

**HYBRID CODE PLACEMENT FOR
HETEROGENEOUS ARCHITECTURES**

MARCELO PEREIRA NOVAES

**HYBRID CODE PLACEMENT FOR
HETEROGENEOUS ARCHITECTURES**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte
Fevereiro de 2018

MARCELO PEREIRA NOVAES

**HYBRID CODE PLACEMENT FOR
HETEROGENEOUS ARCHITECTURES**

Dissertation presented to the Graduate Program in Ciência da Computação of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Ciência da Computação.

ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

February 2018

© 2018, Marcelo Pereira Novaes
Todos os direitos reservados

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Novaes, Marcelo Pereira

N935h Hybrid code placement for heterogeneous
Architectures / Marcelo Pereira Novaes. — Belo
Horizonte, 2018.
xx, 58 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal
de Minas Gerais – Departamento de Ciência da
Computação.

Orientador: Fernando Magno Quintão Pereira
Coorientador: Vinicius Tavares Petrucci

1. Computação – Teses. 2. Compiladores
(Computadores). 3. Arquitetura de computador.
I. Orientador. II. Coorientador. III. Título.

CDU 519.6*33(043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO


Hybrid Code Placement for Heterogeneous Architectures


MARCELO PEREIRA NOVAES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG


PROF. VINICIUS TAVARES PETRUCCI - Coorientador
Departamento de Ciência da Computação - UFBA


DR. ABDOULAYE GAMATIÉ
Centre National de la Recherche Scientifique - LIRMM


PROF. ADRIANO ALONSO VELOSO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 23 de fevereiro de 2018.

Acknowledgments

Em relação a minha dissertação:

- A Universidade de Minas Gerais, pela aceitação no programa, e a CAPES pelo financiamento durante o mestrado. To the CONTINUUM project for funding my scientific endeavors while in Montpellier, France.
- Ao Fernando, que me aceitou, orientou e o qual eu me inspiro ao ve-lo conciliar: resultado, talento, disciplina, e cuidado com as pessoas.
- Ao Vinicius, que aceitou ser meu co-orientador e me direcionou no projeto em muitas oportunidades.
- To the professor Abdoulaye, that believed in our project, hosted me on his lab and joined us in this journey.
- Ao professor Renato Assunção, que me deu liberdade de explorar e escolher qualquer projeto, em qualquer área, mesmo que dessa escolha decorresse a troca de orientador.
- Ao LaC, Laboratório de Compiladores, que é uma família e me aceitou de braços abertos. A todos do ninho (the-dragon-nest).
- Ao professor Adriano que aceitou fazer parte da banca de avaliação.
- A todos os funcionários da UFMG/UFBA/LIRMM que me ajudaram, como o pessoal do CRC/UFMG, a Sônia da UFMG, e a Caroline Lebrun do LIRMM.

Em relação ao lado pessoal:

- Aos meus pais, Celia e Francisco, que sempre me dão apoio em minhas escolhas. A minha irmã Luisa, por ser minha grande companheira nessa vida acadêmica. A minha irmã Leila que dará luz em breve a minha sobrinha Fernanda. A toda a minha família de Belo Horizonte que me acolheu pelo período do mestrado.
- Aos meus amigos, que fazem a vida mais leve, de Salvador, Belo Horizonte e Montpellier. Em especial ao meu primo André, que sempre me apoia e ajuda nas decisões mais importantes e ao Caio, que foi de Salvador para o laboratório.

À todos que de alguma maneira contribuíram para a realização dessa dissertação, meu muito obrigado!

Je remercie tous ceux qui, d'une manière ou d'une autre, ont contribué à la réussite de ce travail!

To all those who helped to the elaboration of this master's dissertation in any way, thank you!

Abstract

Modern computer architectures are becoming each day more heterogeneous. This heterogeneous design emerges through the combination, within the same hardware, of several different processors. Choosing the best hardware configuration for a given program is difficult, because programs, even of moderate size, go through many phases during execution. These phases benefit differently from the same hardware configuration. Researchers have used a plethora of techniques to deal with this problem, but two techniques seem to stand out: dynamically or statically. The former is implemented at the runtime level, be it through an operating system or a scheduler/middleware. The latter is implemented at compilation level. In this dissertation, we investigate the mixing of both approaches achieving a synergy that, otherwise, could not be attained by each technique individually. On this new scheme, we propose an instrumentation framework that produces an adaptive program that uses both source-code and run-time information to make decisions. We have implemented our instrumentation framework in the LLVM compiling infrastructure. To demonstrate that our proposal is useful and effective, this master's dissertation compares this new code generator, called Astro, with state of the art approaches. The goal was reducing the energy consumption of programs while maintaining performance constraints. We evaluate on applications among different benchmarks and embedded boards.

Keywords: Compilers, heterogeneous architectures, energy-efficient systems, adaptive programs, non-assisted instrumentation, machine learning, classification, LLVM.

Resumo

Arquiteturas de computadores modernas estão se tornando a cada dia mais heterogêneas. Essa heterogenidade emerge da possibilidade de combinação, dentro de um mesmo dispositivo, de diferentes tipos de processadores. Escolher a melhor configuração desses processadores é uma tarefa difícil, porque programas, mesmo em tamanhos moderados, apresentam diferentes fases durante a sua execução. Essas fases se beneficiam de forma diferente do dispositivo. Pesquisadores vêm utilizando diversas técnicas para lidar com o problema, mas duas delas se destacam: a dinâmica e a estática. A primeira é implementada em tempo de execução, seja por um sistema operacional ou um escalonador. A última é implementada em tempo de compilação. Nessa dissertação, nós investigaremos a mescla das duas abordagens de modo a conseguir uma sinergia a qual não seria obtida por cada uma das técnicas individualmente. Nesse novo esquema, nós usamos uma ferramenta de instrumentação que cria programas que se migram sozinhos usando ambas informações retiradas do código fonte e informações coletadas em tempo de execução para fazer as decisões. Nós implementamos a ferramenta de instrumentação na infraestrutura de compilação LLVM. Para demonstrar que nossas ideias são úteis e efetivas, essa dissertação de mestrado compara esse novo gerador de código, chamado Astro, com abordagens no estado-da-arte, o utilizando para reduzir o consumo de energia (mantendo restrições de performance) em conjuntos de aplicações entre diferentes placas embarcadas.

Palavras-chave: compiladores, arquiteturas heterogêneas, economia de energia, programas adaptativos, instrumentação de código automática, aprendizagem por reforço, LLVM.

List of Figures

1.1	Energy vs Processing time spent by two PARSEC benchmarks using <code>sim-small</code> inputs. The notation <code>xLyB</code> denotes <code>x</code> <u>L</u> ITTLE cores, and <code>y</code> <u>b</u> ig cores.	5
1.2	(a) Simple matrix multiplication implemented in C. (b) The Nvidia TK1 board. (c) NI 6009 Data Acquisition Device. (d) Synchronization circuit.	7
1.3	(a) Power profile of program seen in Figure 1.2. The NI 6009 sample rate was 1000 samples/sec. (b) Zoom of the power profile obtained during the last phase of the program. We show power data obtained using one low-frequency core (Low), and using one ARM A15 core (High).	8
1.4	Best configurations for seven PARSEC applications, given an acceptable slowdown of 1% or 5% compared to fastest configuration.	9
3.1	The Astro Framework.	22
3.2	Mapping the functions in Figure 1.2 (a) to program phases.	26
3.3	The Actuation Algorithm.	29
3.4	(a) Instrumentation to mine features. (b) Final instrumentation, inserted in production code.	31
4.1	Simulated approximated optimal policy for <code>fluidanimate</code> . The Y-axis represents different hardware configurations. The X-axis shows the application's execution time in seconds.	39
4.2	Comparison between Astro and other 6 strategies.	40
4.3	Supervised Learning applied to the <code>fluidanimate</code> application from PARSEC benchmark in an action space of 24 possible combinations of heterogeneous cores. Three learning models that achieved the best results of the five tested: Gaussian Process Classifier, the Multi-layer Perceptron (MPL), the kNN.	42

4.4	Supervised learning applied to the 10 different phases of the Meabo micro-benchmark [ARM-software]. Fixing the best model for the three approaches, that were the same (the GPC in the hybrid had also the same precision as the Gaussian Naive Bayes). Part (a) of the Figure shows the result considering three dynamic features (relative CPU load to the number of used cores, instruction per cycles and cache miss ratio), (b) shows the result using 16 static features. The (c) shows the result of the hybrid approach, combining both features.	44
4.5	Comparison between three different implementations of Astro. They vary in the way they choose information to feed the neural-network. Dynamic: uses only hardware phases (Definition 3.2.3); Static: uses only program phases (Definition 3.2.2); Hybrid: uses both phases. Each dot along the X-axis represents one execution of <code>fluidanimate</code> . The Y-axis represents the maximum reward observed in that execution. We keep the state of the neural-network across executions, so that it can learn. We show results after 80 of such “episodes”.	46
4.7	This figure shows the binary code size on different applications. The Y-axis represents the size in Kybtes. The X-axis represents the different applications. Plot shows from the original size in black, learning library adds in gray and deploy adds in light gray. Binary size computed via the “ <code>du -h</code> ” command.	49

List of Tables

- 2.1 Comparison between different solutions to . *Level*: at which level the technique is implemented: Architecture (A), Operating System (O), Compiler (C) or Library/Programming model (L). *Code*: "Yes" if there is source code instrumentation/manipulation. *Auto*: "Yes" if it is performed automatically, without user intervention/annotation. *Runtime*: "Yes" if technique considers runtime information. *Learn*: "Yes" if technique adapts/learns a model from the target architecture 19

- 4.1 Time (Top, in seconds) and Energy (Bottom, in Joules) comparison between Astro, GTS and Hipster. Each value represented as pair: median|standard deviation. "Static" is the purely static version of Astro, shown in Figure 3.4 (b). "Hybrid" is the version that uses runtime information to improve on the static decisions, shown in Figure 3.4 (c). Bold values highlight best results; underscore boxes show results within error margin. 48

Contents

Acknowledgments	ix
Abstract	xi
Resumo	xiii
List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 Contributions	3
1.2 Empirical Observation	3
2 Literature Review	13
2.1 Handling the Code Placement Problem	13
2.1.1 The Static solution: decision at compiling time	13
2.1.2 The Dynamic solution: decision at running time	15
2.1.3 Hybrid Motivation: Static vs Dynamic decisions	15
2.1.4 The Hybrid: combining Dynamic and Static information	17
2.1.5 Trade-offs	18
2.2 Non-assisted code-placement	19
3 Astro framework	21
3.1 Reinforcement learning via Q-Learning	22
3.2 Components of the Astro framework	24
3.2.1 Part 1: Phase Partitioning	24
3.2.2 Part 2: Actuation	29
3.2.3 Part 3: Final Code Generation	32
3.3 Relationship between Astro and the most related work	33

4	Evaluation	37
4.0.1	Experimental Setup	37
4.0.2	Results in the Simulated Environment	38
4.0.3	Results in an Actual Device	45
4.0.4	Overhead analysis	47
5	Conclusion	51
5.1	Open questions and clarifications	51
	Bibliography	55

Chapter 1

Introduction

Modern computer architectures are becoming each day more heterogeneous [62]. This heterogeneous design emerges through the combination, within the same hardware, of several different processors, such as big/little multi-core Central Processing Units (CPUs) [13][23], Graphics Processing Units (GPUs) [42] and Digital Signal Processors (DSPs) [49]. An advantage of this design is the possibility of allocating to each application the hardware configuration that best suits it. A hardware configuration consists of a number of cores, their type and their frequency level. We say that a configuration H_1 suits a program better than another configuration H_2 if H_1 runs the program more efficiently than H_2 , according to some metric such as runtime or energy consumption. Nevertheless, even though we have today the possibility of choosing among several configurations, the one that better fits the needs of a certain program, we still have no clear technique to perform this choice seamlessly.

Often a program has regions of code that benefit differently from distinct processors. We call the task of allocating program parts to processors the *code placement problem*. Typically, there are two ways to solve this problem: dynamically, or statically. Dynamic approaches [38; 44; 46] are implemented at the runtime level, be it through an operating system, a middleware or changes in the target program itself. Static approaches [25; 39; 45; 58] are implemented at the compiler level. The main advantage of the dynamic approach is the fact that it can take advantage of runtime information to improve the quality of the choices it makes. Static techniques, in turn, provide reduced runtime cost and better leverage of program characteristics. In this dissertation, we claim that it is possible to join these two approaches, achieving a synergy that, otherwise, could not be attained by each technique individually.

To fundament this claim, we show a few techniques to mix static and dynamic information and discuss a few others. Our final approach starts from a technique that has

been already proven effective to schedule computations in heterogeneous architectures: *Reinforcement learning*. Nishtala *et al.* [44] have shown that reinforcement learning helps in finding good hardware configurations to applications subject to varying dynamic conditions. The beauty of this approach is adaptability: the same principles provide the means to explore a vast universe of states, formed by different hardware setups and runtime data changing over time. Given enough time, well-tuned heuristics let the learning algorithm find a set of scheduling decisions that suits the underlying hardware. Yet, “enough time” can be too much time. The universe of possible runtime states is unbounded, and program behavior is hard to predict without looking into its source code. To speedup convergence, we bring the compiler into the fray.

The compiler gives us two benefits. First, it lets us mine program features, which we can use to train the learning algorithm. Second, it lets us instrument the program. This instrumentation allows the program itself to provide feedback to the scheduler, concerning the code region currently under execution. Based on previous knowledge, collected statically, about characteristics of that region, the scheduler can take immediate action. An action consists in choosing a new state to represent program behavior, and collecting the reward related to that choice. Such feedback is then used to fine-tune and improve scheduling decisions. As we show in Section 4, convergence is faster, and runtime smaller when deciding for static and dynamic combination of information on the state representation.

To validate our ideas, we have materialized them into a framework to instrument and execute applications in heterogeneous architectures: the *Astro System*. To collect static program characteristics, and instrument code, we use the LLVM compilation infra-structure [35]. We show that this new code generator is able to reduce the energy consumption of programs running on an Odroid XU4 architecture. Our experimental results, obtained in well known diverse benchmarks, show that we can achieve energy savings of up to 8%, when compared to state-of-the-art techniques, such as Hipster [44]. Such numbers result from the following contributions:

Observations: in Section 1.2, we demonstrate that the different parts of a program, i.e., syntactic regions within its code, are an important factor determining its performance, in terms of speed and energy, when running on a heterogeneous hardware. This observation points us to the key insight: the possibility of augmenting an adaptive runtime apparatus with awareness of program characteristics.

Compiler: in Section 3.2.1, we explain how to collect and discretize program features, and in Section 3.2.2, we explain how to instrument a program, so to use said features to fine-tune an adaptive code placement algorithm. The choice of program

features and runtime characteristics that this apparatus requires are also original contributions of this work.

Runtime: in Section 3.2.3, we show how to integrate the static information that we collect with a runtime environment that adapts to the environment. We offload the learning process to a server. Hence, runtime adaptation does not burden the application after it stabilizes. The process is gradual: if necessary to learn more, we reconnect the device to the server.

