo A15 sem instruções NEON (ISA parcial). (3) Núcleo A15 de ISA parcial + 2 núcleos A7 de ISA completa - mesma potência máxima (TDP). (4) Núcleo A15 de ISA parcial + 4 núcleos A7 de ISA completa - mesma área.



Fonte: O Autor

(4) liberadas adiciona-se novos núcleos.

Por outro lado, enquanto a migração de tarefas é eficiente em aplicações de *thread* única, este pode não ser o caso em aplicações paralelas. Migrações podem aumentar o

Table C.1: Caracterização das regiões de interesse de aplicações em termos do tamanho da região paralela e da quantidade de instruções SIMD/FP na região serial.

	Benchmark	Razão paralela	Razão de SIMD/FP serial
parsec	bodytrack	99.10%	0.05%
	ferret	98.24%	0.10%
	dedup	98.18%	0.00%
	facesim	97.56%	0.14%
	cholesky	96.56%	0.88%
	freqmine	95.38%	0.01%
	parvec	swaptions	99.99%
fluidanimate		99.67%	0.05%
streamcluster		99.60%	0.01%
canneal		99.58%	0.02%
blackscholes		99.55%	0.01%
vips		98.94%	0.14%
polybench		bicg	100.00%
	fdtd-apml	99.99%	0.00%
	convolution-2d	99.99%	0.00%
	gemm	99.97%	0.06%
	symm	99.95%	0.00%
	syrk	99.90%	0.04%
	syr2k	99.89%	0.04%
	atax	99.73%	0.06%
	2mm	99.27%	1.03%
	mvt	98.82%	1.11%
	gesummv	98.76%	0.46%
	3mm	98.68%	1.66%
	doitgen	98.41%	0.83%
	trmm	82.25%	6.74%
	correlation	81.95%	10.10%
	gramschmidt	78.70%	11.72%
	covariance	77.40%	17.33%
lu	68.71%	0.09%	
splash2x	ocean_ncp	99.85%	0.01%
	barnes	99.74%	0.02%
	ocean_cp	99.63%	0.04%
	radix	99.37%	0.00%
	raytrace	99.19%	0.12%
	lu_cb	98.48%	0.11%
	water_nsquared	97.92%	0.20%
	lu_ncb	97.86%	0.15%
	radiosity	97.81%	0.22%
	fft	97.38%	0.21%
	water_spatial	94.12%	0.66%
	cholesky	75.95%	2.77%

tempo em que uma das múltiplas *threads* precisa para atingir seus pontos de sincronização, criando assim um gargalo em sua execução. Para tais aplicações, é proposto aprimorar o sistema PHISA com um despachador de instruções fortemente acoplado **Tightly Coupled Instruction Offloader (TUNE)**. A arquitetura com TUNE implementa um sistema PHISA cujo núcleo grande implementa parcialmente a ISA e é responsável pela execução das regiões seriais das aplicações. Os núcleos pequenos, por outro lado, implementam toda a ISA completa do sistema e são responsáveis pela execução das regiões paralelas da aplicação. Sempre que a região sequencial da aplicação precisar executar uma instrução não implementada no núcleo grande, o TUNE irá despachar estas operações para os núcleos pequenos de forma transparente. Este sistema nós chamamos de TUNEd PHISA e é exemplificado na figura C.4.

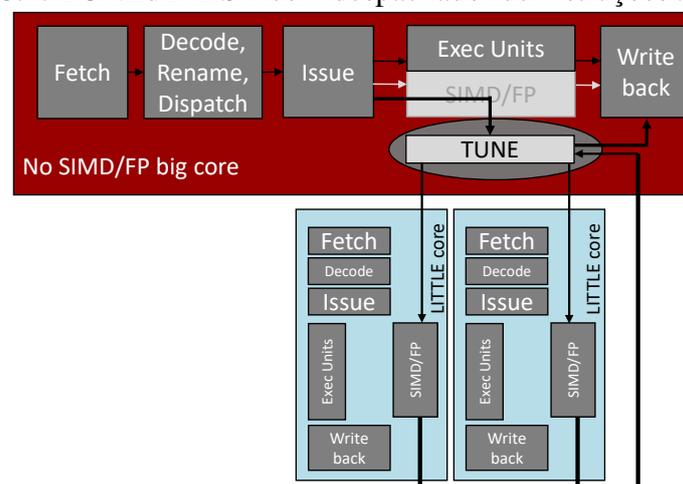
Nesta tese, é mostrado como o PHISA e TUNE podem ser usados para melhorar o desempenho e consumo energético ambos de aplicações seriais e paralelas, quando comparado a projetos tradicionais de AMCs.

