Compiler-assisted adaptive program scheduling in big.LITTLE systems

Marcelo Novaes, Vinicius Petrucci, Abdoulaye Gamatié, Fernando Magno Quintão Pereira

Compiler-assisted Adaptive Program Scheduling in big.LITTLE Systems

Marcelo Novaes
Department of Computer Science
UFMG
Brazil
marcelonovaes@dcc.ufmg.br

Vinícius Petrucci
Department of Computer Science
UFBA
Brazil
vinicius.petrucci@dcc.ufba.br

Abdoulaye Gamatié
LIRMM
CNRS
France
abdoulaye.gamatie@lirmm.fr

Fernando Quintão
Department of Computer Science
UFMG
Brazil
fernando@dcc.ufmg.br

Abstract

Energy-aware architectures provide applications with a mix of low (LITTLE) and high (big) frequency cores. Choosing the best hardware configuration for a program running on such an architecture is difficult, because program parts benefit differently from the same hardware configuration. State-of-the-art techniques to solve this problem adapt the program's execution to dynamic characteristics of the runtime environment, such as energy consumption and throughput. We claim that these purely dynamic techniques can be improved if they are aware of the program's syntactic structure. To support this claim, we show how to use the compiler to partition source code into program phases: regions whose syntactic characteristics lead to similar runtime behavior. We use reinforcement learning to map pairs formed by a program phase and a hardware state to the configuration that best fit this setup. To demonstrate the effectiveness of our ideas, we have implemented the Astro system. Astro uses Q-learning to associate syntactic features of programs with hardware configurations. As a proof of concept, we provide evidence that Astro outperforms GTS, the ARM-based Linux scheduler tailored for heterogeneous architectures, on the parallel benchmarks from Rodinia and Parsec.

Keywords big.LITTLE architecture, Adaptation, Compiler

1 Introduction

Contemporary hardware found in mobile phones and data centers sport multiple ways to reduce energy consumption. Two of these techniques are the combination of low and high power cores (the so called big.LITTLE architectures) [7], and the ability to adjust power and speed dynamically (DVFS) [15]. This design gives us the possibility to allocate to each parallel 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 said 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.

We call the task of allocating parts of a parallel program to processors the *code placement problem*. State-of-the-art approaches solve this problem dynamically or statically. Dynamic solutions [18, 20, 22] are implemented at the runtime level, at the operating system, or via a middleware. Static approaches [11, 19, 21, 31] are implemented at the compiler level. The main advantage of the dynamic approach is the fact that it can use runtime information to weight the choices it makes. Static techniques, in turn, provide reduced runtime cost and better leverage of program characteristics. In this paper, 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 start from a technique that has been proven effective to schedule computations in big.LITTLE architectures: *Reinforcement learning*. Nishtala *et al.* [20] showed that reinforcement learning helps to find good hardware configurations to applications subject to varying dynamic conditions. The beauty of this approach is adaptability: it provides 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 find a set of scheduling decisions that suits the underlying hardware. Yet, "enough time" can be too long. The universe of runtime states is unbounded, and program behavior is hard to predict without looking into its source code. To speedup convergence, we resort to the compiler.

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 shorter.

To validate our ideas, we have materialized them into a framework to instrument and execute applications in heterogeneous architectures: the *Astro System*. Astro collects syntactic characteristics from programs and instruments them using LLVM [14]. Experiments in programs from Parsec [4] and Rodinia [6] running on an Odroid XU4 show that we can obtain speedups of more than 10% over the default GTS scheduler used in ARM-based systems. Such numbers result from the following contributions:

Observations: in Section 2, we demonstrate that the performance of a program running on a heterogeneous architecture vary depending on which part of its text we consider. This observation points us to the key insight: the possibility of augmenting an adaptive runtime apparatus with awareness of program characteristics.

Compiler: in Section 3.1, we explain how to collect and discretize program features, and in Section 3.2, we explain how to instrument a program, so to use said features to fine-tune an adaptive code placement algorithm.

Runtime: in Section 3.3, we show how to integrate the static information that we collect with an adaptive runtime environment. Once we train a program, we generate code that maps different parts of it to suitable hardware configurations.

2 Empirical Observations

This section motivates our work through three empirical observations. First, different hardware configurations yield very different tradeoffs between power consumption and runtime speed for a program (Figure 1). Second, 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 (Figure 2). Third, the best hardware configuration for a program might not suit the needs of a different application (Figure 4). Central to the discussion in this section is the notion of a *hardware configuration*:

Definition 2.1 (Hardware Configuration)**.** *A heterogeneous architecture is formed by a set $P = \{p_1, p_2, \ldots, p_n\}$ of n processors. A hardware configuration is a function $H : P \mapsto Boolean$. If $H(p_i) =$* **True***, then processor p_i is said to be* active *in H, otherwise it is said to be* inactive.