1.1 Contributions

This dissertation will present an overview of the code placement problem and introduce the Astro framework which:

1. Creates an adaptive program that migrates by itself, without human or scheduler intervention.
2. Exploits both source code and run-time information to decide for the best choice, achieving a synergy that only with one of them wasn't possible.
3. Evaluates different learning approaches as supervised, for a simplified space problem, and reinforcement learning techniques for a larger space problem.
4. To the best of our knowledge, Astro is the first system that feeds a learning model with both static and dynamic information to solve the placement of computations in a heterogeneous device.

Awards: A fork proposal of the Astro project, called “Intelligent DVFS”¹ [Int], has earned a Google Research Award for Latin America 2017 [Goo]. So, since August 2017, the lab has receiving funding from Google Inc. The project is an implementation of the method targeting mobile devices with the Android Operating System. The project is executed by Fernando (advisor) and Junio (a first year master student in our lab).

1.2 Empirical Observation

We motivate our work by presenting and discussing observations obtained from running a few experiments. First, we find that different hardware configurations yield

¹It was one of 27 approved projects from 281 proposals were submitted from nine different countries in Latin America.[Goo]

very different tradeoffs between power consumption and runtime speed for a program (Fig. 1.1). As a second observation, we notice that this behavior happens because programs have *power phases*: depending on the operations that they perform, they might consume more or less power per time unit (Fig. 1.2). Finally, we observe that the best hardware configuration for a program might not suit the needs of a different application (Fig. 1.4). Central to the discussion in this section is the notion of a *hardware configuration*:

Heterogeneous architectures Heterogeneous Computing (HC) was defined as “the well-orchestrated and coordinated effective use of a suite of diverse high-performance machines (including parallel machines) to provide superspeed processing for computationally demanding tasks with diverse computing needs. An HC system includes heterogeneous machines, high-speed networks, interface” [32]. Over the time, the heterogeneity was extended at different levels such as inside the same computer. So, HC, later called Heterogeneous Architectures, could be broadly defined as “the use of different processing cores to maximize performance” [11]. Recent architectures have a different type of cores integrated into the same chip. It is usually organized by clusters. One example used in this dissertation’s evaluation are Heterogeneous Processors of the ARM vendor, known as ARM Big.LITTLE processors.

Definition 1.2.1 (Hardware Configuration) *A heterogeneous architecture is formed by a set $P = \{p_1, p_2, \dots, p_n\}$ of n processors. A hardware configuration is a function $H : P \mapsto \text{Boolean}$. If $H(p_i) = \mathbf{True}$, then processor p_i is said to be active in H , otherwise it is said to be inactive.*

The Universe of Hardware Configurations We observe that the same application might benefit differently from different hardware configurations. This benefit is measured in terms of processing time and energy consumption. Figure 1.1 illustrates this fact. The figure shows how two benchmarks from the PARSEC suite – *Freqmine* and *Streamcluster* – fare on an Odroid XU3/XU4 board². This hardware features 4 Cortex-A15 2.0Ghz cores and 4 Cortex-A7 1.4Ghz cores. Following a nomenclature adopted by ARM, we shall call the A15 cores *big*s, and the A7 cores *LITTLE*s. By switching on and off the different cores, we have 24 different hardware configurations³

²Both XU3 and XU4 adopt the same Samsung Exynos 5422 big.LITTLE ARM processor

³We have $24 = 5 \times 5 - 1$ configurations, because we do not count the setup in which all cores are off.

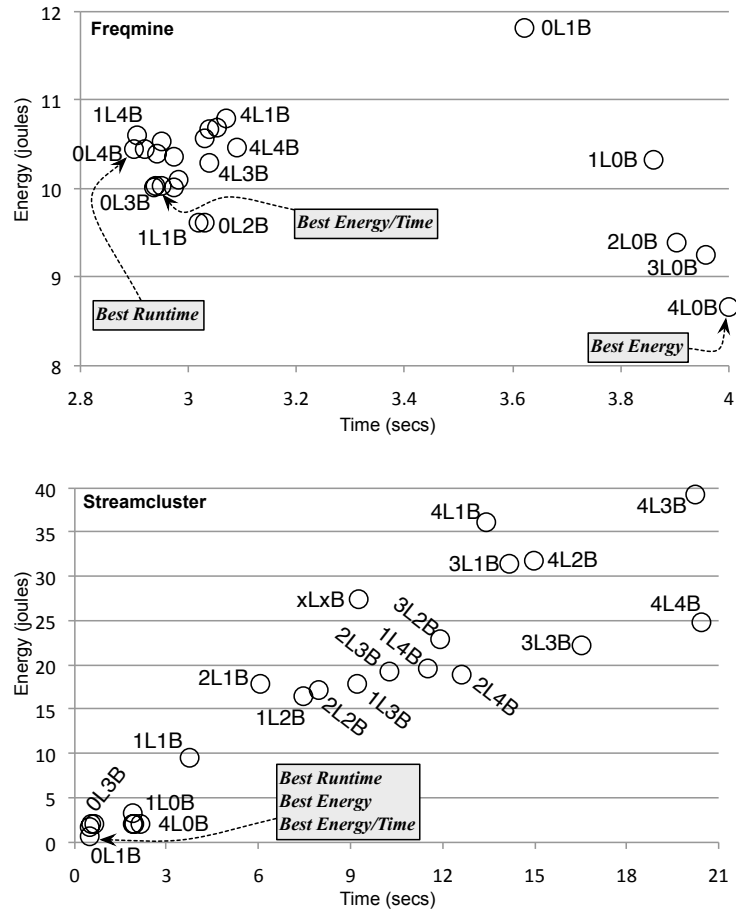


Figure 1.1. Energy vs Processing time spent by two PARSEC benchmarks using *sims*small inputs. The notation $xLyB$ denotes x LITTLE cores, and y big cores.

Each dot in the figure represents the average of 10 executions on the same configuration, using the smallest⁴ input available in PARSEC. Variance is almost negligible, staying under 1% in every sample, for the two benchmarks. The X-axis shows the sum of the execution times of processors active in a particular configuration; hence, it is not clock time. Energy is measured with the Odroid XU3 on-board power measurement circuit and refers to work performed within the processors only; thus, peripherals are not considered.

Figure 1.1 lets us conclude that the energy and runtime footprint of applications vary greatly across different hardware configurations. For instance, the most time efficient configuration for *Freqmine* is 0L4B, i.e., four bigs and no LITTLEs (2.90secs,

⁴This experiment takes approximately 12 days using the largest inputs.

10.43J). However, the most energy efficient configuration is 4L0B (4.01secs, and 8.65J). Results are not the same for **Streamcluster**. The best energy configuration is 0L1B (0.48secs, 0.69J). This is also the most time efficient configuration. **Freqmine** shows more parallelism than **Streamcluster**; therefore, it benefits more from a larger number of cores. This diversity of scenarios happen because programs have different computing scenarios (program *phases*). Energy and runtime behavior are similar within the same phase, and potentially different across different phases.

Program phases The instantaneous power consumed by a program is not always constant. In other words, a program has *power phases*. To demonstrate this fact, we shall consider the program in Figure 1.2 (a). This is an artificial example, which we have crafted to emphasize the different phases that a program undergoes during its execution. This program performs the following actions: (i) read two matrices from text files; (ii) multiply them and (iv) prints all the matrices in the standard output. In between each of these actions we have interposed commands to read data from the standard input.

Figure 1.3 shows the power profile of this program. This chart has been produced with JetsonLeap [7], an apparatus that let us measure the energy consumed by programs running on the Nvidia TK1 Jetson board⁵. JetsonLeap is formed by three components: the target Nvidia board (Figure 1.2 (b)), a data acquisition device, which reads the instantaneous power consumed by the board (Figure 1.2 (c)), and a synchronization circuit, which lets us communicate to the power meter which program event is running at each instant (Figure 1.2 (c)).

Distinct phases exist within the same program because it might use the hardware resources differently, depending on which part of it is running. By reading performance counters, we know that during matrix multiplication, CPU is at its maximum usage. During the input/output operations, this utilization drops slightly, and other components of the hardware, such as its serial port, are more exercised instead. This fall is steep once the program is waiting for user inputs. The CPU is not the only hardware component that accounts for power dissipation. The JetsonLeap apparatus measure energy for the entire hardware. Thus, the under utilization of the CPU does not mean that overall power consumption will decrease. Nevertheless, variations in the CPU usage are likely to cause variations in the power profile of the program.

⁵In this section we use two different experimental setups: Odroid XU4 and Tegra TK1. The former gives us the richness of configurations seen in Figure 1.1. This diversity is absent on the latter, that has only one LITTLE core. However, the TK1 board gives us access to JetsonLeap, and, consequently, the ability to measure energy per programming events.

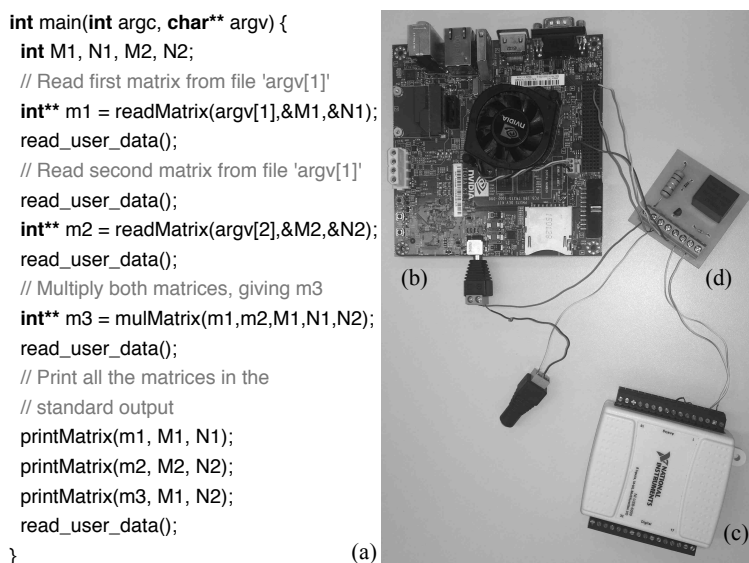


Figure 1.2. (a) Simple matrix multiplication implemented in C. (b) The Nvidia TK1 board. (c) NI 6009 Data Acquisition Device. (d) Synchronization circuit.

Discovering such program phases by means of purely dynamic techniques is possible, yet difficult. As we shall demonstrate in Section 4, we can use profiling techniques, à la Hipster [44], to identify variations in program behavior. However, this approach has two shortcomings. First, distinct program parts, with very different resource demands in terms of memory, CPU, disk and such, can display similar dynamic characteristics. For instance, we could imagine a scenario in which function `read_user_data`, in Figure 1.2 is implemented via busy waiting. In this case, instead of the valleys observed in Figure 1.3, we would encounter a power line similar to that produced by CPU-intensive functions like `mulMatrix`. Second, profiling-based techniques face a tradeoff between precision and overhead. Fast detection asks for high sampling rates; thus burdening the application which originally we intended to optimize. On the other hand, purely static approaches are not better either. Although likely to yield lower adaptation overhead, they fail to account for information only available at runtime such as varying input sizes. For instance, a static scheduler might decide always run `mulMatrix` and `read_user_data` in different configurations. However, when operating on matrices that are too small, the cost of changing the hardware configuration might already overshadow the possible gains available through more parsimonious usage of the architecture’s resources.

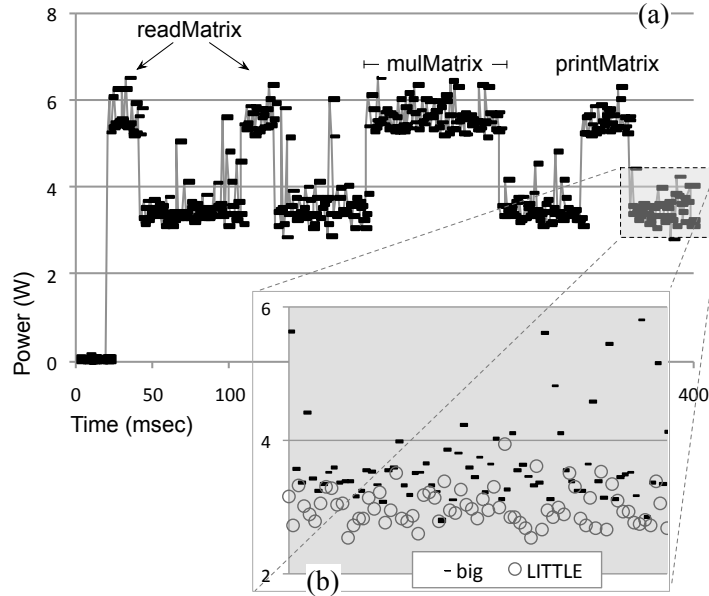


Figure 1.3. (a) Power profile of program seen in Figure 1.2. The NI 6009 sample rate was 1000 samples/sec. (b) Zoom of the power profile obtained during the last phase of the program. We show power data obtained using one low-frequency core (Low), and using one ARM A15 core (High).

The Search for the Ideal Architecture Configuration In face of the data presented in this section, it comes as no surprise that the best architecture configuration, in terms of runtime or energy consumption, differs among programs.

Figure 1.4 shows the best configurations that we have found on the Odroid XU4 setup, for six different PARSEC applications. In this case, we define the best configuration as the one that spends less energy, given a certain slowdown compared to the fastest configuration. Clearly, there is not a single winner. Configurations vary among programs, and even within the same program, given different acceptable slowdowns.

Saving Energy while maintaining Performance Constraints Save energy while maintaining performance is a multi-dimensional optimization problem. Ideally, we want the maximum performance at the cost of the minimum of energy.

The multi-dimensional optimization is applied inside a program phase. This phase is usually defined by one of the two ways:

1. We fix a code section, regardless of the time it takes to execute. At the end of the code section, we analyze the process behavior inside it.

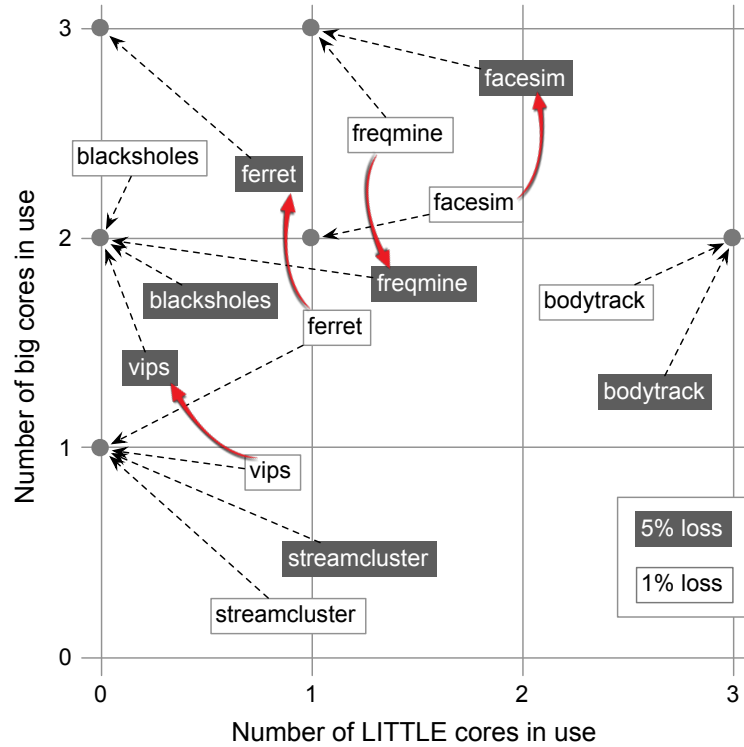


Figure 1.4. Best configurations for seven PARSEC applications, given an acceptable slowdown of 1% or 5% compared to fastest configuration.