C.2 Arquiteturas desenvolvidas

C.2.1 PHISA Multi Núcleos

O desenvolvimento do PHISA multi núcleos parte da ideia de que é possível abrir mão de componentes específicos de alguns núcleos para adicionar outros componentes que potencialmente podem acelerar mais aplicações. No caso desta tese, são adicionados,

Figure C.4: TUNEd PHISA com despachador de instruções SIMD/FP.



Fonte: O Autor

na mesma área ou potência dos componentes de SIMD e FP, novos núcleos pequenos. Esta estratégia cria um processador multi núcleo com maior capacidade de executar diferentes aplicações em paralelo.

Contudo, como o sistema agora possui núcleos que não são capazes de executar algumas instruções do conjunto completo da ISA, se faz necessário criar uma forma de manter o suporte a essas instruções nos nesses núcleos parciais. Para isso, foram criados diferentes escalonadores que mantêm essa compatibilidade migrando tarefas dos núcleos parciais para núcleos completos quando essas tarefas requerem o uso das instruções removidas. Além disso, também se explora o uso de emulação de instruções em núcleos parciais.

C.2.2 TUNEd PHISA

O TUNEd PHISA é uma extensão do sistema PHISA pensado para otimizar aplicações *multi-thread*. Como a migração de tarefas feita no sistema PHISA normal pode criar desbalanceamento de *threads*, o TUNEd PHISA propõe explorar o despacho de instruções diretamente de núcleos parciais para as unidades funcionais dos núcleos paralelos.

A proposta do TUNEd PHISA é usar um AMC composto de um núcleo grande e parcial para executar as regiões seriais da aplicação (já que essas executam usando apenas uma *thread*) e vários núcleos pequenos e completos para executar as regiões paralelas. Essa idéia parte da observação de que nas aplicações paralelas, as operações de SIMD e FP são majoritariamente executadas nas regiões paralelas e raramente nas seriais, de forma que remover o suporte a essas instruções do núcleo grande não causará grande impacto.

Contudo, essas instruções ainda precisam ser executadas nos núcleos grandes nos casos em que elas ocorrem. Para isso, é desenvolvido um despachador de instruções que é chamado de TUNE. O TUNE é ligado diretamente no estágio de despacho de operações no pipeline do núcleo grande. Ele irá dividir as operações (quando necessário) SIMD ou FP e despachá-las diretamente para as unidades de execução dos núcleos pequenos. Após a execução, os resultados retornam para o núcleo grande, onde eles serão comitados e escritos no banco de registradores deste núcleo. Portanto, o núcleo grande não muda seu processo de execução das instruções removidas e ele vê as unidades SIMD e DP dos núcleos pequenos como se fossem suas próprias unidades. Contudo, como essas unidades

estão em núcleos mais afastados, a latência de execução dessas instruções irá aumentar.

C.3 Metodologia

Nesta tese foram usadas diferentes ferramentas de modelamento e simulação de arquiteturas de computadores. Para extrair os dados de área e potência dos diferentes processadores modelados, foi utilizado o McPAT, uma ferramenta amplamente usada na comunidade. Já para as simulações, foi usado o gem5 (também amplamente usado para simular diversas organizações e arquiteturas) e um simulador desenvolvido propriamente para o PHISA multi núcleos.

Além disso, foram usadas diversas aplicações de várias fontes. Usamos benchmarks de vários conjuntos usados para avaliar o desempenho de sistemas dos mais variados ramos. Também foram usadas aplicações reais (de bibliotecas de código aberto) de processamento de imagens, vídeo e internet das coisas.

Várias configurações de sistemas foram testadas. Além disso, comparamos nossos sistema PHISA e TUNEd PHISA com configurações de sistemas reais, como o big.LITTLE e DynamIQ da ARM e o Niagara da SUN.

C.4 Resumo dos resultados

Os resultados demonstram a efetividade dos sistemas PHISA em relação aos sistemas tradicionais de ISA única. O PHISA consegue, com o mesmo Thermal Design Power (TDP) máximo e em uma área menor, ter melhor desempenho e consumo energético do que os sistemas tradicionais. Também avaliamos o uso de emulação de instruções junto com a migração de tarefas e verificamos que é possível usar essa estratégia para diminuir o número de migrações no sistema.

Do ponto de vista do TUNEd PHISA, os resultados também mostram que com o aumento do número de núcleos do sistema (na mesma área original), é possível aumentar a exploração de Thread-Level Parallelism (TLP) e obter mais desempenho do que o sistema tradicional. Esse aumento do desempenho é grande o suficiente para compensar a perda devido aos despacho de instruções nas regiões seriais (devido ao uso de um núcleo grande parcial). Além disso, o consumo de energia no sistema com TUNE também é menor do que no sistema tradicional.