First Observation. The same application might benefit differently from different hardware configurations. This benefit

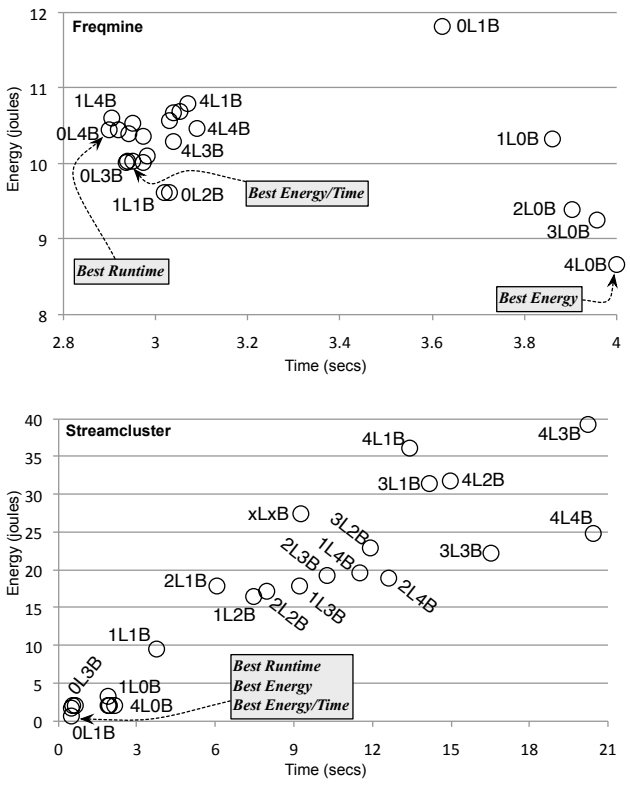

Figure 1. Energy vs Processing time spent by two PARSEC benchmarks using simsmall inputs. The notation xLyB denotes x LITTLE cores, and y big cores.

is measured in terms of processing time and energy consumption. Figure 1 shows how two benchmarks from the PARSEC suite – Freqmine and Streamcluster – fare on an Odroid XU4 board featuring 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 *bigs*, and the A7 cores *LITTLEs*. By switching on and off the different cores, we have 24 different hardware configurations[1]

Each dot in the figure represents the average of 10 executions on the same configuration, using the smallest[2] 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 lets us conclude that the energy and runtime footprint of applications vary greatly across different hardware

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

[2]This experiment would take 12 days using the largest inputs.

configurations. For instance, the most time efficient configuration for Freqmine is 0L4B, i.e., four bigs and no LITTLEs (2.90secs, 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 *phases*. Energy and runtime behavior are similar within the same phase, and potentially different across different phases.

Second Observation. The instantaneous power consumed by a program is not always constant. In other words, a program has *power phases*. Figure 2 (a) shows a program 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 3 shows the power profile of this program. This chart has been produced with JetsonLeap [3], an apparatus that let us measure the energy consumed by programs running on the Nvidia TK1 Jetson board[3]. JetsonLeap is formed by three components: the target Nvidia board (Figure 2 (b)), a data acquisition device, which reads the instantaneous power consumed by the board (Figure 2 (c)), and a synchronization circuit, which lets us communicate to the power meter which program event is running at each instant (Figure 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 is 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.

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 [20], to identify variations in program behavior. However, this approach has two shortcomings. First, distinct program parts, with very different resource demands

```
int main(int argc, char** argv) {
  int M1, N1, M2, N2;
  // Read first matrix from file 'argv[1]'
  int** m1 = readMatrix(argv[1],&M1,&N1);
  read_user_data();
  // Read second matrix from file 'argv[1]'
  read_user_data();
  int** m2 = readMatrix(argv[2],&M2,&N2);
  read_user_data();
  // Multiply both matrices, giving m3
  int** m3 = mulMatrix(m1,m2,M1,N1,N2);
  read_user_data();
  // Print all the matrices in the
  // standard output
  printMatrix(m1, M1, N1);
  printMatrix(m2, M2, N2);
  printMatrix(m3, M1, N2);
  read_user_data();
}
```

(a)

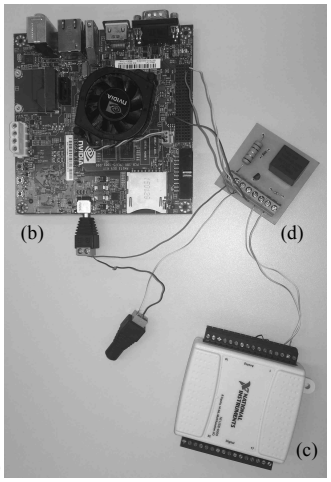

Figure 2. (a) Simple matrix multiplication implemented in C. (b) The Nvidia TK1 board. (c) NI 6009 Data Acquisition Device. (d) Synchronization circuit.

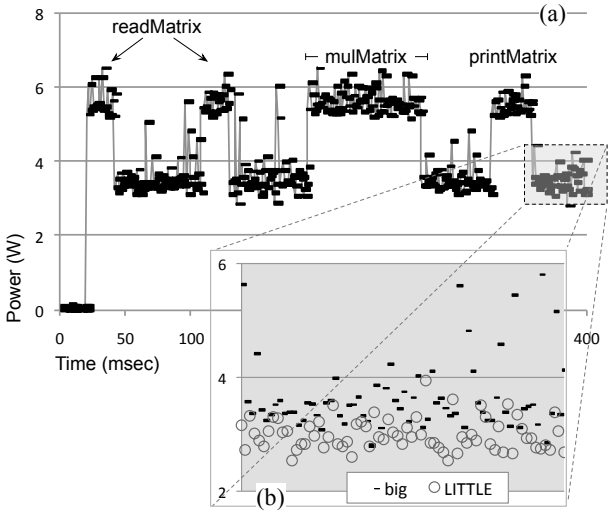

Figure 3. (a) Power profile of program seen in Figure 2. The NI 6009 sample rate was 1000 samples/sec. (b) Zoom of the power profile obtained during the last phase of the program.