2. We fix a specific time interval, regardless of where the application is going to stop at the end. At the end of the time interval, we analyze the process behavior inside it.

Henceforth, we call a program phase defined by any of the approaches above as phase, section, region, application region or even a code section. Inside this region, a power usage metric could be, for example, Average Power or Peak Power spent in a code section or in a fixed interval. For the performance metrics, we could consider at least 3 approaches. First, the Elapsed Time (if we fix a code section). Second, estimating how much of the application is complete⁶ (if we fix a time interval measurement). Third, use a throughput metric such as average instructions per second (what can be also application specific metrics such as a number of frames/seconds). We actually use the energy itself as a power consumption metric, even though it is quite clear that by considering energy as a metric we are already optimizing for performance also. Finally,

⁶would require previous information of instructions in a previous run

for a given application region, we have two inputs, power related and a performance-related metric and we want ideally maximize performance while minimizing power consumption.

Energy-efficiency metrics There exist plenty of metrics for the multi-dimensional problem approached in the context of hardware configuration. Before starting with multi-dimensional metrics, it is important to notice that getting one-dimensional only metric such as peak or average power, would be impractical metric for this kind of problem, as noticed by [51], a metric “should not reward a system that consumes almost no power and completes almost no work”. We extend that, by stating that in the domain of embedded and mobile devices, a metric should not reward a system that does meaningful work but consumes an exorbitant amount of power, due to the battery restrictions. There are many approaches and metrics to handle this multi-dimensional problem in the literature. Some of them are:

1. Find a metric that combines both power and performance.
2. Minimizing or maximizing one of the dimensions and apply constraints on the other dimension.

Basic metrics that consider both dimensions are energy (product of time and power), and the energy-delay product. Furthermore, metrics such as Energy per instruction (*Joules/Instructions*) or *MIPS/watt* have being used by chip vendors [26]. Furthermore, tuning these formulas in the way such as $MIPS^{alpha}/watt^{beta}$, where *alpha* and *beta* tends balance performance and energy, have been used as well. Examples would be the $MIPS^2/watt$ what relates to $energy * delay$ and $MIPS^3/watt$ which is related to $energy * delay^2$ [33].

We have focused on approach (2). We minimize energy given some slack of performance degradation permitted. This is a more clear way to provide performance guarantees from the user perspective. In the domain of heterogeneous architectures especially in mobile devices and embedded architectures, we want to provide performance guarantees in an easier way while keeping the main goal of saving energy. The decision for the shape of a function that will balance these metrics for the system, can be found on 3.2.4.

Heterogeneous multi-processors example Regarding the understanding of opportunities in one of the Big.LITTLE setups, we have two different clusters, the A7 cluster and the A15 cluster combined inside the same chip. The so-called little cluster, A7, is

designed for low load applications. The architecture is composed by processors in-order, with a dual issue, with one integer ALU and one partial ALU, one float point unit and pipeline stages varying from 8 to 10 stages depending on the instruction executed. The so-called Big cluster is more robust. In the setup A7-A15, it is composed of the A15 cores. Each of the A15 is an out-of-order processor, has a triple issue, two float point units, achieving a higher throughput on intensive loads. However, its pipeline stages vary from 15 to 24 stages depending on the executed instruction. As the number of pipeline stages usually leads to more energy consumption, the big cluster spends more energy than the little cluster. Furthermore, the low cluster frequency can be adjusted to work at a maximum of 1.4 GHz while the big cluster varies up to 2.3 GHz which also increases the energy spent.

Core migration and DVFS When handling heterogeneous processors as commented above, among the options of actions, we can deal with the dynamic voltage and frequency scaling (DVFS) as well as core migrations. A core migration over heterogeneous cores in a naive architecture could be unfeasible due to the cache incoherence between the cores. So, some heterogeneous architectures facilitate full-coherency between L2 caches of each one of the clusters (big and little cluster). In the Big.LITTLE architecture, it is done by the CCI-400 bus. Another motivation to do a core migration between two different type of cores in a big.LITTLE architecture is the small overhead. According to [24] it takes around 20.000 cycles, in a 1 GHz processor it would take 20 microseconds, in practice close to 30 micro-seconds what is lower comparing for example with a DVFS change that is expected in a few hundreds of microseconds. The big.LITTLE architectures also have some capabilities that motivate the migration between cores such as the possibility of migrating interruptions among the cores via the GIC-400 bus.

Next chapters In this chapter, we have highlighted key motivation behind our design: (i) a modern heterogeneous hardware exposes a number of different configurations that is too large to be evaluated manually; (ii) a program presents power phases, which can be more easily detected by methods that are aware of structural properties of the code. Thus, we claim that effective adaptation demands knowledge of program characteristics. Such information is readily available to the compiler; however, it is hard to be precisely acquired by techniques unaware of the program's structure. In the next chapters, we shall describe a few methodologies mixing static and dynamic analyses, to find good hardware configurations for the code sections invoked during the

execution of a program. After a few evaluations, we end up with a general methodology, henceforth called the Astro system.

Chapter 2

Literature Review

In this chapter we explain and survey the main techniques used nowadays to execute code in heterogeneous architectures. The goal of this section is to give the reader enough context to understand the benefits and limitations of the Astro framework.

2.1 Handling the Code Placement Problem

2.1.1 The Static solution: decision at compiling time

A purely static solution retrieves information from the code and based on their values, maps phases of the code to processor units. Examples of that is Etino [48]. A summary of static characteristics are:

- Source code available, source of predictions
- If code section is executed with periodicity, it can use history to predict.
- No need of monitoring execution
- Changes on types of workloads can lead to very bad allocation mistakes, leading to over requiring or minimizing actual need. Impacts direct on cost.
- We can instrument the placement in the binary already, so no need for a middle-ware manage the changes of workload.

In the table bellow, there are some static features used in our solution. Features 1-4 are from [25]. They are a subset of the original set of features presented in the dissertation. The features are retrieved for each function (as a function is our learning unit) instead of for a full program in its original propose. Furthermore, we collect via a static pass in the LLVM compiling framework. Some features were not selected as they were strongly related with GPUs which are not our target architectures. Features 10-16 and 19-23 are a subset of the features present on [3] extended with a few features

for functions instead of each loop.

#	Static code features (by function)
1	# i/o instructions (estimative)
2	# memory accesses (estimative)
3	# int operations (estimative)
4	# float operations (estimative)
5	is this code section right before a sleep lib call? (flag)
6	is this code section right before a network lib call? (flag)
7	# locks (estimative)
8	is this code section right before a sync. barrier? (flag)
9	# total of instructions (estimative)
10	# branch instructions within function
11	max loop step within function
12	max loop nest depth within function
13	# math lib calls (estimative)
14	contains loops? (flag)
15	loops perfectly nested (flag)
16	any loop has calls (flag)
17	any loop has branches? (flag)
18	contains nested loops (flag)
19	all loops have constant lower bound? (flag)
20	all loops have constant upper bound? (flag)
21	all for loops have constant stride? (flag)
22	all for loop have unit stride? (flag)
23	all loops are simple within function
24	is this an internal OpenMP function call?
	...

Someone can use these raw features, as in the table, or compose one or more of them to create new features such as: normalized int operations per total of instructions.

2.1.2 The Dynamic solution: decision at running time

#	Dynamic enviro features (per core)
1	CPU utilization
2	Number of instructions executed
3	Number of last level cache misses
4	Number of branch miss-predictions
5	Number of cache accesses
7	Number of memory reads
8	Number of memory writes
	...

A purely dynamic solution retrieves information from the workload, hardware and the process. Someone can use these raw features, as in the table, or compose one or more of them to create new features such as: Instructions per cycle (IPC), Cache Miss Ratio (CM ratio) or Cache Miss rate (CM rate). Based on it, maps phases of the code to processor units. Examples of that are Hipster ([44]) and OctopusMan ([46]). We suggest a simple table with the dynamic features used in our solution.

Dynamic characteristics:

- No code, so no prediction based on it.
- Accurate decisions based on real time analysis of the hardware and processes.
- If workload has periodicity, it can use history to predict.
- Usually have showed good results, but with the cost of monitoring execution

2.1.3 Hybrid Motivation: Static vs Dynamic decisions

We want to motivate that dynamic and static features are complementary and both of them are necessary for our solution. Consider the following C where we need to set two architectures, before "mult" and before "out":

```

1 int main() {
2     N = 100;
3     in(X,N); in(A,N); in(B,N); // initialize matrices of size NxN
4     setArchitecture(?) // (1)
5     mult(X,A,B,N); // Does X=A*B; "*" is matrix multiplication op.
6     setArchitecture(?) // (2)
7     out(X,N); // Outputs the X matrix of size NxN
8     free(X);free(A);free(B);
9     return 0;

```

10 }
}

In the question marks at (1) and (2) we are asked to choose between "LOW CORE", "HIGH CORE", to pass to the argument, representing respectively a code placement for an architecture of mult and out functions. The "LOW CORE" denotes a configuration towards low power consumption exploitation. The "HIGH CORE" denotes a configuration with high performance capabilities. What would be a reasonable choice? We start by stating that the best choice in terms of total energy would be: (1) "LOW CORE" and for (2) also "LOW CORE", making it as the following:

```

1 // Known best choice: Optimal Solution
2 int main() {
3     //...
4     setArchitecture("LOW CORE") // (1)
5     mult(X,A,B,N);
6     setArchitecture("LOW CORE") // (2)
7     out(X,N);
8     // ...
9 }
```

We could choose for take into account only static information first, i.e. the static approach only, the approach considering the static code features presented. We probably would set the matrix multiplication to HIGH CORE core and the output to the LOW CORE core. ¹

```

1 // Static only decision
2 int main() {
3     //...
4     setArchitecture("HIGH CORE") // (1)
5     mult(X,A,B,N);
6     setArchitecture("LOW CORE") // (2)
7     out(X,N);
8     // ...
9 }
```

¹For the code mentioned, an action based in a static analysis that could see more than its own function information would probably go to the optimal solution. For example, for this example, a simple constant propagation (a compiler optimization) plus computing the estimated number of instructions via static inference might easily solve it. However, keep in mind the N=100 in the source code is an oversimplification. The same input could be harder to retrieve from source code or even run-time based. So, we argue that the static choice would still be valid for one of the two: N=100 in the source code, but a simple static analysis just looking at the function level (our example); or a fancy static analysis but with input information that is hard to retrieve or run-time based.

However, the input size is 100×100 and the actual best fit (by now only looking in terms of most energy savings), both of them should run in the low-core.

Following a only dynamic approach: we could use a perf command, see the cpu usage of multiplication, as it is low, we set (1) to "LOW POWER". Then, as out start to execute, we see that the resource usage is even lower, so, no necessity to turn on the big core we set as "LOW POWER". So, it would answer it right. However, first, the signal of low usage can not be very precise when monitoring this part of the code, there is a delay of our action and state that the profiler is showing us, more than that, besides that, we become dependent to monitor the execution the whole time. Using the static to predict this phase, we could make this decision ahead of time, becoming more independent of the dynamic part, a great IO density in the next function is a strong indicative (signal) that we should use less processing. Even though the runtime overhead on dynamic approaches can be sometimes (as in [44]) negligible for servers, when we deal with embedded architectures and energy concerns, this overhead is not negligible.

2.1.4 The Hybrid: combining Dynamic and Static information

An hybrid code placement solution is a method which combines both information (source code and dynamic reads) in the placement decision problem. It tries to couple these two worlds helping placement on heterogeneous architectures. A proposed hybrid code would work as following:

1. Component 1: initial instrumentation
 - a) Instrument a source code A such as compute all static features of code phases (for simplicity, assume every function) and mark at run time which phase is running, generating a binary B.
 - b) During B running time you should be able to know which phase is running and know its static features (characteristics) values.
2. Component 2: learning (during execution of B)
 - a) Define a sampling interval which you will collect dynamic features and the function which is running.
 - b) Make a decision (according to a learning model) about which hardware configuration changes.
 - c) Act in the system changing to the decided configuration.

- d) Collect performance measures such as Energy and Performance in order to evaluate the action made.

3. Component 3: final instrumentation

- a) In the final of the learning, you will have actions for every sampling interval (decision point) and consequently to the function that were running at that interval.
- b) See which are the actions made for every phase and decide which ones remains.
- c) Take A and generate a new binary B on each phase with hardware adjusts instruction commands.

2.1.5 Trade-offs

	Static	Dynamic	Hybrid
Scheduling responsiveness	Compiler	Operating system or Middleware	Both
Advantages	Can use structure of the program to take better decisions; If the code is executed with periodicity, it can use history to predict; No run-time overhead	Possibility to use run-time information to improve the quality of choices; If the workload is executed with periodicity, it can use history to predict	Best of both worlds: runtime information for accurate decision and reduction of overhead due to predictable program characteristics
Disadvantages	Same program can work completely different on, for example, different input sizes; Sometimes we do not know which parts of the code are going to run	High overhead from the runtime monitoring; Sometimes, not clear understanding of the program behavior, causing sometimes predictions not very reliable	The two sources of overhead: runtime monitor overhead plus the instrumentation. Requires a careful combination of static and dynamic decisions.

<i>Work</i>	<i>Level</i>	<i>Source</i>	<i>Auto</i>	<i>Runtime</i>	<i>Learn</i>
[48]	C	Yes	Yes	No	Yes
[43]	L	No	Yes	Yes	Yes
[17]	C	Yes	Yes	No	Yes
[6]	C	Yes	Yes	Yes	No
[52]	C/L	Yes	No	Yes	No
[36]	C/L	Yes	No	Yes	No
[30]	A/L	Yes	No	No	No
[37]	A	No	Yes	No	No
[56]	A	No	Yes	No	No
[44]	O	No	Yes	Yes	Yes
[46]	O	No	Yes	Yes	No
[19]	O	No	Yes	Yes	Yes
[5]	L	Yes	No	No	No
[47]	O/C	Yes	Yes	Yes	No
[55]	O/C	Yes	Yes	Yes	No
[16]	O/C	Yes	Yes	Yes	No
Astro	O/C	Yes	Yes	Yes	Yes

Table 2.1. Comparison between different solutions to . *Level*: at which level the technique is implemented: Architecture (A), Operating System (O), Compiler (C) or Library/Programming model (L). *Code*: “Yes” if there is source code instrumentation/manipulation. *Auto*: “Yes” if it is performed automatically, without user intervention/annotation. *Runtime*: “Yes” if technique considers runtime information. *Learn*: “Yes” if technique adapts/learns a model from the target architecture

2.2 Non-assisted code-placement

The problem of scheduling computations in heterogeneous architectures (Definition 3.0.1) has attracted much attention in recent years, as Mittal and Vetter have thoroughly discussed [40]. Table 2.1 provides a taxonomy of previous solutions to this problem. We group them according to the level (e.g., ../..) at which they are implemented, and to the way they answer each of the following four questions:

Source: is the program’s code modified?

Auto: is user intervention required?

Runtime: is runtime information exploited?

Learn: is there any adaptation to runtime conditions? Perhaps the most important difference among the several strategies proposed to solve concerns the moment at which said strategy is used. In the rest of this section, we consider the following three possible choices: at compilation time, at runtime, or both.

