Mapping Computations in Heterogeneous Multicore Systems with Statistical Regression on Program Inputs

Junio Cezar Ribeiro da Silva, Lorena Leão, Vinicius Petrucci, Abdoulaye Gamatié, Fernando Magno Quintão Pereira

Mapping Computations in Heterogeneous Multicore Systems with Statistical Regression on Program Inputs

JUNIO CEZAR RIBEIRO DA SILVA, Universidade Federal de Minas Gerais
LORENA LEÃO, Universidade Federal de Minas Gerais
VINICIUS PETRUCCI, Universidade Federal da Bahia and University of Pittsburgh
ABDOULAYE GAMATIÉ, LIRMM, Univ. Montpellier, CNRS
FERNANDO MAGNO QUINTÃO PEREIRA, Universidade Federal de Minas Gerais

A hardware configuration is a set of processors and their frequency levels in a multi-core heterogeneous system. This paper presents a compiler-based technique to match functions with hardware configurations. Such a technique consists in using multivariate linear regression to associate function arguments with particular hardware configurations. By showing that this classification space tends to be convex in practice, this paper demonstrates that linear regression is not only an efficient tool to map computations to heterogeneous hardware, but also an effective one. To demonstrate the viability of multivariate linear regression as a way to perform adaptive compilation for heterogeneous architectures, we have implemented our ideas onto the Soot Java bytecode analyzer. Code that we produce can predict the best configuration for a large class of Java and Scala benchmarks running on an Odroid XU4 big.LITTLE board; hence, outperforming prior techniques such as ARM's GTS and CHOAMP, a recently released static program scheduler.

CCS Concepts: •**Software and its engineering** → **Compilers;** •**Computing methodologies** → *Parallel programming languages; Machine learning;*

ACM Reference format:
Junio Cezar Ribeiro da Silva, Lorena Leão, Vinicius Petrucci, Abdoulaye Gamatié, and Fernando Magno Quintão Pereira. 2021. Mapping Computations in Heterogeneous Multicore Systems with Statistical Regression on Program Inputs. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2021), 35 pages.
DOI: 10.1145/3478288

1 INTRODUCTION

Several modern computer architectures combine, into a single device, fast and slow processing cores able to execute the same set of instructions (Kumar et al. 2004; Orgerie et al. 2014; Singh et al. 2020). Fast cores perform computations efficiently, but are power-hungry; Slow cores show the inverse behavior. The ARM big.LITTLE design, ubiquitous in smartphones, is an example of such technology (Hähnel and Härtig 2014). As an illustration, the Apple A14, launched in September of 2020, has two high-performance cores called *Firestorm* and four energy-efficient cores called *Icestorm*, all running the ARMv8.4-A instruction set (Gurman et al. 2020). The combination of heterogeneous processors featuring multiple frequency levels gives programmers many configurations to choose from when running their applications. However, performing this choice is challenging (Azhar et al. 2019; Nejat et al. 2020; Nishtala et al. 2017; Paul et al. 2020; Petrucci et al. 2015).

 1539-9087/2021/1-ART1 $15.00
DOI: 10.1145/3478288

A recent solution to this problem is CHOAMP, a compilation technique invented by Sreelatha *et al.* (Sreelatha et al. 2018). CHOAMP uses supervised machine learning to map program functions to the configuration that best fits them. Sreelatha *et al.* capture characteristics of the target architecture's runtime behavior. From this knowledge, they predict the ideal configuration to a program, given its syntactic characteristics. Sreelatha *et al.*'s approach is fully static: interventions on the program remain confined into the compiler, and no extra runtime support is required from the hardware. This approach has been made popular by Shelepov *et al.* (Shelepov et al. 2009)'s HASS system, a scheduler for same-ISA heterogeneous systems.

We observe that CHOAMP and HASS share a fundamental shortcoming: they do not consider program inputs when performing mapping decisions. As we explain in Section 2, there exist programs for which the best hardware configuration for a given function varies depending on the function's inputs. We make the case that inputs are key to determine good mappings between programs and configurations supported by the evidence that such mappings do not necessarily converge to a single, ideal configuration, as the size of inputs grows.

Our Solution. In this paper, we introduce a compilation approach to map program parts to hardware configurations. Our technique explicitly takes function inputs into consideration when deciding which hardware configurations to use. Input-based code optimizations are not a new idea, as we explain in Section 5.3; however, to the best of our knowledge, this paper is the first to use related techniques to find optimized hardware configurations for programs running on big.LITTLE architectures. As we discuss in Section 3, our idea is based on statistical regression *on the values of program inputs*. Given a function *foo*, a collection of its inputs $\{t_1, t_2, \ldots, t_m\}$ available for training, plus a set of hardware configurations $\{h_1, h_2, \ldots, h_n\}$, we run $foo(t_i), 1 \leq i \leq m$, onto a sample of the configuration space $\{h_j \mid 1 \leq j \leq n\}$. Training gives us the ideal configuration for each input, in terms of a measurable goal, such as runtime or energy consumption. When producing code for *foo*, we augment its binary representation with this knowledge to predict the best configuration for potentially unseen inputs.