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 2 is implemented via busy waiting. In this case, instead of the valleys observed in Figure 3, we would encounter a power line similar to that produced by CPU-intensive functions like mulMatrix. Second, profiling-based techniques face a trade-off 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

[3]In 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. 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.

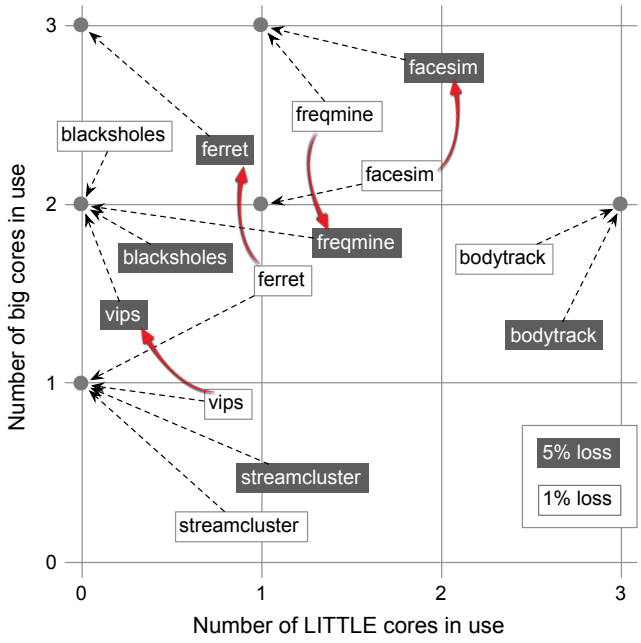

Figure 4. Best configurations for seven PARSEC applications, if we accept an slowdown of 1% or 5% to save more energy.

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.

Third Observation. The best architecture configuration, in terms of runtime or energy consumption, differs among programs. Figure 4 shows the best configurations that we have found on the Odroid XU4 setup, for six PARSEC applications. 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.

In the rest of this paper, we shall describe a general methodology, henceforth called the Astro system, which mixes static and dynamic analyses, to find good hardware configurations for the functions invoked during the execution of a program. In this section, 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.

3 The Astro System

This section describes the design and implementation of our approach to solve the problem of finding good hardware configurations for programs. We state this problem as follows:

Definition 3.1. *Scheduling of Programs in Heterogeneous Architectures (SPha)*
***Input**: a program P, its input I, hardware configurations $H_1, \ldots H_n$, energy threshold E, and 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%.*

In this paper, we solve SPha using an assortment of techniques, which give us the means to generate code that is well adapted to different architectures and workloads. Figure 5 provides a general overview of these techniques, emphasizing the different stages over which we go in the process of solving SPha. Section 3.1 describes program instrumentation, a necessary step to partition a program into phases. Section 3.2 goes over actuation; and Section 3.3 discusses the generation of the final program. However, before we move into the particulars of our solution to SPha, we provide a

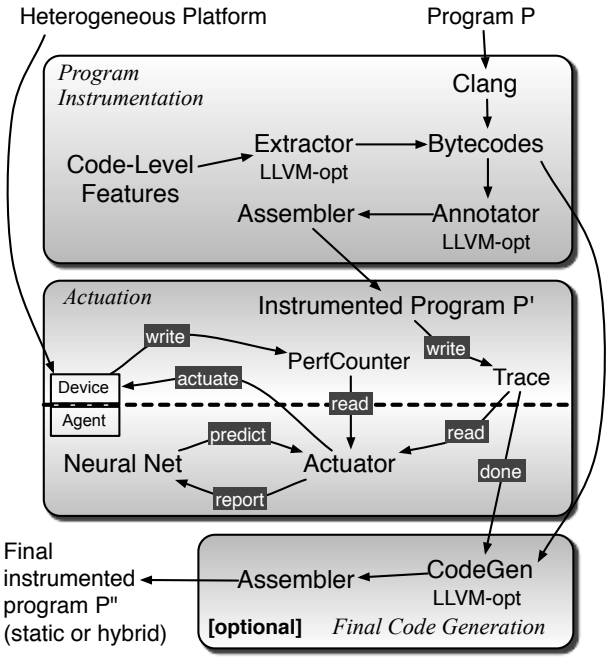

Figure 5. The Astro System.

brief introduction to Q-Learning, the flavour of reinforcement learning that we have adopted.

Q-Learning. Q-learning is a reinforcement learning algorithm [28]. Given some notion of state (Definition 3.2) and reward (Definition 3.7), it finds an optimized policy to perform the best action (Definition 3.9). 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 $R : S \times A \rightarrow \mathbb{R}$, 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 says that the results of an action depends only on the state where the action was taken, regardless of any other prior states.

3.1 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 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 3.1.1, and hardware phases are discussed in Section 3.1.2.

Definition 3.2 (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.*

3.1.1 Program Phases

Static Program Phases depend only on the syntax of a program. Definition 3.3 formalizes this notion. A static program phase is not equivalent to a *program region*, because different regions can present the same set of feature ranges. Example 3.4 clarifies the meaning of these definitions.