Static Solutions. Purely static approaches work at compilation time. They might be applied by the compiler, either automatically, i.e., without user intervention [16; 29; 36; 52; 48; 55], or not. In the latter case, users can use annotations [39], domain specific programming languages [36; 52] or library calls [5] to indicate where each program part should run. The main benefit of static techniques is low runtime overhead: because every scheduling decision is taken before the program runs, no dynamic checks are necessary to schedule computations. However, these techniques tend to be inflexible: they are unable to take runtime information into consideration; hence, the same program phase is always scheduled in the same way. In Table 2.1, techniques implemented at either the compiler or library levels are purely static.

Dynamic Solutions. Purely dynamic approaches take into account runtime information. They can be implemented at the architecture level [50; 37; 30; 56; 60], or at the virtual machine (VM)/OS level [46; 44; 63; 21; 53; 6]. By leveraging runtime information, the system can use environment information, unknown at compilation time, to solve . Examples of such information include varying input sizes and resource demands. However, there may be some overhead on accurately collecting and processing runtime data. Besides, because scheduling decisions are taken on-the-fly, usually the scheduler cannot spend much time weighting choices. Thus, even though these algorithms use runtime information, they might still take suboptimal decisions, due to their inability to spend much time solving hard scheduling problems.

Hybrid Solutions. Approaches that mix static and dynamic techniques are called *hybrid*. Astro is a hybrid method, and built around the idea of being a customized framework for other hybrid methods. Other hybrid approaches to this problem exist [47; 16; 55]. Piccoli *et al* [47] have used a compiler to instrument a program with guards that determine, based on input sizes, where each loop should run. Cong and Yuan [16], in turn, partition a program in phases, as in our approach, and use runtime information to schedule computation so as to minimize the energy consumed by the program. Finally, Tang *et al.* [55] use a compiler to populate a program code with markers, so that low-priority applications can manage their own contentiousness to ensure the QoS of high-priority co-runners. None of these previous work use any form of learning technique to adapt the program to runtime conditions, as Table 2.1 indicates in the column *Learn*. Once guards are created, they always behave on the same way. That is the main difference between these previous approaches and the Astro method.

Chapter 3

Astro framework

This chapter describes the design and implementation of the Compile assisted Adaptive Code Placement in Heterogeneous Systems, called Astro Framework. This is our approach to solve the problem of finding good hardware configurations for programs. We state this problem as follows:

Definition 3.0.1 SCHEDULING OF PROGRAMS IN HETEROGENEOUS ARCHITECTURES (SPHA)

Input: a program P plus its input I , a suite of hardware configurations H_1, H_2, \dots, H_n , an energy threshold E , and a performance threshold S .

Output: P' , a new version of P , which switches between configurations, and process I using $E\%$ less energy, with a slowdown of no more than $S\%$.

We solve SPHA using an assortment of different techniques, which, once combined, give us the means to generate code that is well adapted to different architectures and workloads. Figure 3.1 provides a general overview of these techniques, emphasizing the different phases over which we go in the process of solving SPHA. Section 3.2.1 describes the phase partitioning. Program instrumentation is necessary to the program partitioning phase. Section 3.2.2 goes over actuation; and Section 3.2.3 discusses the generation of the final program. Before we move into the particulars of our solution to SPHA, we provide a brief introduction to Q-Learning, the flavour of reinforcement learning that we have adopted. In the next section, we also explain how we map the characteristics of SPHA into the Q-learning framework. Before we came up with this approach, we have tested others. For instance, we Supervised Learning solutions in a simplified problem, as seen in Section 4.0.2.1.

3.1 Reinforcement learning via Q-Learning

Q-learning is a reinforcement learning algorithm [54]. Given some notion of state (Definition 3.2.1) and reward (Definition 3.2.4), it finds an optimized policy to perform the best action (Definition 3.2.5). Q-learning is attractive because there is no need to know in advance the precise results of the actions before we perform them; that is, we learn about the environment as we perform actions on it. A Markov Decision Process (MDP) drives Q-learning. A MDP is given by a set of states S , a set of possible actions A , a reward function $R : S \times A \rightarrow R$, and a state transition mapping $T : S \times A \rightarrow S$ that describes the effects of taking each action in each state of the environment. The Markov property states that the results of an action performed in a state will depend only on that particular state, regardless of any other prior states.

In Q-learning, the function $Q(s, a)$ is specified to tell us how good an action a is given that we are in a certain state s . Intuitively, it approximates the best possible sum of rewards from s till the final state, assuming that we perform always optimal choices. Predicting optimal choices from any given state is usually impossible; thus,

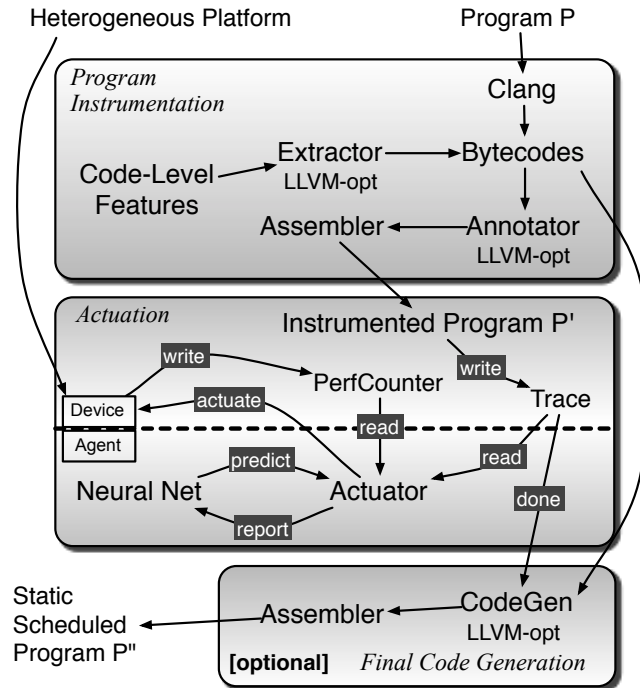


Figure 3.1. The Astro Framework.

we approximate Q with the *Bellman Equation*:

$$Q_{target}(s, a) = R(s, a) + \gamma \operatorname{argmax}_{a'}(Q(s', a')).$$

Here, s' is the state that follows from the application of action a onto s , and a' is the best possible action taken from s' . $R(s, a)$ is the immediate reward computed from the environment. Value $\gamma \in [0, 1]$ represents a balance between prior and immediate reward. At each step in the Q-learning algorithm, we choose the action a that maximizes the function $Q(s, a)$.

Traditionally, the Q-learning algorithm uses a tabular form to represent the $Q(s, a)$ mappings, but this approach cannot scale and generalize well to high dimensional feature spaces. Similarly to recent prior work on reinforcement learning [41], we use a neural network to approximate the $Q(s, a)$ function used to compute the expected reward from a given state s and action a .

To find a good approximation of $Q(s, a)$, we use a neural network (NN), to track its current value. Thus, "learning" means actually updating the NN. At each iteration of our learning process, we update the NN using Stochastic Gradient Descent (SGD) [10] over the squared error over the target values (via Bellman equation) and the values obtained by our network:

$$\sum (Q_{target} - Q)^2.$$

3.1.0.1 Motivation for the model used

Motivation to a reinforcement learning model When building a supervised solution for an application in the PARSEC benchmark such as fluidanimate, we took roughly a whole day of execution. We could retrieve the labels for its phases by exhaustively executing the application in the 24 different possible combinations of big and little cores in the ARM Big.LITTLE board. Increasing this action allowing five frequency changes for every configuration in a DVFS setup, we would have $24^5 = 7,962,624$ configurations to evaluate. Furthermore, we are assuming that the placement for one phase does not affect the other, what is not true. Therefore, if we expand the problem space or break the assumption of phases don't affect each other or even consider a multiple application scenario, it is clear unfeasible to work with the traditional supervised learning approach. We could try to work around this problem trying to restrict our space as [43], keep the assumptions, and maybe use heuristics to extrapolate from a learned space to the actual problem space, but we have recognized that reinforcement learning solution fits to the problem more naturally.

Influences to the model used The prediction model design (definition of state, action and reward function) of Astro was based on three main approaches. They are [18], [31], [61] and [57].

The [18] handles a thermal optimization, it has a multi-objective function which tries to reduce the thermal dissipation and maximize performance. On [31], has a similar problem which solves via reinforcement learning and considers temperature and performance as a multi-objective reward function. However, its solution is based in the manipulation of idle periods, not our focus. Some ideas, such as the combination of multi-objective functions in a reward function, and the shape of the function (with a restriction applied to the function as a subtraction to the original value) came from these articles. Other than these three main articles, [57] uses reinforcement learning and has a dynamic approach trying to formalize the problem of allocating (and migrating) resources in a heterogeneous multi-core architecture. The reward function is given in terms of the CPU and memory usage. The reward function is a single-objective, trying to reduce only the waiting time.

Related to reinforcement learning using neural networks, it had a great impact on [41]. In this article, neural networks extends the use of a q-table on their solution. However, the problem and the motivation for the use of a neural network are different. In the problem approached by the article, there is an inherent representational complexity in the problem (they did not know, without simplifying too much, how to model the states of the games). A state, thus, was represented as images of the game, while in this dissertation we assume we know how to represent our environment given the dynamic and static characteristics we have chosen. Other than that, the article presents an agent that could play several the games well aiming an agent which can play well a class of problems really well. In our case, we specify a single problem, the code placement on heterogeneous architectures minimizing energy with performance restrictions.

3.2 Components of the Astro framework

3.2.1 Part 1: Phase Partitioning

A running program might cause the hardware to go over an infinite number of different states. Because this universe is unbounded, Definition 3.2.1 discretizes the notion of a *State*. In that definition, S is a *Program Phase* and D is a *Hardware Phase*. Program phases are discussed in Section 3.2.1.1, and hardware phases are discussed in Section 3.2.1.2.

Definition 3.2.1 (State) *A state is a triple $\langle H, S, D \rangle$ representing a hardware configuration H , a program phase S and a hardware phase D .*

3.2.1.1 Program Phases

Static Program Phases depend only on the syntax of a program. Definition 3.2.2 formalizes this notion. We emphasize that a static program phase is not equivalent to a *program region*, because different regions can present the same set of feature ranges. Example 3.2.1.1 clarifies the meaning of these definitions.

Definition 3.2.2 (Program Phase) *A code-level feature (also called code feature or simply feature) is a syntactic characteristic of a program, such as number of n -nested loops or instruction mix. A feature range is a contiguous interval of values that a feature can assume, and that partitions the feature space into equivalence classes. A program phase S is a group of feature ranges, covering different features.*

The density of arithmetic and logical instructions is a code-level feature, which we obtain by dividing the number of such opcodes by the total number of program instructions. We can define different feature ranges covering this metric, such as $[0, 0.25)$, $[0.25, 0.50)$ and $[0.5, 1.00]$. The highest number of nested loops in a program yields another feature. In this case, possible ranges are $[0, 1]$, $[2, 3]$ and $[4, +\infty]$. Finally, an expectation on the number of I/O routines called in a function gives us a third feature. A possible way to calculate it is: $\sum_i 10^n$, for every I/O call i nested into n loops. We can, again, define intervals for this metric, e.g., $[0, 1)$, $[1, 10)$ and $[10, 100)$ and $[100, +\infty]$. The $3 \times 3 \times 4$ possible combinations of such feature ranges gives us 36 program phases. If we collect these features for each function in the program code, then we can map any of them to one of these program phases.

Being a syntactic characteristic of a region within a program text, we can mine program phases using well known parsing techniques. The text that ends up executed is the binary representation of a program, often written in a high-level language. Thus, ideally parsing should be performed in that binary representation. Recovering high-level structure from binary programs involves a number of undecidable problems [27]. To circumvent this shortcoming, we recommend mining features from the intermediate program representation that the compiler manipulates before producing executable code. In this work, we have implemented a *Phase-Extractor* using the LLVM compiler, via its `opt` bytecode analyzer.

The result of mining program features is a map that assigns phases to program regions. This map depends on the choice of program region. Many different granularities

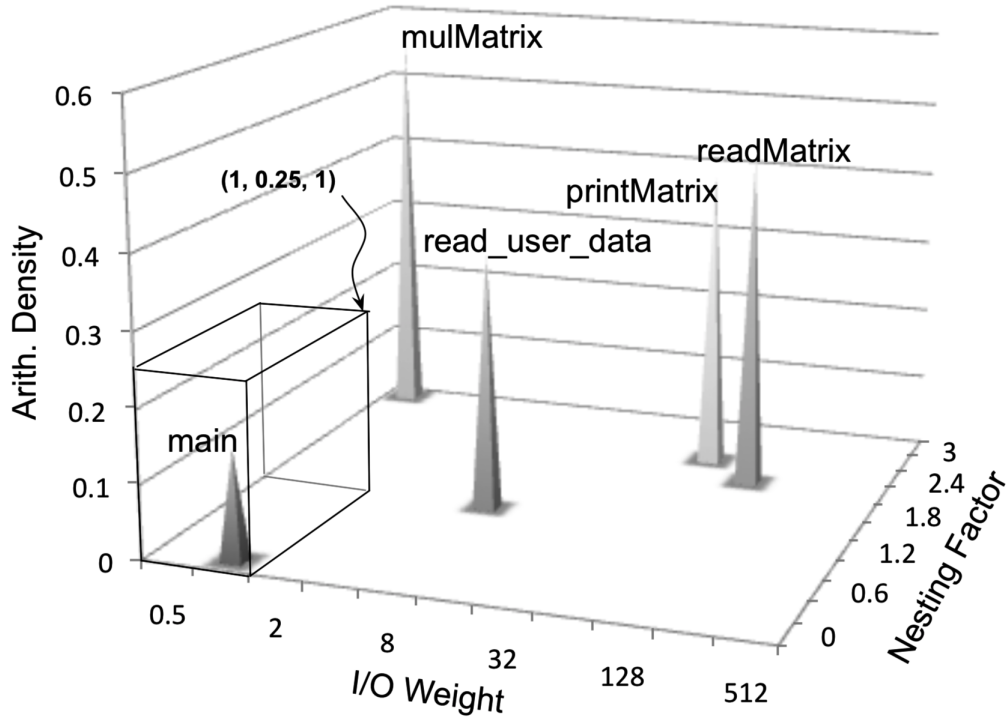


Figure 3.2. Mapping the functions in Figure 1.2 (a) to program phases.

of regions are possible, such as instruction, basic block, loop, Single-Entry-Single-Exit block [20], etc. We have chosen to work mostly at the granularity of functions. The “mostly” in this case, refers to the fact that we also change phases before and after library calls that cause the program to block waiting for some event (see the Barrier phase, in the discussion that follows). Pragmatically, this means that the instrumented program adds logic to change phases at the entry point of functions, and around certain library calls.

Figure 3.2 shows the five functions in Figure 1.2, classified according to features seen in Example 3.2.1.1. We are assigning these functions hypothetical values. Because we have three features, we can map them into a three-dimensional space. Each phase corresponds to a cube in this space. Figure 3.2 shows the sub-space that corresponds to the phase: $\text{Arith.Density} \in [0, 0.25)$, $\text{I/O Weight} \in [0, 1)$ and $\text{NestingFactor} \in [0, 1)$. Function `main`, in our example, fits in this phase.