Our Results. We have implemented our technique in Soot (Vallée-Rai et al. 1999), a bytecode optimizer, and have evaluated it onto an Odroid XU4 big.LITTLE architecture. Soot lets generate code that, at runtime, changes the hardware configuration per program function, based on training knowledge. We call this code generator the Jinn-C compiler, a tool that reads and outputs Java bytecodes. As we explain in Section 4, we have evaluated Jinn-C on the Program Based Benchmark Suite (Shun et al. 2012) used by Acar *et al* (Acar et al. 2018), and on programs from Renaissance (Prokopec et al. 2019). We have evaluated Jinn-C with two objective functions: runtime and energy consumption. We measure energy for the entire board using physical probes, following Bessa *et al*'s methodology (Bessa et al. 2017). Below we summarize the benefits of our solution in the context of the existing literature:

- **Simple**: we show that, for typical parallel code, the value of scalar inputs and the size of aggregate inputs yield useful information to feed linear regression models, because the function that maps these values to optimized hardware configurations form a convex space.
- **Effective**: in most of our benchmarks, ten input sets are already sufficient to let us train a predictor to a high level of accuracy. Variety is, of course, important: the more different the inputs we have, the more accurate the predictions we perform.
- **Efficient**: our approach does not require active runtime monitoring. Inputs must be evaluated upon function invocation, and only then. Evaluation amounts to one vector-matrix multiplication, and is proportional to the number of function arguments (not to their sizes) times the number of valid hardware configurations. Training is costly, but this cost is paid offline, before programs run in production mode.

- **Semi-Automatic**: our approach requires minimum user interference. Developers annotate which functions must be adapted. No further interventions are necessary.
- **Easily-deployable**: our solution does not require runtime monitoring; thus, it can be deployed in any modern hardware and operating system, independent on them providing advanced runtime capability such as performance counters. We require only the capability to change the hardware configuration at runtime.

Convex Space. This journal paper extends an earlier conference paper of ours (da Silva et al. 2020), with more experiments and a more detailed explanation of our technique. The most important addition over that first publication is an empirical demonstration that statistical functions that map program inputs to hardware configurations tend to be convex. As we show in Section 4.4, convexity means that by varying only one function argument, while fixing the others, the ideal configuration is unlikely to oscillate, for instance, going from h_i to h_j and then back to h_i. We have not found one single case, among 18 benchmarks, each with 14 different input sets, where such oscillations could be discerned. The consequence of this observation, discussed in Section 4.4.2, is that derivative-based search methods are expected to converge to an optimal result, and linear regression is capable to accurately predict this optimum. We emphasize that this observation shows an intuitive tendency that has been verified in the benchmarks used in this paper, for the inputs available to them. Although it is easy to construct benchmarks whose space of (inputs $\times$ ideal hardware configurations) is not convex, we believe that such benchmarks are unlikely to emerge as part of well-known algorithms, as we explain in Section 4.4.

2 OVERVIEW

The term *hardware configuration* is used with different meanings by different researchers, thus we shall restrict ourselves to the following definition:

Definition 2.1 (Hardware Configuration). Let $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ be a set of *n processors*, and let *Freq* be a function that maps each processor to a list of possible *frequency levels*. A hardware configuration is a set of pairs $h = \{(\pi, f) \mid \pi \in \Pi, f \in Freq(\pi)\}$. If $(\pi_i, f_j) \in h$, for some $f_j \in Freq(\pi_i)$, then processor π_i is said to be *active* in h with frequency f_j, otherwise it is said to be *inactive*.

Example 2.2 (Hardware Configuration). The HardKernel Odroid XU4 has four big cores $\{b_0, b_1, b_2, b_3\}$ and four LITTLE cores $\{L_0, L_1, L_2, L_3\}$. Big cores have 19 frequency levels $\{200MHz, 300MHz, \ldots, 1.9GHz, 2.0GHz\}$. LITTLE cores have 14 $\{200MHz, \ldots, 1.5GHz\}$. This SoC supports any number of active processors; however, big cores must always use the same frequency level. The same is true for LITTLE cores. An example of hardware configuration is $\{b_0, b_2\} \times 2.0GHz$, $\{L_1, L_2, L_3\} \times 1.3GHz$.