Definition 3.3 (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.*

Example 3.4. *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 number of nested loops 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 heuristic to estimate it is: $\Sigma_i 10^n$, for every I/O call i nested into n loops. Potential intervals for this metric are $[0, 1)$, $[1, 10)$, $[10, 100)$ and $[100, +\infty]$. The $3 \times 3 \times 4$ possible combinations of these 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.*

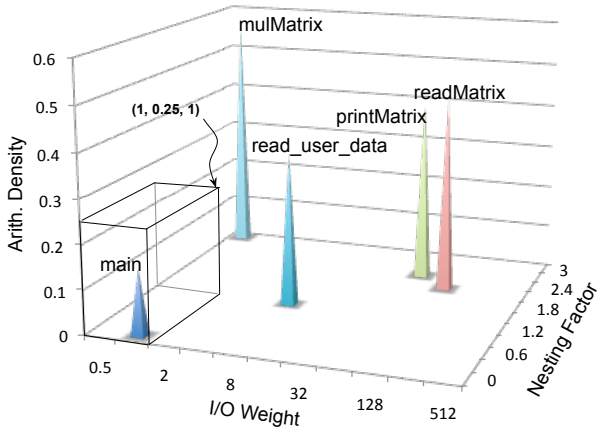

Figure 6. Mapping the functions in Figure 2 (a) to program phases.

In this paper, we mine (e.g., collect) features from the intermediate program representation that the compiler manipulates before producing executable code. We have implemented a *Phase-Extractor* using the LLVM compiler. 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 of regions are possible, such as instruction, basic block, loop, Single-Entry-Single-Exit block [9], 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 amounts to say that the instrumented program adds logic to change phases at the entry point of functions, and around certain library calls.

Example 3.5. *Figure 6 shows the five functions in Figure 2, classified according to features seen in Example 3.4. 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 6 shows the sub-space that corresponds to the phase:* Arith.Density $\in [0, 0.25)$, *I/O Weight* $\in [0, 1)$ *and* NestingFactor $\in [0, 1)$. *Function main, in our example, fits in this phase.*

Our Choice of Program Phases. In our 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. We use the following features:

- 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 when the program invokes a multi-thread barrier that forces it to wait for some blocking event.
- Net: true when the program invokes a library call that forces it to wait for some network-related event.
- Sleep: true when the program invokes a sleep library call that forces it to wait unconditionally.

We have defined four program phases, which appear as combinations of the features above. This choice is arbitrary. 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 paper: that static features greatly enhance the dynamic scheduling of computations in heterogeneous hardware. The program phases that we shall consider in Section 4 are:

- Blocked: Barrier = true or Net = true or Sleep = true or Locks-Dens > 0.5;
- I/O Bound: IO-Dens + Mem-Dens > 0.5 and not(Blocked) and Locks-Dens = 0;
- CPU Bound: Int-Dens + FP-Dens > 0.5 and not(Blocked);
- Other: in case none of the previous relations hold.

3.1.2 Hardware Phases

While the program phases seen in Section 3.1.1 depend only on syntactic program characteristics, *hardware phases* depend on the dynamic state of the hardware:

Definition 3.6 (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 $\{C_1, C_2, \ldots, C_n\}$, where each C_i is partitioned into R_i phases, then a hardware phase is any combination within the product $R_1 \times R_2 \times \ldots \times R_n$.*

The monitoring of hardware phases does not require program instrumentation. Instead, an *actuator* reads the state of hardware performance counters periodically. Modern architectures already provide an array of performance counters that can be queried. In this paper, we consider four kinds of counters to define hardware phases:

- 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 $[0, .1\%), [.1\%, .5\%), [.5\%, +\infty)$;

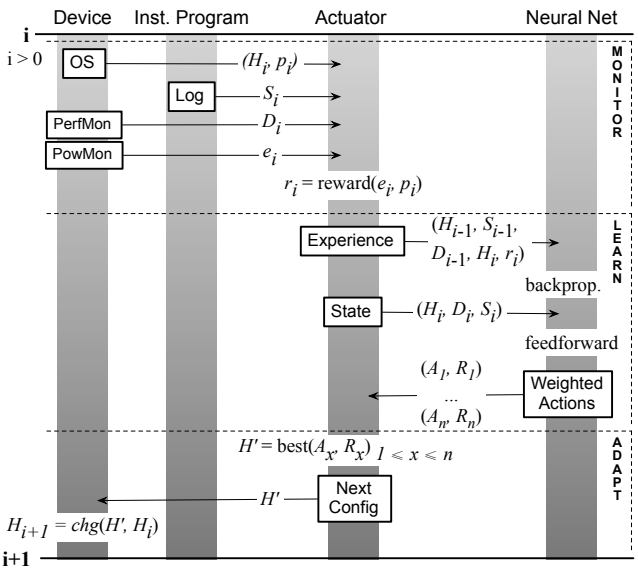

Figure 7. The Actuation Algorithm.

- CPU: utilization of the CPU, in the ranges $[0, 20\%)$, $[20\%, 50\%)$, $[50\%, +\infty)$.

Each counter is partitioned in three buckets. Therefore, we consider a total of $3 \times 3 \times 3 \times 3 = 81$ hardware phases.

3.2 Actuation

The heart of the Astro system is the Actuation Algorithm outlined in Figure 7. Actuation consists of *phase monitoring*, *learning* and *adapting*. These three steps happens at regular intervals, called *check points*, which, in Figure 7, we denote by i and i+1. The rest of this section describes these events.