Discrete Choice of Program Phases. In our discrete implementation, we combine four code features to determine program phases. These features are all “densities”, i.e., they represent a certain quantity of instructions normalized by the total of instructions

in the target function. The features that we have used are listed below:

IO-Dens: proportion of library calls that perform I/O operations;

Mem-Dens: proportion of instructions that access memory (loads and stores);

Int-Dens: proportion of arithmetic and logic instructions that operate on integer types.

FP-Dens: proportion of arithmetic and logic instructions that operate on floating point types.

Locks-Dens: proportion of lock instructions.

Barrier: true if the program has invoked a multi-thread barrier library call that forces it to wait for some blocking event.

Network: true if the program has invoked a networking library call that forces it to wait.

Sleep: true if the program has invoked a sleep library call that forces it to wait.

When handling with our discrete variables, we have defined four program phases, which appear as combinations of the features above. This choice is arbitrary. There exists an infinite number of different possibilities to partition programs in phases. We have opted for a simple partitioning, involving only a handful of features for convenience, as this choice already lets us support the main thesis of this dissertation: that static features greatly enhance the dynamic scheduling of computations in heterogeneous hardware. A more extensive search for good program phases is, in itself, an interesting project, but lays outside the scope of this work. The program phases that we shall consider in the experiments using the simulator in Section 4 are:

Sync: $\text{Barrier} = \text{true}$ or $\text{Sleep} = \text{true}$ or $\text{Network} = \text{true}$ or $\text{Locks}/\text{total} > 0.01$

I/O Bound: $\text{IO-Dens} + \text{Mem-Dens} > 0.5$
and $\text{Sync} = \text{false}$;

CPU Bound: $\text{Int-Dens} + \text{FP-Dens} > 0.5$
and $\text{Sync} = \text{false}$ and $\text{I/O Bound} = \text{false}$

Other: in case none of the previous relations hold.

Continuous Choice of Program Phases. When using a deep reinforcement learning, each static feature collected corresponds to a neuron in the input layer. For the main our experiment, on Table 4.1, we use this continuous approach with no combination of variables.

3.2.1.2 Hardware Phases

Whereas the program phases seen in Section 3.2.1.1 depend only on syntactic program characteristics, *hardware phases* depend on the dynamic state of the hardware. We define the notion of hardware phase as follows:

Definition 3.2.3 (Hardware Phase) *A Performance Counter is any monitor that collects dynamic information about the hardware state, such as CPU performance and cache miss rate. The domain over which the performance counter ranges can be partitioned into phases. Given a collection of performance counters $\{C_1, C_2, \dots, C_n\}$, where each C_i is partitioned into R_i phases, then a hardware phase is any combination within the product $R_1 \times R_2 \times \dots \times R_n$.*

The monitoring of hardware phases do not require program instrumentation. Instead, an *actuator* reads the state of the hardware performance counters directly and periodically. Modern computer architectures already provide an array of performance counters that can be queried. When such is not the case, it is still possible to approximate hardware phases in software. As an example, Walker *et al.* [59] describe how to estimate current levels of power dissipation with high reliability using performance counters.

Discrete Choice of Hardware Phases. We consider four kinds of dynamic information in order to define hardware phases. We show, next to each dynamic feature its discretization into ranges we used in our initial experiments:

IPC: instructions per cycle in the ranges $[0, .5)$, $[.5, 1.0)$, $[1.0, +\infty)$;

CMA: cache misses per cache accesses in the ranges $[0, 1\%)$, $[1\%, 5\%)$, $[5\%, +\infty)$;

CMI: cache misses per instruction executed, in the ranges $[0, .1\%)$, $[.1\%, .5\%)$, $[.5\%, +\infty)$;

CPU: utilization of the CPU, in the ranges $[0, 20\%)$, $[20\%, 50\%)$, $[50\%, +\infty)$. Each of these counters is partitioned in three buckets. Therefore, we consider a total of $3 \times 3 \times 3 \times 3 = 81$ different hardware phases.

Continuous Choice of Hardware Phases. The continuous consider as hardware phases the hardware features plus one feature being the configuration which the hardware (dynamic) features were collected. When using a deep reinforcement learning, each hardware feature corresponds to a neuron in the input layer.

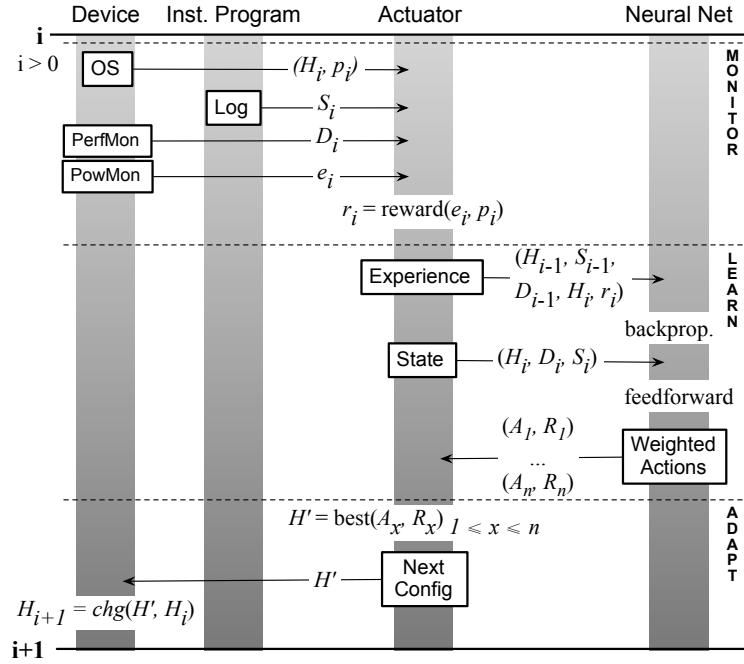


Figure 3.3. The Actuation Algorithm.

3.2.2 Part 2: Actuation

The heart of the Astro system is the Actuation Algorithm outlined in Figure 3.3. Actuation consists of *phase monitoring*, *learning* and *adaptation*. These three steps happens at regular intervals, called *check points*, which, in Figure 3.3, we denote by i and $i+1$. The rest of this section describes these events.

3.2.2.1 Monitoring

To collect information that will be later used to solve SPHA, Astro reads four kinds of data. Figure 3.3 highlights this data:

From the Operating System (OS): current hardware configuration H and instructions p executed since last check point.

From the Program (Log): the current program phase S .

From the device's performance counters (PerfMon): the current hardware phase D .

From the power monitor (Powmon, [59]): the energy e consumed since the last check-point. The monitor collects this data at periodic intervals, whose granularity is configurable. Currently, it is 500 milliseconds. The recording of the current program

phase is aperiodic, being carried out by instrumentation inserted in the program by the compiler. As discussed in Section 3.2.1.1, information is logged at the entry point of functions, and immediately before and after the invocation of library calls that might cause the program to enter a dormant state. The hardware configuration is updated whenever it changes. The metrics e and p lets us define the notion of *reward* as follows:

Definition 3.2.4 (Reward) *The reward is the set of observable events that determine how well the learning algorithm is adapting to the environment. The reward is computed from a pair (e, p) , formed by the Energy Consumption Level e , measured in Joules per second (Watt), and the CPU Performance Level p , measured in number of instructions executed per second.*

The metric used in the reward is given by a weighted form of performance per watt, namely $MIPS^\gamma/Watt$, where γ is a design parameter that gives a boosting performance effect in the system. This is usually a trade-off between the performance and energy consumption. To optimize a metric of performance per watt we can set $\gamma = 1.0$. We find that a value of $\gamma = 2.0$ is more suitable to al low emphasis on performance gains: the reward function optimizes (in fact, maximizes the inverse of) the energy delay product per instruction, given by $Watt/IPS^2$; letting $IPS = I/S$ we have $(Watt \times S \times S)/I^2 = (Energy \times Delay)/I^2$. This aims to minimize both the energy and the amount of time required to execute thread instructions [12].

Optimization. Logging is a costly operation, because it involves writing information in memory that the program shares with the Monitor. To reduce this overhead, we use a threshold to separate “short” from “long” functions. Only the so called long-running functions are instrumented. We say that a function is long if its size, measured in terms of weighted instructions, surpasses a certain threshold T . The weight w of an instruction ι , referred to as $w(\iota)$, is given by $w(\iota) = 10^n$, where n is the number of nested loops surrounding ι . Currently, we let $T = 1,000$. As we shall see in Section 4, this threshold leads to a substantially reduced overhead.

Continuing with Example 3.2.1.1, Figure 3.4 (a) shows the instrumentation of function `main` (Figure 1.2) to log program phases while (b) shows the final binary.

3.2.2.2 Learning

The learning phase uses the methodology discussed in Section 3.1. As illustrated in Figure 3.3, a key component in this process is a multi-layer Neural Network (NN) that receives an experience input collected from the Monitor. The NN outputs the actions

```

int main(int argc, char** argv) {
  save_feature_ranges (
    0.12, /* Arithmetic Density */
    0.8, /* IO weight */
    0, /* Nesting factor */
    False /* Sleeping state */);
  // Read first matrix from file 'argv[1]'
  int** m1 = readMatrix(argv[1],&M1,&N1);
  toggle_sleeping_state (
    True /* Known blocking function */);
  read_user_data();
  toggle_sleeping_state (
    False /* Back into activity */);
  // Read second matrix from file 'argv[1]'
  ... same as original figure.
}

```

(a)

```

int main(int argc, char** argv) {
  /* Conf == 1 is 0L1B */
  determine_active_configuration (1);
  // Read first matrix from file 'argv[1]'
  int** m1 = readMatrix(argv[1],&M1,&N1);
  /* Conf == 0 is 1L0B */
  determine_active_configuration (0);
  read_user_data();
  /* Conf == 1 is 0L1B */
  determine_active_configuration (1);
  // Read second matrix from file 'argv[1]'
  ... same as original figure.
}

```

(b)

Figure 3.4. (a) Instrumentation to mine features. (b) Final instrumentation, inserted in production code.

and their respective rewards to the Actuator so that a new system adaptation can be carried out. To ease our explanation, we split learning in two steps: *back-propagation* and *feed-forwarding*, as outlined in Figure 3.3.

Back-Propagation. In this step, we perform an update to the NN using the experience data given by the Actuator (Figure 3.3). The experience data correspond to a triple of present state, performed action and obtained reward. The state consists of a hardware configuration (H_{i-1}), static features (S_{i-1}) and dynamic features (D_{i-1}) at check points $i-1$. The action performed at check point $i-1$ makes the system move from hardware configuration H_{i-1} to H_i . The reward is given by r_i , received after the action is taken. The NN consists of a number of layers including computational nodes, i.e., neurons. The input layer uses one neuron to characterize each triple (*state, action, reward*). The output layer has one neuron per action/configuration available in the system. We use the method of gradient descent to minimize a loss function given by the mean squared error between Q_{target} and $Q_{predicted}$, where Q_{target} is computed using the *Bellman Equation* (Section 3.1) and $Q_{predicted}$ is computed by the NN.

Feed-Forward. In this step, we perform predictions using the previously trained

NN to derive the best course of actions in the system. Within the NN, each node is responsible to accumulate the product of its associated weights and inputs. When the resulting value is above a given threshold, the corresponding neuron fires and induces an activation; otherwise, it is deactivated. This step is less time consuming than back-propagation, because information only moves from the input layer of the NN to the last layer. Given as input a state (H_i, D_i, S_i) at check point i , the result of the feed-forward step is an array of pairs $A \times R$, where A is an *action*, and R is its *corresponding reward*, as estimated by NN. Actions determine configuration changes; rewards determine the expected performance gain, in terms of energy and time, that we are likely to obtain with this change.

3.2.2.3 Adaptation

At this phase, Astro takes an *action*. Together with states and rewards, actions are one of the three core notions in Q-learning. We define it below, in the context of the Astro system:

Definition 3.2.5 (Action) *Action is the act of choosing the next hardware configuration H to be adopted at a given checkpoint.*

An action potentially changes the current hardware configuration; hence, adapting the program according to the knowledge inferred by the Neural Network. Following Figure 3.3, we start this step by choosing, among the pairs $\{(A_1, R_1), \dots, (A_n, R_n)\}$, the action A_x associated with the maximal reward R_x . A_x determines, uniquely, a hardware configuration H' . Once H' is chosen, we proceed to it. However, the adoption of a configuration is contingent on said configuration being available. Cores might not be available because they are running higher privilege jobs, for instance. Astro tries to enable Next Configuration. If nothing changes, in the next monitoring time Astro will notice and the system representation remains in the configuration H_i active at check point i . Such choice is represented, in Figure 3.3, by the function $H_{i+1} = chg(H', H_i)$. Regardless of this outcome, we move on to the next check point, and to a new actuation round.

3.2.3 Part 3: Final Code Generation

Astro can impose a heavy burden on the runtime system due to the overhead of monitoring and training the NN. This last task – learning – is the most time consuming and can be offloaded; hence, running outside the device. However, if, on the one hand,

offloading removes the cost of executing a neural network in the device, it adds a communication cost, which is also non-negligible. It is possible to reduce this overhead, if we are willing to reduce the program’s ability to adapt to the environment. With this goal, we present a form of code generation that can be used alongside Astro. After we have “trained” a program to a given architecture, we imprint an approximation of this knowledge directly in that program’s code. In Figure 3.1, this step is named *Final Code Generation*.

Code generation consists in inserting instrumentation into the target program. Instrumentation is inserted in the same regions modified in order to mark program phases (see Section 3.2.1.1): at the entry point of functions, and around particular library calls. Example 3.2.3 illustrates this instrumentation. We notice that the same optimizations discussed in Section 3.2.2.1 apply to the final program. Thus, we avoid annotating functions that are too short, since they are unlikely to cause large variations in the program’s runtime behavior.

Figure 3.4 shows the final actuation code for the program early seen in Figure 1.2. Function `determine_active_configuration` tries to move the program to a particular configuration: the one that has produced the largest rewards for that program phase.

The code that we generate in this way performs a form of *static scheduling*. It always maps the same program region to the same hardware configuration. As we show in Section 4, static scheduling yields lower runtime overhead than Astro’s dynamic scheduling. However, as we have observed empirically, even with the runtime overhead, Astro’s extra flexibility usually pays off in terms of energy and performance.

Astro Static and Hybrid deployments The code generation can be in two ways. We call them the: Static Deployment and Hybrid Deployment. In the Static Deployment, after Astro is trained, we can determine, via instrumentation, the exact actions that must be taken in the source code. The Static Deployment is not adaptive. In the Hybrid Deploy, once trained, we retrieve static and dynamic information, consult our weights and then take an action.

3.3 Relationship between Astro and the most related work

Among the discussed related works in the previous chapter, the ones we believe are closer related to the work proposed in this dissertation are: Etino ([48]), DeepTune

([17]), Caloree ([43]), Octopus-Man ([46]) and Hipster ([44]). Therefore, we discuss similarities and differences of them and the proposed solution.

Etino [48] introduces a tool which annotates C programs with OpenACC or OpenMP 4.0 directives. It solves the code placement problem for a CPU-GPU heterogeneous setup. The work has interesting insights such having some context sensitiveness, achieved via the analysis of the call graph of the program. Regarding their cost model, they use a machine learning, specifically simulated annealing, to calibrate the cost model for a given heterogeneous architecture. Their work is a successful example of a static approach. However, besides their contributions, it has a few issues that we tried to address on Astro. Firstly, their method is not adaptive after the calibration and they suffer from not having the input size information. These are addressed on Astro via the extraction of the dynamic information online. Another limitation is that it cannot generate code that runs on multiple processors concurrently. We address it on Astro by dynamically changing the multi-threaded code affinity. Finally, Astro remains open the question from [48]: is a function a good granularity choice for a static code placement in heterogeneous architectures? Even though Astro also takes a few code sections that are not functions, such as the lines before a network call or a barrier, most of the instrumented code sections are functions and they usually work fine. As an example, "parallel for" OpenMP loop annotations are mapped to the LLVM IR as internal OpenMP functions. So, instrumenting by function, we instrument all these OpenMP parallel loops.

DeepTune [17] introduces a tool for automatically constructing optimization heuristics without features. It builds a predictor with the raw source code as input. Their predictor was able to efficiently allocate resources for many applications in well-known benchmarks in a heterogeneous system. Specifically, they could predict if a computation could be mapped to a CPU or a GPU. In some way, it motivates the use of a neural network in this field. Furthermore, it brings an interesting question of using or not static features (which are heuristically chosen). Deciding on the set of static features to be used in a predictor can be harmful and prone to errors. It also usually requires much effort from experts and computation resources to trials. Their results were better than one of the state-of-the-art methods which use static features from experts. However, the work also has a limited scope comparing to the current dissertation, the results are harder to explain due to the inherent black-box way of predicting from the Neural Networks. Besides that, the learning model was built in a supervised way and in a binary scenario. In our scenario, a fully supervised learning is no feasible

for our full problem which joins the space of combinations of cores in a heterogeneous multi-processor board plus dvfs space is much greater than the binary scenario.

Caloree [43] introduces a control system which aims to meet latency requirements with minimal energy in complex, dynamic environments. It is composed by a junction of a control system with a learning framework. It breaks the resource allocation into two sub-tasks: learning how interacting resources affect speedup and controlling speedup to meet latency requirements with minimal energy. According to [43], it was used on heterogeneous ARM big.LITTLE architectures in both single and multi-application scenarios improving performance and reducing energy consumption. The work brings interesting insights. As examples, the exploration of DVFS along with core migrations, which lead us to focus more on the DVFS optimization space. Besides that, the careful try to explain the allocation choices based on the program behaviors lead us to follow the same path of being carefully on explaining and debugging allocation results. However, the work has a few limitations compared with this dissertation, specifically the restriction to the only homogeneous type of cores usage and the limit to at maximum four of the eight available cores.

Octopus-Man [46] introduces a tool which schedules tasks in an heterogeneous hardware. It works applying QoS (quality of service) restrictions while minimizing the energy consumed. However, its target applications are in the cloud such as Memcache and Websearch, while the focus of this dissertation is broader. Moreover, Astro requires the source code, what is not necessary for this approach.

Hipster [44] introduces a tool which is the closest to our proposed research. It also has an approach to heterogeneous architectures, specifically, heterogeneous multi-processors. It can be considered an evolution of the Octopus-man which joins its heuristic approach to a reinforcement learning technique. However, the algorithm is still in the context of cloud applications. Due to the fact they are cloud applications, the idea of QoS restrictions has a more clear definition. In a general application, we would need to translate a QoS restriction (that is in terms of requisitions per second in a web search for example) to latency or percentage of the application completed. Another difference is that Hipster does not use any static feature. Other than that, we use more dynamic features than Hipster by using the PMUs. Finally, the Hipster model is not only driven by the reinforcement learning, but also by a heuristic (the Octopus-man heuristic), in order to avoid instabilities. This combination of learning plus heuristics was called Hybrid on [44].

Chapter 4

Evaluation

This chapter presents an experimental evaluation of the Astro system over several benchmarks running on a heterogeneous multi-core system. In the process of evaluating Astro, we shall provide answers to the following research questions:

RQ1: How close can Astro be from an optimal oracle?

RQ2: How does Astro compare against fixed and immutable best configuration choices?

RQ3: How does Astro perform when compared with state-of-the-art program schedulers?

RQ4: Is the combination of static and dynamic information really necessary? How does Astro perform when any of this data is missing?

RQ5: How does Astro behave on an actual device?

RQ6: What is the runtime overhead of the Astro system?

4.0.1 Experimental Setup

We shall be using two experimental setups: the first consists of experiments performed on an actual device. The other consists on experiments performed on *program traces*, which we shall call *simulation*. We had to use simulation in some of the experiments, because they involve testing exhaustively every available hardware configuration. These experiments are described in Section 4.0.2. Section 4.0.3 reports the experiments performed on the actual device: the Odroid XU4 development board with a big.LITTLE ARM processor (Samsung Exynos 5422) featuring 4 big cores (Cortex-A15 2.0 Ghz) and 4 LITTLE cores (Cortex-A7 1.4 Ghz). This device has also been used to produce

the traces used for simulation. We report CPU power consumption via [59]. Performance monitoring happens at the OS level. Following common practice [9; 64], we use IPC (Instructions per Cycle) as a metric of computational load and LLC (Last Level Cache) miss rate to characterize memory behavior.

We have tested our technique in applications taken from Parsec [8] (2 apps) and Rodinia [15] (6 apps) benchmarks. Regarding the Parsec collection, we could not compile six of those missing to the XU4 board, even without Astro’s instrumentation. Regarding the rodinia collection, `mummergpu` wasn’t a fit, as it targets GPUs, `kmeans` has problems on compiling in C++ (needed to compile in C++ because of the current Astro C++ library), as it is originally in C, and `lud` had too small inputs available. The other applications could run successfully. We also used Meabo (micro-benchmark from ARM) [ARM-software].

4.0.2 Results in the Simulated Environment

The experiments that we report in this section are difficult to carry out on an actual device, because reproducing them would be very challenging and time consuming. For instance, how to approximate an optimal oracle on a hard device? At each checkpoint we would need to be able to choose the configuration that yields the best payoff; however, that would require us either peering into future events; or rolling back to test unchosen configurations. To circumvent such difficulties, we have approximated the exhaustive execution of configurations by generating traces for every hardware configuration. These traces lets us simulate different behaviors, by choosing, at each checkpoint, the reward offered by one of them. Different policies can guide this choice: optimal, best fixed and random for instance. Producing such traces is very time consuming, thus, we have produced them only for the Parsec benchmarks seen in Figure 1.4. In the rest of this section, we report results only for the longest trace, produced for `fluidanimate`, whose execution time has varied from 410 (~ 7 min) seconds to up to 7,000 (~ 2 h), depending on the configuration, when initialized with its standard input. We omit the other traces because they lead to similar conclusions.

The Figure 4.1 presents an approximated optimal policy, also called an Oracle. This Oracle is composed by the best set of configurations for every execution time slot. The objective functional for this Oracle is energy minimization. The Oracle was produced using our simulator for the trace of one execution of the `fluidanimate` application. We used the native input size for the application. This image seems to support our claim that the problem is slight complex. The expected application execution (oracle policy), migrates to 13 different configurations during its execution.

Figure 4.2 compares seven different scheduling strategies built on top of this simulator, applied on fluidanimate.

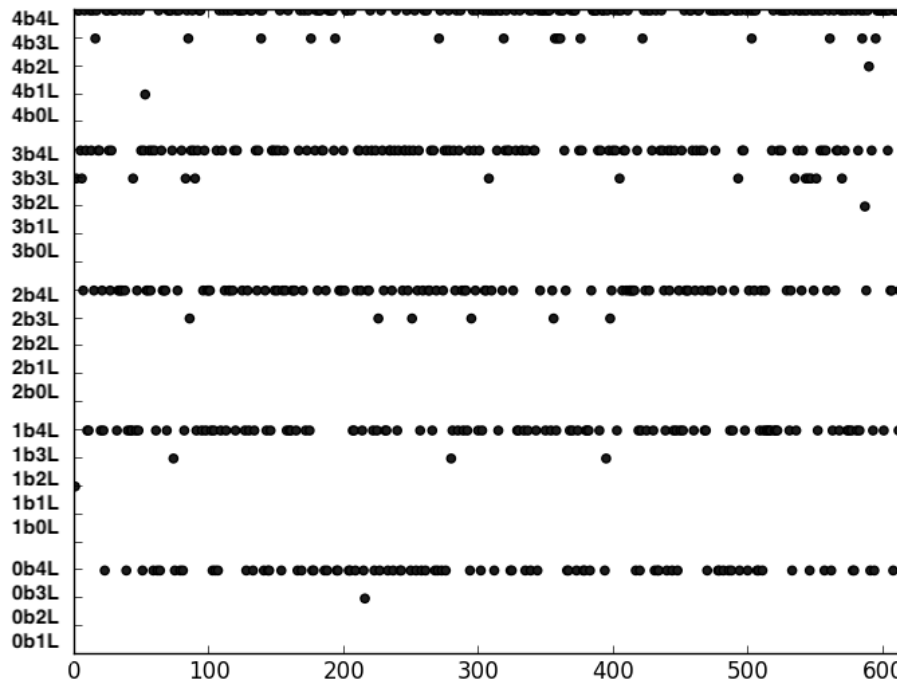


Figure 4.1. Simulated approximated optimal policy for fluidanimate. The Y-axis represents different hardware configurations. The X-axis shows the application's execution time in seconds.

4.0.2.1 Pre-Astro: Applying Supervised Learning

Our final goal was a learning algorithm which receiving online feedback adapts to a given program with the help of static and dynamic information. In order to achieve it, we went by steps. In a first moment, the most natural way to learn which is the best configuration would be receiving help from an external agent. This agent knows the best configuration for a few code sections which we can base our learning. This is called supervised learning technique. For our problem, it is a simpler approach and easier to converge. However, it is time consuming creating this oracle agent.

Our general method to apply Supervised Learning in the system was the following:

1. Identify code sections via static analysis.

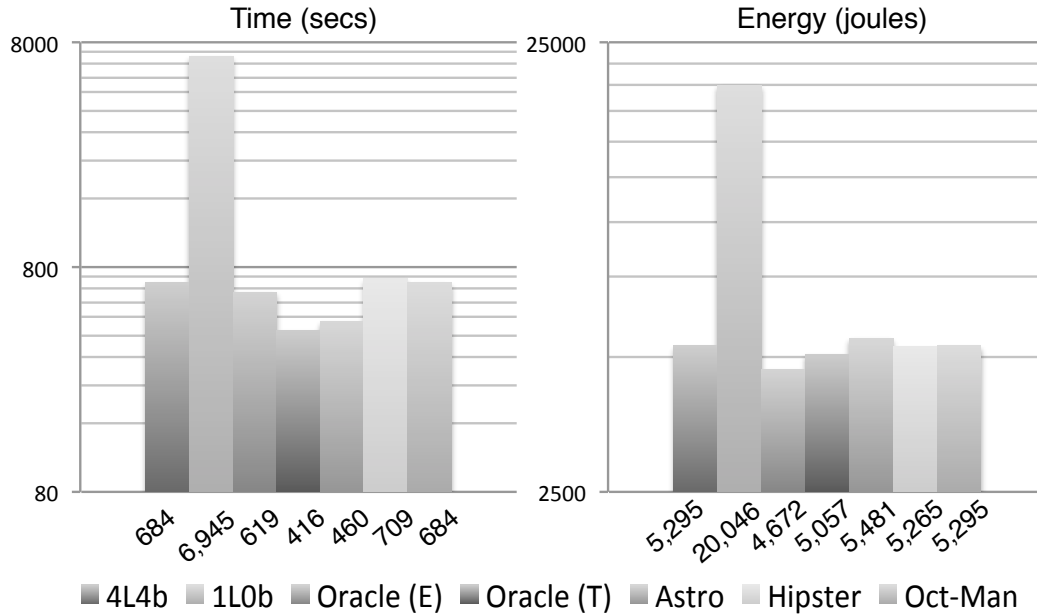


Figure 4.2. Comparison between Astro and other 6 strategies.

2. Once identified code sections, instrument each of them in the beginning and in the end.
3. Start power and performance measurements monitoring sampling and logging results with timestamps
4. Execute the program for every possible configuration we are considering

In the end of the process we must have the following information for a given code section:

- Static analysis information of the code section.
- Two timestamps $[s, e]$, being ' s ' the timestamp of the beginning of the section and ' e ' the timestamp in the end of it.
- The performance measure units of the program during the section and CPU usage per core.
- The amount of energy spent per cluster in the section.

A learning unit is a vector $v = [S_0, \dots, S_N, H, D_0, \dots, D_N]$, being S, H and D, static, configuration and dynamic information respectively. They represent a multi-dimensional point in the multi-dimensional feature space of our learning problem. On every learning unit, we aim to discover a label in a particular moment. We call a

label, the optimal hardware configuration for the code section given a function, that can be, for example, the most energy efficient configuration within at maximum 5% of performance degradation. So, given the metric, you must retrieve all the labels for the code sections.

Once defined the Oracle for the code sections. We can build a learning unit of a code section, as a vector [Static Information, Current Configuration, Dynamic Information] and a label corresponding to the best configuration (action). We define a learning model and split the code units we have in training and test sets. We use the learning sets to train our learning model and use the test set to evaluate our learning model.

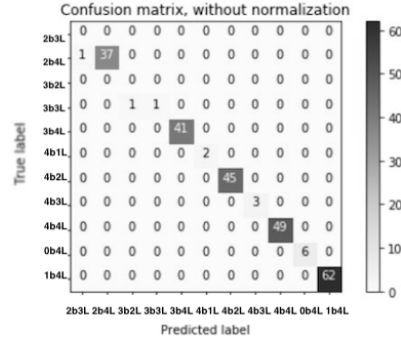
Results In order to evaluate the supervised approach, we did two experiments. The first one explores the DVFS scenario on a couple applications in the Rodinia benchmark [15] and Meabo [ARM-software], a multi-purpose multi-phase micro-benchmark from ARM. The second experiment focused on a few hot functions from a couple applications in the PARSEC benchmark. We have tested 5 classification learning models multi-class for the tests. The dataset, composed of the vectors of information, were separated as a training and test set. The test set corresponds to 40% of the original input, being the other 60% used in the training of the data. For the second experiment, we evaluate different code sections inside one application, the Fluidanimate of PARSEC benchmark using the native input. Furthermore, we restrict the optimization space to those which appears at least one time in the training or test set (it did not seem reasonable apply to a learning method for a class which the algorithm has never seen).

Thus, from the five algorithms used, three of them performed well the task. They were the Gaussian Process Classifier, the multi-layer perceptron (MPL) and the kNN. The best among them was the GPC followed by the kNN. We believe that the amount of data was small to the neural network leading to a smaller precision. The kNN also has got pretty close to the GPC, loosing only in some classifications of the configuration 2b4L. At the GPC, all the following metrics were much higher than the expected: precision (our metric to indicate exactness), recall (our metric to indicate completeness) and f1-score (harmonic average of precision and recall). The Hamming loss was lower as well, means that the set test and true set are pretty similar. In order to evaluate trade-offs of the problem, two algorithms which did not perform well the task were the Naive Bayes with a Bernoulli distribution a priori and the SVC classifier. We believe it warns us to cautiously choose the a priori distribution which fits with the data.