3.2.1 Monitoring

To collect information that will be later used to solve SPHA, Astro reads four kinds of data. Figure 7 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 [32]): the energy e consumed since the last checkpoint.

The monitor collects this data at periodic intervals, whose granularity is configurable. Currently, it is 500 milliseconds. The recording of the program phase is aperiodic, following from instrumentation inserted in the program by the compiler. As discussed in Section 3.1.1, information is logged at the entry point of functions, and around 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:

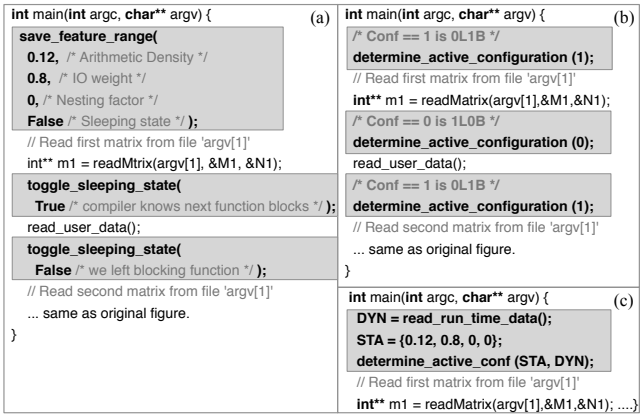

Figure 8. (a) Instrumentation to mine features. (b) Final instrumentation, inserted in production code.

Definition 3.7 (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 for energy, we let $\gamma = 1.0$. A value of $\gamma = 2.0$ emphasizes 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 [5].

Example 3.8. *Continuing with Example 3.5, Figure 8 (a) shows the instrumentation of function main (Figure 2) to log program phases.*

3.2.2 Learning

The learning phase uses the Q-learning algorithm. As illustrated in Figure 7, a key component in this process is a multi-layer Neural Network (NN) that receives inputs collected by the Monitor. The NN outputs the actions and their respective rewards to the Actuator so that a new system adaptation can be carried out. Following common methodology, learning happens in two phases: *back-propagation* and *feed-forwarding*. During back-propagation we update the NN using the experience data given by the Actuator (Figure 7). Experience data is a triple: the current state, the action performed and the reward thus obtained. 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. During the feed-forward phase, we perform predictions using the trained NN. Each node of the NN is responsible to accumulate the product of its associated weights and inputs. 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 *reward*, estimated by NN. Actions determine configuration changes; rewards determine the expected performance gain, in terms of energy and time, that we expect to obtain with the change. We use the method of gradient descent to minimize a loss function given by the difference between the reward predicted by the NN, and the actual value found via hardware performance counters.

3.2.3 Adaptating

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

Definition 3.9 (Action)**.** *Action is the act of choosing the next hardware configuration H to be adopted at a given checkpoint.*

An action may change the current hardware configuration; hence, adapting the program according to the knowledge inferred by the Neural Network. Following Figure 7, we start this step by choosing, among the pairs $\{(A_1, R_1), \ldots, (A_n, R_n)\}$, 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 adopt 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. If the Next Configuration is accessible, Astro enables it; otherwise, the whole system remains in the configuration H_i active at check point i. Such choice is represented, in Figure 7, 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.

3.3 Code Scheduling

After we have trained a program to a given architecture, we imprint this knowledge directly in that program's code. In Figure 5, 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 to mark program phases (see Section 3.1.1): at the entry point of functions, and around particular library calls. Example 3.10 illustrates this instrumentation.

Example 3.10. *Figure 8 shows the final actuation code for the program in Figure 2. Function determine_active_configuration tries to move the program to the configuration that has produced the largest rewards for that program phase. We consider two versions of instrumentation: static, as in Figure 8(b), and hybrid, as in Figure 8 (c). The latter can read hardware status to improve the decision making process.*

The static scheduling discussed in Example 3.10 always maps the same program region to the same hardware configuration. Hybrid scheduling might change decisions, given enough runtime information. As we show in Section 4, the static scheduling yields lower runtime overhead than Astro's hybrid scheduling. However, this modus operandi is unable to adapt the program to its workload; and cannot recover from bad decisions. A striking example is the benchmark ParticleFilter (see Fig. 10 in Section 4.2). In this case, even with the runtime overhead, the flexibility of hybrid instrumentation paid off in terms of energy and speed.

4 Evaluation

This section presents an experimental evaluation of the Astro system over several parallel benchmarks running on a big.LITTLE 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 compare against state-of-the-art schedulers?
- **RQ4**: How does Astro behave on an actual device?
- **RQ5**: How much does Astro increase code size?

Experimental Setup. We use two experimental setups: *program traces*, henceforth called *simulation*; and an actual device, the Odroid XU4. Experiments in Section 4.1 use simulation because they involve testing exhaustively every hardware configuration. Experiments in Section 4.2 run on an 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), running on Linux odroid 3.10.63, using the "performance" frequency governor, with cores at maximum speed. This device was also used to produce the simulation traces. We report CPU power consumption via PowMon [32]. Astro is implemented on LLVM 3.8.

Benchmarks. The simulation traces used in Section 4.1 were produced on Parsec's FluidAnimate [4]. Experiments on Section 4.2 use eight benchmarks from Rodinia and Parsec. These are the only programs that we can currently instrument, as our LLVM module does not recognize mangled C++ routines yet (to discover program phases such as I/O density – Sec. 3.1.1). We used FluidAnimate to obtain the initial learning parameters; hence, we do not use it for validation.

4.1 Results in the Simulated Environment

In this section we report results that are hard to obtain on an actual device, because they involve exhaustive search on the universe of valid hardware configurations. 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 time consuming, thus, we have produced them only for fluidanimate. We took between 410 seconds to up to 7,000 seconds to produce each trace, depending on the hardware configuration. Figure 9 compares seven different scheduling strategies built on top of this simulator, applied on fluidanimate.

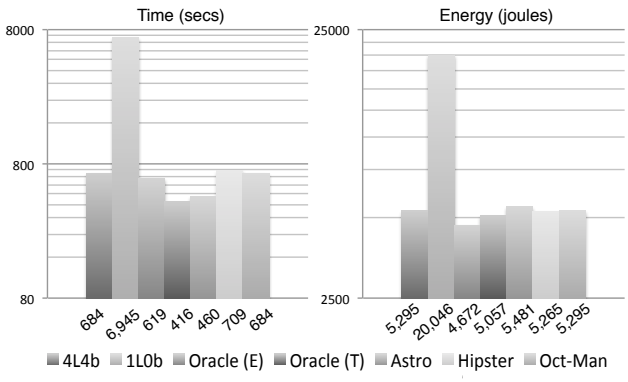

Figure 9. Comparison between Astro and a system that chooses the next configuration randomly.

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*. Our oracle is not an optimal global solution to SPHA. 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 9 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 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 have found out that 4b4L tends to slowdown programs at critical sections, due to an excess of conflicts between threads. Astro eventually learns to use configurations with less cores at these program phases; hence, speeding up execution. Figure 9 also shows 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 tried to implement, on the simulator, two well-known schedulers for big.LITTLE architectures: Hipster [20] and Octopus-Man [22]. The implementation of Hipster used in Figure 9 differs slightly from the original description of Nishtala *et al*, 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. Octopus-Man is the profiling mechanism used in Hipster; hence, it does not use the notion of reward. Astro produces code that runs 17% faster than Hipster, and 15% faster than Octopus-Man. However, Astro uses 6% more energy than the former, and 4% more than the latter.

4.2 Results in an Actual Device

RQ4: How does Astro behave on an actual device? Figure 10 shows the runtime (5 samples) of three different solutions to SPHA: Astro (purely static or hybrid), and Global Task Scheduling (GTS). 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 threads excessively on big cores and letting little cores underutilized. Numbers reported for Astro include all the overhead of monitoring and adapting the target application.

Astro, in its static or hybrid flavours, yields faster code than GTS in six benchmarks, and more energy efficient code in five. We show two p-values next to each plot: S and H. The former is the probability that the static and purely dynamic (GTS) samples come from the same distribution. The latter relates the hybrid and purely dynamic distributions. The closer to zero, the more statistically significant are our results. We emphasize that GTS is a state-of-the-art approach, widely used in operating systems running on ARM hardware, and the fact that Astro can consistently outperform it testifies in favour of the benefits of syntax awareness when taking scheduling decisions. There is no clear winner between the hybrid and static versions of Astro. We observer that the former tends to be better in more regular (kernel-like) applications, such as CFD and sradv2. We also observe strong correlation between runtime and energy consumption, except for Swaptions. In that case, the Static version of Astro tends to avoid using the high-frequency cores, a fact that leads to slower runtime, but also to less power dissipation. In ParticleFilter the static version was penalized for a wrong scheduling decision: it stays in 1b2L, and the lack of runtime information prevents it from fixing this choice.

RQ5: How much does Astro increase code size? There are three different versions of instrumented programs: those used during Astro's learning phase; the programs that use static instrumentation; and the programs that use hybrid instrumentation. The binary size of the last two is the almost the same: it consists of code that collects data, plus the Astro library. The only different between static and dynamic instrumentation is the code used to collect dynamic data in the latter version. This different is too small; hence, in Figure 11 we include both types of binaries in the same bar: Instrumented. As the figure shows, most of the size overhead imposed by Astro is due to its dynamic library. This increase is constant across benchmarks. The amount of instrumentation in binaries grows linearly with the program size. This growth tends to be very small. As evidence to this small growth, in the Learning phase, binaries do not use any dynamically linked library; thus, code size expansion is due to instrumentation only, and it is small, as seen in Figure 11.

5 Related Work

The problem of scheduling computations in heterogeneous architectures (Definition 3.1) has attracted much attention in recent years. Table 1 provides a taxonomy of previous solutions to this problem. We group them according to how 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 SPHA concerns the moment when they are used: at compilation time, at runtime, or both.