The confusion matrix, also called error matrix, details the differences between

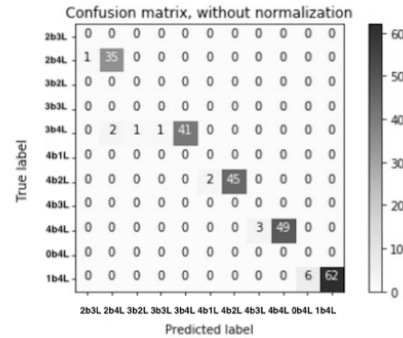
GaussianProcessClassifier : 0.987903225806
 Jaccard similarity: 0.987903225806
 Hamming loss: 0.0120967741935

	precision	recall	f1-score	support
13	0.00	0.00	0.00	1
14	0.97	0.97	0.97	37
17	0.00	0.00	0.00	1
18	0.50	1.00	0.67	1
19	0.98	1.00	0.99	41
21	1.00	1.00	1.00	2
22	1.00	1.00	1.00	45
23	1.00	1.00	1.00	3
24	1.00	1.00	1.00	49
4	1.00	1.00	1.00	6
9	1.00	1.00	1.00	62
avg / total	0.98	0.99	0.98	248



Neural Net : 0.935483870968
 Jaccard similarity: 0.935483870968
 Hamming loss: 0.0645161290323

	precision	recall	f1-score	support
13	0.00	0.00	0.00	1
14	0.97	0.95	0.96	37
17	0.00	0.00	0.00	1
18	0.00	0.00	0.00	1
19	0.91	1.00	0.95	41
21	0.00	0.00	0.00	2
22	0.96	1.00	0.98	45
23	0.00	0.00	0.00	3
24	0.94	1.00	0.97	49
4	0.00	0.00	0.00	6
9	0.91	1.00	0.95	62
avg / total	0.88	0.94	0.91	248



Nearest Neighbors : 0.95564516129
 Jaccard similarity: 0.95564516129
 Hamming loss: 0.0443548387097

	precision	recall	f1-score	support
13	0.00	0.00	0.00	1
14	0.97	0.86	0.91	37
17	0.00	0.00	0.00	1
18	0.00	0.00	0.00	1
19	0.85	1.00	0.92	41
21	0.00	0.00	0.00	2
22	0.96	1.00	0.98	45
23	1.00	1.00	1.00	3
24	1.00	1.00	1.00	49
4	1.00	0.83	0.91	6
9	0.98	1.00	0.99	62
avg / total	0.94	0.96	0.95	248

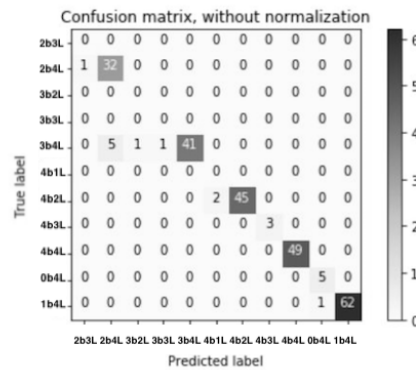


Figure 4.3. Supervised Learning applied to the fluidanimate application from PARSEC benchmark in an action space of 24 possible combinations of heterogeneous cores. Three learning models that achieved the best results of the five tested: Gaussian Process Classifier, the Multi-layer Perceptron (MPL), the kNN.

predicted and expected labels. In our matrix, the rows represents the expected configurations and the columns represents the predicted ones. The numbers inside the matrix represents how many times that expected configuration (row) was labelled as the predicted configuration (column). The principal diagonal (left top to bottom right) shows how many times we matched it right. The matrix is a good way to see which errors our system has done. For example, for the first matrix on Figure 4.3, at the second row and first column, it was expected 2b4L and the system predicted 2b3L. Also, fourth row and third column, it was expected 3b3L and our system predicted 3b2L.

Figure 4.4 shows how the static features can leverage good predictions. We had 39 possible actions (combination of different cores and frequency changes). Using only static features we could achieve a 100% of precision. Analyzing subsets of the features, we saw that using only four static features, a subset of the 16 static features initially used, we could achieve the same 100%. The four features were: number of IO, number of math ops, number of MEM operations, number of INT operations. These variables values are weighted by the number of loops. Among them, the number of INT operations seems to have most influence in the high accuracy, achieving alone around 94% accuracy by itself.

RQ1: how close is Astro to an optimal oracle? The data collected for every possible configurations lets us know, for each part of the program, which configuration consumes less energy and has the best performance. We then combine these 24 traces into a single trace, choosing, at each check point, a particular configuration. This "optimal" trace is what we call the *Oracle*. Notice that our oracle does not provide guarantees of finding a global optimal solution to . Rather, it is a greedy approximation: given that at check-point i we are at configuration H_i , what is the configuration that gives us the best reward at check-point $i + 1$.

Figure 4.2 shows two oracles: (E) and (T). The former yields optimal energy consumption; the latter yields optimal execution time. Astro's reward function prioritizes time over energy; hence, it leads to execution times close to T. If we schedule Fluidanimate with Astro, its final runtime is only 10% slower than T. However, it is more energy hungry: it uses 8% more energy than T, and 15% more energy than E.

RQ2: How does Astro compare against fixed and immutable best configuration choices? If we fix the hardware configuration, then 4b4L (4 bit, 4 LITTLE cores) gives us the best runtime and the best energy consumption for the simulation of Fluidanimate. This configuration is 45% slower than Astro, yet it is 4% more energy efficient. The fact that Astro, and the energy oracle, could beat 4b4L is surprising. We believe that

GaussianProcessClassifier : 0.559210526316					
Jaccard similarity: 0.559210526316					
Hamming loss: 0.440789473684					
		precision	recall	f1-score	support
	0	0.00	0.00	0.00	33
(a)	1	0.64	0.91	0.75	77
	2	0.35	0.62	0.45	24
	3	0.00	0.00	0.00	18
avg / total		0.38	0.56	0.45	152
<hr/>					
GaussianProcessClassifier : 1.0					
Jaccard similarity: 1.0					
Hamming loss: 0.0					
		precision	recall	f1-score	support
	0	1.00	1.00	1.00	33
(b)	1	1.00	1.00	1.00	77
	2	1.00	1.00	1.00	24
	3	1.00	1.00	1.00	18
avg / total		1.00	1.00	1.00	152
<hr/>					
Gaussian NB : 0.914473684211					
Jaccard similarity: 0.914473684211					
Hamming loss: 0.0855263157895					
		precision	recall	f1-score	support
	0	0.72	1.00	0.84	33
(c)	1	1.00	0.83	0.91	77
	2	1.00	1.00	1.00	24
	3	1.00	1.00	1.00	18
avg / total		0.94	0.91	0.92	152

Figure 4.4. Supervised learning applied to the 10 different phases of the Meabo micro-benchmark [ARM-software]. Fixing the best model for the three approaches, that were the same (the GPC in the hybrid had also the same precision as the Gaussian Naive Bayes). Part (a) of the Figure shows the result considering three dynamic features (relative CPU load to the number of used cores, instruction per cycles and cache miss ratio), (b) shows the result using 16 static features. The (c) shows the result of the hybrid approach, combining both features.

this outcome is due to the Linux Scheduler. It is responsible to map the threads in flight to the big and LITTLE cores. once threads are assigned, they will tend to stay statically allocated to favor cache locality. By sometimes changing the hardware configuration, Astro has the chance to force the Linux scheduler to perform a new load balancing of threads across cores. To give the reader some perspective on this result, we have also shown the configuration that yields the slowest and more power hungry

execution: 1b0L. It is almost 15 times slower than Astro, and spends 3.6x more energy.

RQ3: How does Astro perform when compared with state-of-the-art program schedulers? We have compared Astro against Hipster [44] and Octopus-Man [46]. The implementation of Hipster differs slightly from the original description of Nishtala, although we have reused much of their code base. Hipster was originally conceived to deal with cloud workloads; hence, we had to customize its state and reward function for multithreaded programs. In this experiment, both, Hipster and Astro use the same reward function, which tends to minimize energy consumption, although it also provides incentive to reduce runtime. Octopus-Man is the profiling mechanism used in Hipster; hence, it does not use the notion of reward. Overall, Astro produces code that runs faster than Hipster: when scheduled with Astro, Fluidanimate runs 37% faster. However, Astro uses 4% more energy.

RQ4: Is the combination of static and dynamic information really necessary?

Another interesting question concerns how Astro performs when compared against a purely dynamic approach, or a purely static approach. In the context of this evaluation, we call *purely static* a version of Astro that performs the scheduling using only program phases (see Definition 3.2.2). Along a similar direction, we call *purely dynamic* the version of Astro that only uses hardware phases to solve . Figure 4.5 compares all these approaches. Overall, they tend to produce the same maximum reward after some executions of the same application. However, the hybrid implementation of Astro tends to learn faster. After 80 executions of the same application, we have observed that this maximum reward varies less than 8% across different episodes. If we use only static information, then this variance is above 12%. If we use only dynamic data, then we get a variance above 13%.

4.0.3 Results in an Actual Device

Table 4.1 shows the runtime behavior of four different solutions to : Astro (purely static and hybrid approaches), Global Task Scheduling (GTS), and Hipster. GTS is a scheduling algorithm developed by ARM. This scheduler is aware of the different compute capabilities of big and LITTLE cores in the system. It uses historical data of the running tasks and active cores to determine where each individual thread will run. By tracking the load information at runtime, GTS migrates tasks that are compute-intensive to big cores and those that are less intensive to little cores. Load balancing heuristics are periodically executed to minimize concentrating compute-intensive

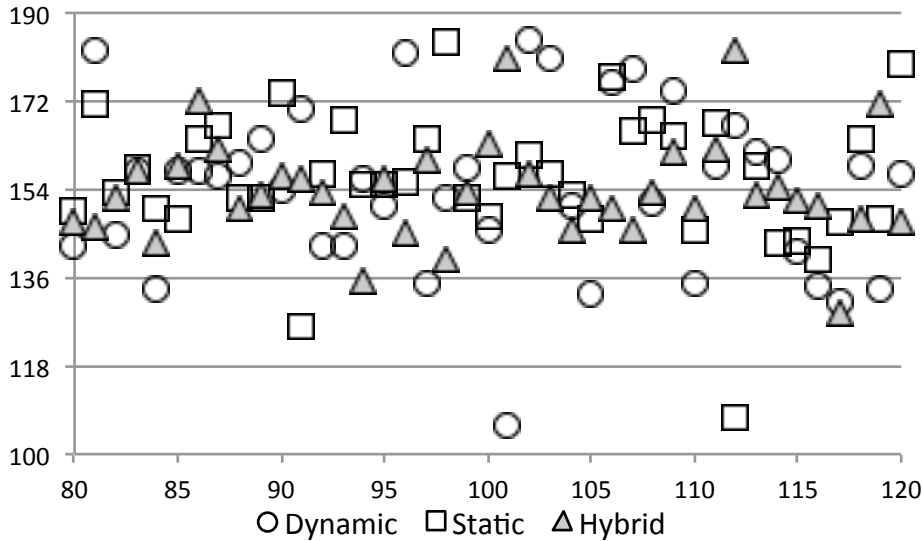


Figure 4.5. Comparison between three different implementations of Astro. They vary in the way they choose information to feed the neural-network. **Dynamic:** uses only hardware phases (Definition 3.2.3); **Static:** uses only program phases (Definition 3.2.2); **Hybrid:** uses both phases. Each dot along the X-axis represents one execution of `fluidanimate`. The Y-axis represents the maximum reward observed in that execution. We keep the state of the neural-network across executions, so that it can learn. We show results after 80 of such “episodes”.

threads excessively on big cores and letting little cores underutilized. In our setting, GTS has access to all the eight cores available. The numbers reported for Astro include all the overhead of monitoring and adapting the target application.

In the first five benchmarks in Table 4.1 were taken from Rodinia. Astro, in its static or hybrid flavours, outperforms GTS in four samples; however, our results tend to vary among multiple runs. We have included error margins for this experiment (averages of five samples). Thus, even though Astro tends to be more efficient than GTS, in three, out of five results, we are within the error interval. This means that if we consider the best version of GTS, and the worst version of Astro, then we get opposite results. In general the static version of Astro tends to outperform its dynamic version, due to the reduced overhead. However, in one of the experiments, involving `ParticleFilter`, the static version was largely outperformed by the other approaches, due to a bad scheduling decision (`1b2L`). In this case, the lack of runtime information seems to prevent it from adjusting early decisions that turn out to be inefficient. The fact that Astro can consistently outperform it on these five applications testifies in favour

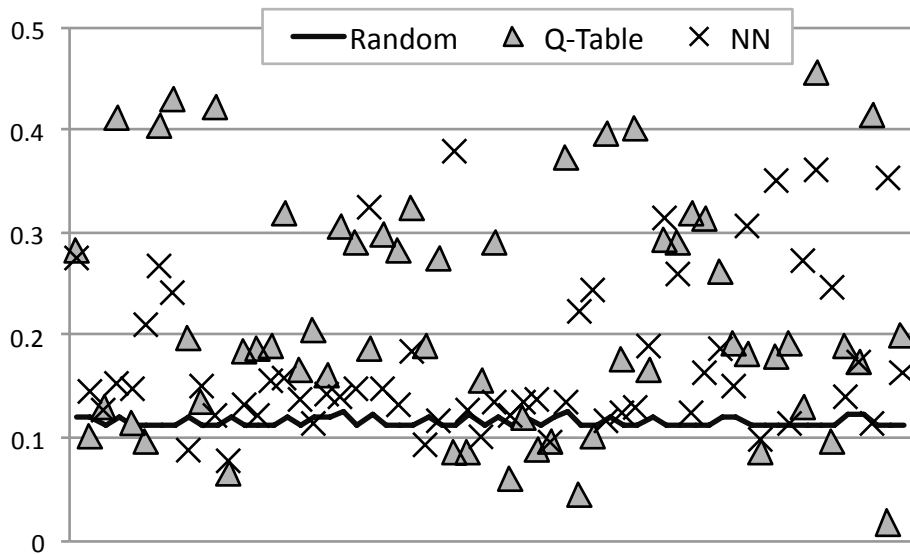


Figure 4.6. Comparison between three different implementations of Astro. They vary in the way they choose information to feed the neural-network. NN: uses a multi-layer perceptron in order to learn the q-function; Q-table: uses a table as the a way to learn the q-function; The Y-axis represents the normalized average reward observed in that execution. We show results after 80 of such “episodes”.

of considering program structure when taking scheduling decisions.

On BFS and Swaptions applications, the actions learned on Static method were equal as for all its phases (4b4L). For these scenarios, Astro suffers a bit from its overhead, as it is analyzed on Section 4.0.4. Finally, on Fluidanimate, we had only one phase instrumented, and the decision turned out to be bad, 1 little and 2 bigs. The Static didn’t do good fixed on this configuration, and the Hybrid, on the process of leaving this bad configuration ended up wasting more energy and time. Energy is calculated in Joules by multiply the sampling interval, which was 0.1 seconds, to each instantaneous power given in Watts by Powmon [59].

4.0.4 Overhead analysis

The cost of the Astro overhead can be decomposed in two leading sources: the instrumented code added to the original code and the PMUs monitor overhead. Regarding the former, the code, in the hybrid deploy, consults performance measure units and static features, then adapts to chosen configuration.