Purely static approaches work at compilation time. They might be applied by the compiler, either automatically, i.e., without user intervention [8, 12, 16, 24, 26, 29], or not. In the latter case, developers can use annotations [19], domain specific programming languages [16, 26] or library calls [1] to indicate where each program part should run. In Table 1, techniques implemented at either the compiler or library levels are purely static. *Purely dynamic* approaches take into account runtime information. They can be implemented at the architecture level [13, 17, 25, 30, 33], or at the virtual

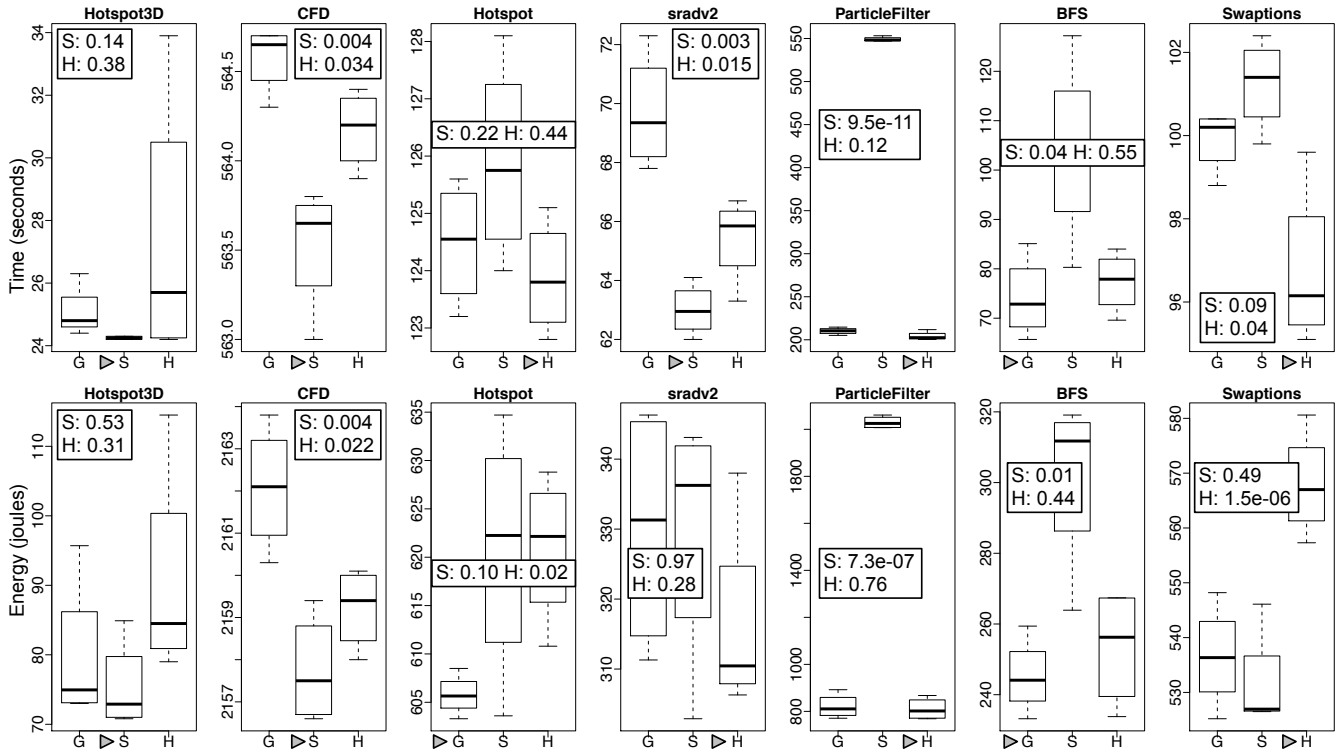

Figure 10. Time (Top) and Energy (Bottom) comparison between Astro and GTS (G). "Static (S)" is the purely static version of Astro (Fig. 8b). "Hybrid (H)" is the version that uses runtime information to improve on the static decisions (Fig. 8c). Numbers in boxes are p-values for the Static and Hybrid approaches, compared to GTS. Grey triangles indicate winning strategies.

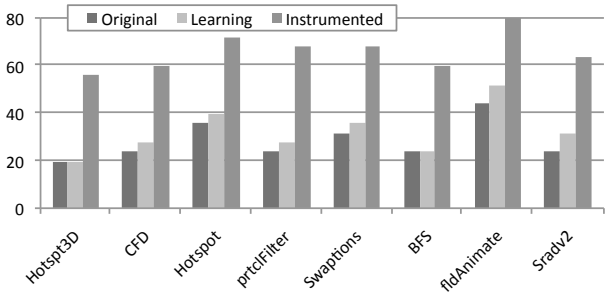

Figure 11. Code size increase. Y-axis shows code size (Kb).

machine (VM)/OS level [2, 10, 20, 22, 27, 34]. By leveraging runtime information, the system can use environment information, unknown at compilation time, to solve SPHA. 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. Approaches that mix static and dynamic techniques are called *hybrid*. Astro is a hybrid method. Other hybrid approaches to this problem exist [8, 23, 29].

Work	*Level*	*Source*	*Auto*	*Runtime*	*Learn*
[24]	C	Yes	Yes	No	Yes
[2]	C	Yes	Yes	Yes	No
[26]	C/L	Yes	No	Yes	No
[16]	C/L	Yes	No	Yes	No
[13]	A/L	Yes	No	No	No
[17]	A	No	Yes	No	No
[30]	A	No	Yes	No	No
[20]	O	No	Yes	Yes	Yes
[22]	O	No	Yes	Yes	No
[1]	L	Yes	No	No	No
[23]	O/C	Yes	Yes	Yes	No
[29]	O/C	Yes	Yes	Yes	No
[8]	O/C	Yes	Yes	Yes	No
Astro	O/C	Yes	Yes	Yes	Yes

Table 1. Comparison between different solutions to SPHA. *Level*: at which level the technique is implemented: Architecture (A), Operating System (O), Compiler (C) or Library/Programming model (L). *Code*: "Yes" if approach requires source code. *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.

None of these previous work use any form of learning technique to adapt the program to runtime conditions, as Table 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.

6 Conclusion

This paper has presented Astro, a program scheduler for big.LITTLE architectures. 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. 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 for parallel programs.

References

[1] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.* 23, 2 (2011), 187–198.

[2] Rajkishore Barik, Naila Farooqui, Brian T. Lewis, Chunling Hu, and Tatiana Shpeisman. 2016. A Black-box Approach to Energy-aware Scheduling on Integrated CPU-GPU Systems. In *CGO*. ACM, 70–81.

[3] Tarsila Bessa, Pedro Quintão, Michael Frank, and Fernando Magno Quintão Pereira. 2016. JetsonLeap: A Framework to Measure Energy-Aware Code Optimizations in Embedded and Heterogeneous Systems. In *SBLP*. Springer, 16–30.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*. ACM, 72–81.

[5] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. 2000. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. *IEEE Micro* 20 (November 2000), 26–44. Issue 6.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*. IEEE, 44–54.

[7] Hongsuk Chung, Munsik Kang, and Hyyun-Duk Cho. 2012. *Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology.* Technical Report. Samsung.

[8] Jason Cong and Bo Yuan. 2012. Energy-efficient Scheduling on Heterogeneous Multi-core Architectures. In *ISLPED*. ACM, 345–350.

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *TOPLAS* 9, 3 (1987), 319–349.

[10] Francisco Gaspar, Luis Taniça, Pedro Tomás, Aleksandar Ilic, and Leonel Sousa. 2015. A Framework for Application-Guided Task Management on Heterogeneous Embedded Systems. *ACM Trans. Archit. Code Optim.* 12, 4, Article 42 (Dec. 2015), 25 pages.

[11] Dominik Grewe and Michael FP O'Boyle. 2011. A static task partitioning approach for heterogeneous systems using OpenCL. In *Compiler Construction*. Springer, 286–305.

[12] A. Jain, M. A. Laurenzano, L. Tang, and J. Mars. 2016. Continuous shape shifting: Enabling loop co-optimization via near-free dynamic code rewriting. In *MICRO*. ACM/IEEE, 1–12.

[13] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2012. Bottleneck Identification and Scheduling in Multithreaded Applications. In *ASPLOS*. ACM, 223–234.

[14] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE, 75–88.

[15] Etienne Le Sueur and Gernot Heiser. 2010. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *HotPower*. USENIX Association, Berkeley, CA, USA, 1–8.

[16] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *MICRO*. ACM, 45–55.

[17] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. G. Dreslinski, T. F. Wenisch, and S. Mahlke. 2016. Exploring Fine-Grained Heterogeneity with Composite Cores. *IEEE Trans. Comput.* 65, 2 (2016), 535–547.

[18] Christos Margiolas and Michael F. P. O'Boyle. 2016. Portable and Transparent Software Managed Scheduling on Accelerators for Fair Resource Sharing. In *CGO*. ACM, 82–93.

[19] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. 2017. DawnCC: Automatic Annotation for Data Parallelism and Offloading. *TACO* 14, 2 (2017), 13:1–13:25.

[20] Rajiv Nishtala, Paul M. Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads. In *HPCA*. IEEE, 409–420.

[21] Cedric Nugteren and Henk Corporaal. 2014. Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs. *TACO* 11, 4 (2014), 35:1–35:25.

[22] Vinicius Petrucci, Michael A Laurenzano, John Doherty, Yunqi Zhang, Daniel Mosse, Jason Mars, and Lingjia Tang. 2015. Octopus-man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. In *HPCA*. IEEE, 246–258.

[23] Guilherme Piccoli, Henrique N. Santos, Raphael E. Rodrigues, Christiane Pousa, Edson Borin, and Fernando M. Quintão Pereira. 2014. Compiler Support for Selective Page Migration in NUMA Architectures. In *PACT*. ACM, New York, NY, USA, 369–380.

[24] Gabriel Poesia, Breno Guimaraes, Fabricio Ferracioli, and Fernando Magno Quintao Pereira. 2017. Static Placement of Computation on Heterogeneous Devices. In *OOPSLA*. ACM, 1–18.

[25] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. 2009. Thread Motion: Fine-grained Power Management for Multi-core Systems. In *ISCA*. ACM, 302–313.

[26] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *SOSP*. ACM, 49–68.

[27] Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. 2014. Price Theory Based Power Management for Heterogeneous Multi-cores. In *ASPLOS*. ACM, 161–176.

[28] Richard S. Sutton and Andrew G. Barto. 1998. *Introduction to Reinforcement Learning* (1st ed.). MIT Press, Cambridge, MA, USA.

[29] Lingjia Tang, Jason Mars, Wei Wang, Tanima Dey, and Mary Lou Soffa. 2013. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. In *ASPLOS*. ACM, 89–100.

[30] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *ISCA*. IEEE, 213–224.

[31] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *TACO* 9, 4 (2013), 54:1–54:23.

[32] Matthew J. Walker, Stephan Diestelhorst, Andreas Hansson, Anup K. Das, Sheng Yang, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2016. Accurate and Stable Run-Time Power Modeling for Mobile and Embedded CPUs. *TCAD* 36, 1 (2016), 106–119.

[33] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh. 2015. Neural acceleration for GPU throughput processors. In *MICRO*. 482–493.

[34] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In *ASPLOS*. ACM, 545–559.