Time (s)	GTS	Static	Hybrid	Hipster
hotspot3D	24.8 0.8	24.3 0.1	25.7 4.6	23.0 0.4
cfd	564.6 0.2	563.7 0.4	564.2 0.3	604.1 20.0
hotspot	<u>124.5</u> 1.1	125.8 1.8	123.8 1.0	131.0 0.6
sradv2	69.4 2.0	62.9 0.9	65.9 1.5	60.1 0.8
particlefilter	210.4 3.9	548.3 2.8	<u>202.4</u> 5.1	200.7 3.3
bfs	<u>72.9</u> 8.2	103.9 19.2	<u>77.9</u> 6.1	71.1 8.2
fluidanimate	302.0 1.2	589.1 2.0	833.9 6.9	311.0 2.6
swaptions	100.2 0.7	101.4 1.1	96.2 2.0	103.0 0.5
Energy (J)	GTS	Static	Hybrid	Hipster
hotspot3D	<u>75.0</u> 10.8	72.9 6.6	84.5 16.2	110.8 1.9
cfd	2162.1 1.5	2157.5 1.3	2159.4 1.0	2293.7 59.2
hotspot	605.6 2.2	622.3 13.1	622.1 7.7	665.6 4.1
sradv2	331.3 17.9	336.3 18.5	310.4 14.6	340.6 22.1
particlefilter	810.6 52.6	2025.1 26.2	<u>801.7</u> 47.5	926.6 16.6
bfs	244.1 10.8	311.7 25.5	256.2 16.7	259.0 19.6
fluidanimate	1491.9 28.8	2568.4 111.6	3163.6 155.3	1578.3 25.5
swaptions	536.4 9.5	526.9 9.7	567.0 9.7	512.7 5.1

Table 4.1. Time (Top, in seconds) and Energy (Bottom, in Joules) comparison between Astro, GTS and Hipster. Each value represented as pair: median|standard deviation. “Static” is the purely static version of Astro, shown in Figure 3.4 (b). “Hybrid” is the version that uses runtime information to improve on the static decisions, shown in Figure 3.4 (c). Bold values highlight best results; underscore boxes show results within error margin.

Improvements In order to reduce these overheads a few techniques were applied:

1. we forced a minimum time between any two consecutive instrumentation.
2. we only instrument (1)“hot code segments” (we also call hot segments) and (2)“code segments with special characteristics”. We define “hot code segments” by doing a estimate of the number of instruction statically. If it is more than a constant ($c=1000$), we then instrument this function. The constant we defined after a comparison if we were taking the same hot segments as pointed out by the vtune profiler[Intel]. We “define code segment with special characteristics” if it consists in a thread sync, network sync or thread sleep region.
3. any file consulted by the instrumented code were mapped to memory (tmpfs), which is also important in the Astro deployment.

Binary size Regarding the amount of instrumentation code introduced at binary level, the Astro library still introduces non-negligible extra code to the applications. Code re-factoring and usage of fine-tuned compiling optimization flags (e.g. `-O3` instead of the original application compiling flags), might mitigate this issue. Figure 4.1 illustrates shows the amount of extra code inserted per application.

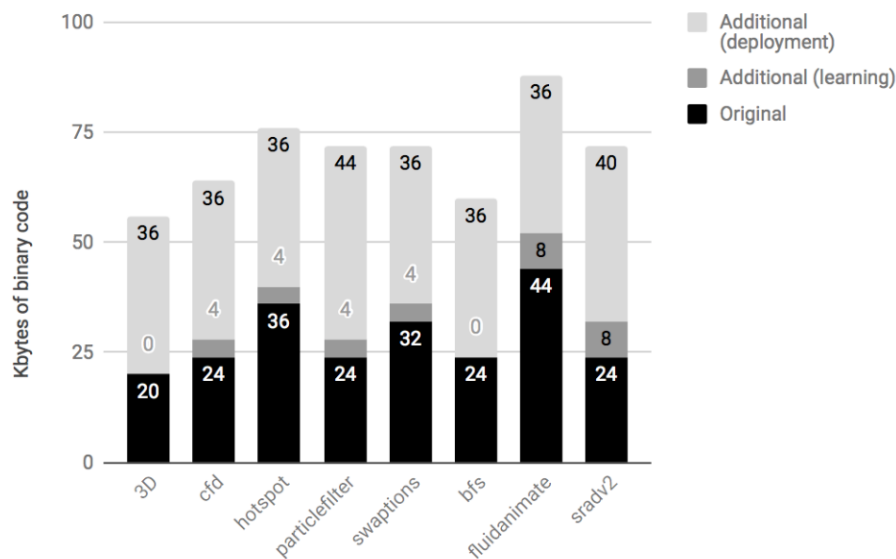


Figure 4.7. This figure shows the binary code size on different applications. The Y-axis represents the size in Kbytes. The X-axis represents the different applications. Plot shows from the original size in black, learning library adds in gray and deploy adds in light gray. Binary size computed via the “`du -h`” command.

The original size of the binaries are in Kbytes. The hotspot 3D (called 3D, in the figure) for example, has 20KB, CFD has 24KB, hotspot and particle-filter 24KB, and so on, represented in black. In dark gray, we show how many bytes were added to the learning phase (first binary instrumentation). We might recall that during learning phase, we need to instrument the application to output its phases and dynamic information. At this point, hotspot 3D stays around 20KB (20KB plus a few more bytes), CFD has 28KB (the original 24KB plus 4KB due to the learning instrumentation), hotspot ends up with 40KB, particle-filter ends up with 28KB. They are represented by the gray color. Once learning is done, we generate the final binary. This final binaries, for both static or hybrid approach, adds roughly 36-40KB to the original binary code, shown in light gray. The CFD application, for example, is about 36KB larger than its original size, ending up with 60KB). For those applications, during the learn-

ing phase, we observed additional few bytes (0KB) up to 8KB. In the final binary, it varied from 36KB to 44KB.

Chapter 5

Conclusion

This dissertation has presented overview of the Code Placement Problem and has introduced a framework, Astro, a program scheduler tailored to heterogeneous architectures. It creates an adaptive program that migrates by itself, without human intervention. Astro uses machine learning to adapt a program to runtime conditions. However, it departs from previous approaches, also based on machine learning, because it takes program characteristics into consideration. To this end, Astro relies on the compiler to identify program regions that contain similar syntactical features. We classify these features in sets called program phases, and track, at runtime, which program phase is currently valid. When combined with dynamic data, this information lets a neural network train the program, so to maximize some metric of efficiency, such as energy or runtime. By combining static and dynamic information, we are, effectively, building architecture-aware code optimizations.

5.1 Open questions and clarifications

The ideas around Astro can still be explored along many different directions, for this work still leaves many unanswered questions, such as which program features are more effective when generating code optimizations and which dynamic data is really essential to train a program to its environment. We hope to provide answered to these questions in the future.

Multiple applications scenario Even though we isolate applications in order to test the Astro system, we could clarify that, in theory, the design of Astro is not fixed to a single-application environment. When multiple applications are already running on a hardware, the dynamic information of the hardware will change and this should

lead the instrumented program to adapt changing its affinity to this environment. However, in a scenario which we have multiple applications trying to fit in the same hardware, we should not allow more than one application to adjust it, for example, via DVFS. One application should be able to adjust the hardware, but others should not, limiting themselves to the core-affinity adjusts.

In the multiple adaptive applications trying to adjust the hardware for their needs, it brings interesting questions, such as:

- Is it feasible to put in current operating systems, many instrumented programs which can adjust the hardware for themselves?
- If not, what would we need to modify in these operating systems?
- Should a program let the operating system know that it will take care of scheduling itself?
- Would it be the case of having an operating system that operates with at least three kinds of programs: programs that are responsible for scheduling themselves but cannot modify the hardware, programs responsible for scheduling themselves and that can modify the hardware and last programs which the operating system will take care of the allocation?
- How to synchronize these programs around a common optimization goal? For example, do they need a cooperation factor affecting their rewards?

Problem on handling application's code Astro has a few limitations inherent to its design and related to handling application's code. Firstly, the current version of Astro needs the application's source. Source codes are sometimes not available. Providing the system as an open source project address this issue, so individuals can take Astro and apply on its own source code, without the need of sharing with us their code. In the future, one could also instrument the binary code instead of the source code. There are two other limitations which have hitherto exposed to us: some applications cannot change the structure of their programs, in order to meet, for example, a memory layout; and some critical applications cannot have any kind of instrumentation as this new code can be a future source of bugs or exposition to malicious attacks. For those two last, the expected solution would be removing the instrumentation component, which is not addressed, as it is our main component to the goal of producing adaptive programs. Notice that this limitation is common among tools that perform dynamic analysis via code instrumentation [34; 14; 22].

Miscellaneous A few interesting questions we have approached might be revisited and are still open: (1) How to normalize a floating point value which you do not know

its range, in the performance metrics scenario? We currently do a local normalization by the total number of instructions. Other approaches as estimating minimum and maximum and updating these values over time seems interesting. (2) What other approaches to produce an Oracle would be interesting to have? We have decided for a greedy oracle. Having execution traces on all the configurations, it and decides for the best action that maximizes reward at that moment. Maybe we should have considered more elaborated Oracles with the traces we had?

Bibliography

- [Goo] Conheça os vencedores do programa de bolsas de pesquisa do google para a américa latina. Available at <https://brasil.googleblog.com/2017/08/conheca-os-vencedores-do-programa-de.html>.
- [Int] Month: September 2017. Available at <http://homepages.dcc.ufmg.br/~juniocezar/intelligentDVFS/2017/09/>.
- [3] Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O’Boyle, M. F., Thomson, J., Toussaint, M., and Williams, C. K. (2006). Using machine learning to focus iterative optimization. pages 295--305.
- [ARM-software] ARM-software. Arm-software/meabo. Available at <https://github.com/ARM-software/meabo>.
- [5] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187--198.
- [6] Barik, R., Farooqui, N., Lewis, B. T., Hu, C., and Shpeisman, T. (2016). A black-box approach to energy-aware scheduling on integrated CPU-GPU systems. In *CGO*, pages 70--81. ACM.
- [7] Bessa, T., Quintão, P., Frank, M., and Pereira, F. M. Q. (2016). Jetsonleap: A framework to measure energy-aware code optimizations in embedded and heterogeneous systems. In *SBLP*, pages 16--30. Springer.
- [8] Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The parsec benchmark suite: Characterization and architectural implications. In *PACT*, pages 72--81. ACM.
- [9] Blagodurov, S., Zhuravlev, S., and Fedorova, A. (2010). Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1--8:45.

- [10] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177--186. Springer.
- [11] Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., and Storaasli, O. O. (2010). State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1--33.
- [12] Brooks, D. M., Bose, P., Schuster, S. E., Jacobson, H., Kudva, P. N., Buyuktosunoglu, A., Wellman, J.-D., Zyuban, V., Gupta, M., and Cook, P. W. (2000). Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20:26--44.
- [13] Buchty, R., Heuveline, V., Karl, W., and Weiss, J.-P. (2012). A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. *Concurrency and Computation: Practice and Experience*, 24(7):663--675.
- [14] Cadar, C., Dunbar, D., and Engler, D. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209--224, Berkeley, CA, USA. USENIX Association.
- [15] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *HISWC*, pages 44--54. IEEE.
- [16] Cong, J. and Yuan, B. (2012). Energy-efficient scheduling on heterogeneous multi-core architectures. In *ISLPED*, pages 345--350. ACM.
- [17] Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. (2017). End-to-end deep learning of optimization heuristics. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, pages 219--232. IEEE.
- [18] Das, A., Shafik, R. A., Merrett, G. V., Al-Hashimi, B. M., Kumar, A., and Veeravalli, B. (2014). Reinforcement learning-based inter-and intra-application thermal optimization for lifetime improvement of multicore systems. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1--6. IEEE.
- [19] Delimitrou, C. and Kozyrakis, C. (2014). Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS*, pages 127--144. ACM.
- [20] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319--349.

- [21] Gaspar, F., Taniça, L., Tomás, P., Ilic, A., and Sousa, L. (2015). A framework for application-guided task management on heterogeneous embedded systems. *ACM Trans. Archit. Code Optim.*, 12(4):42:1--42:25.
- [22] Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05*, pages 213--223, New York, NY, USA. ACM.
- [23] Greenhalgh, P. (2011a). Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, pages 1--8.
- [24] Greenhalgh, P. (2011b). Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 17.
- [25] Grewe, D. and O'Boyle, M. F. (2011). A static task partitioning approach for heterogeneous systems using opencl. In *International Conference on Compiler Construction*, pages 286--305.
- [26] Grochowski, E. and Annavaram, M. (2006). Energy per instruction trends in intel microprocessors. *Technology@ Intel Magazine*, 4(3):1--8.
- [27] Harris, L. C. and Miller, B. P. (2005). Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33(5):63--68.
- [Intel] Intel. Intel vtune performance analyzer.
- [29] Jain, A., Laurenzano, M. A., Tang, L., and Mars, J. (2016). Continuous shape shifting: Enabling loop co-optimization via near-free dynamic code rewriting. In *MICRO*, pages 1--12. ACM/IEEE.
- [30] Joao, J. A., Suleman, M. A., Mutlu, O., and Patt, Y. N. (2012). Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, pages 223--234. ACM.
- [31] Khan, U. A. and Rinner, B. (2014). Online learning of timeout policies for dynamic power management. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):96.
- [32] Khokhar, A. A., Prasanna, V. K., Shaaban, M. E., and Wang, C.-L. (1993). Heterogeneous computing: Challenges and opportunities. *Computer*, 26(6):18--27.

- [33] Lago, P., Gu, Q., Bozzelli, P., et al. (2014). A systematic literature review of green software metrics.
- [34] Lakhotia, K., Harman, M., and Gross, H. (2013). AUSTIN: An open source tool for search based software testing of C programs. *Inf. Softw. Technol.*, 55(1):112--125.
- [35] Lattner, C. and Adve, V. S. (2004). "llvm: A compilation framework for lifelong program analysis & transformation". In *CGO*, pages 75--88. IEEE.
- [36] Luk, C.-K., Hong, S., and Kim, H. (2009). Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, pages 45--55. ACM.
- [37] Lukefahr, A., Padmanabha, S., Das, R., Sleiman, F. M., Dreslinski, R. G., Wenisch, T. F., and Mahlke, S. (2016). Exploring fine-grained heterogeneity with composite cores. *IEEE Transactions on Computers*, 65(2):535--547.
- [38] Margiolas, C. and O'Boyle, M. F. P. (2016). Portable and transparent software managed scheduling on accelerators for fair resource sharing. In *CGO*, pages 82--93. ACM.
- [39] Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., and Pereira, F. M. Q. a. (2017). DawnCC: Automatic annotation for data parallelism and offloading. *TACO*, 14(2):13:1--13:25.
- [40] Mittal, S. and Vetter, J. S. (2015). A survey of cpu-gpu heterogeneous computing techniques. *Comput. Surv.*, 47(4):69:1--69:35.
- [41] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529--533.
- [42] Nickolls, J. and Dally, W. J. (2010). The gpu computing era. *Micro, IEEE*, 30(2):56--69.
- [43] Nikita Mishra, Connor Imes, J. D. L. H. H. (2018). Caloree: Learning control for predictable latency and low energy. In *The 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (To appear)*. ACM.
- [44] Nishtala, R., Carpenter, P. M., Petrucci, V., and Martorell, X. (2017). Hipster: Hybrid task manager for latency-critical cloud workloads. In *HPCA*, pages 409--420. IEEE.