The notion of hardware configuration leads to an interesting problem in the field of *adaptive compilation*. In the words of Cooper *et al* (Cooper et al. 2005), "*an adaptive compiler uses a compile-execute-analyze feedback loop to find the combination of optimizations and parameters that minimizes [optimizes] some performance goal, such as code size or execution time*". In this paper we are interested in solving the adaptive compilation problem introduced by Definition 2.3.

Definition 2.3. Input-Aware Scheduling in Single-ISA Heterogeneous Architectures (Isha) **Input**: a function F, its input i, a set of hardware configurations $H = \{h_1, \ldots h_n\}$, and a cost function $O^i_F : H \mapsto \mathbb{R}$, which determines the cost of running F with input i on configuration $h \in H$. Examples of cost functions include runtime, energy, energy-delay product, throughput, etc. **Output**: a configuration $h \in H$ that minimizes O^i_F.

As Section 3 explains, we insert code at the entry point of functions to switch the hardware configuration; hence, providing a solution to Isha. From that region on, the program will run in

the chosen configuration, until its execution flow finds the entry point of another function that has also been instrumented. Any function might run at any hardware configuration, for although we target a heterogeneous architecture, big and LITTLE cores run the same instruction set. Notice that moving computations across cores bears a cost: cores do not share L1 caches, and big and LITTLE clusters do not share the L2 cache, although all the cores share the main memory (Weber et al. 2017, Sec.2.2). This cost is embedded in the numbers that we report in Section 4. In other words, the speedups or regressions that we observe experimentally already include the cost of warming up caches due to switching computation between cores.

2.1 Program Inputs and Hardware Configuration

Compilers, such as Gcc or Clang, do not try to capitalize on differences between cores when producing binary programs: the same executable runs in both cores. Nevertheless, we know of research artifacts that take these differences into consideration; for example, CHOAMP is a recent technique in this direction (Sreelatha et al. 2018). CHOAMP matches program features, such as branches, barriers, reductions and memory access operations with the ideal configuration for each function. CHOAMP has been evaluated on the OpenMP version of the NAS benchmark suite (Bailey et al. 1991) with great benefit, producing code that was 65% more energy-efficient than the default Linux scheduler.

After CHOAMP trains a regression model, the same core configuration decision applies for a function, regardless of its actual inputs. This shortcoming of purely static approaches has been well-known. Quoting Nie and Duan: "*since the properties they have collected are based on the given input set, those offline profiling approaches are hard to adapt to various input sets and therefore will drastically affect the program performance*" (Nie and Duan 2012). We corroborate this observation and show that it is possible to find different programs for which the ideal hardware configuration varies according to their inputs. Example 2.4 illustrates this finding with an actual experiment.

Example 2.4. Function task in Figure 1 inserts into a global map all the values stored in a stream. Values are associated with a key, whose size varies according to the formal parameter keySize. Task has a synchronized block; hence, it can be safely executed by multiple threads. The number of threads is an *implicit input*. These three values: size of input stream, size of keys, and number of threads, form a three-dimensional space, which Figure 1 illustrates. The ideal hardware configuration for task varies within this space. Figure 2 illustrates this variation for 3×25 different input sets. The notation XbYL denotes X big cores, and Y LITTLE cores. In this experiment, we have set *Freq(b) = 1.8GHz*, for any big core *b*, and *Freq(L) = 1.5GHz*, for any LITTLE core *L*.

The construction of a key, at line 5 of Figure 1 is a CPU-heavy, synchronization-free task. The larger the key, the more incentive we have to use the big cores. However, the updating of globalMap at line 9 is a synchronization-heavy task: the more synchronization we have, the heavier is the penalty on the big cores, relative to the LITTLE ones. Indeed, as already observed by Kim *et al.* (Kim et al. 2014), context switches are more expensive in the big than in the LITTLE cores. Reasons for this heavier penalty include the larger pipelines used in the big cores. Whereas the ARM A15 (big) core features a pipeline with 15 integer and up to 25 floating point stages, the A7 (LITTLE) core features a pipeline with only 8 stages (Weber et al. 2017, Sec.2.2). Memory accesses are also more expensive (relative to execution cycles) on the faster cores: L2 latency for big cores is 21 cycles while for LITTLE cores it is 10 (Greenhalgh 2011). Furthermore, the larger the input streams, the more often we access the synchronized region between lines 7 and 10 of Figure 1. We can observe results similar to those seen in Example 2.4 in algorithms like Integer Sort, a benchmark used by Sreelatha *et al.* (Sreelatha et al. 2018), which we re-evaluate in Section 4.

```
// The number of threads is a hidden input
void task(Stream<Value> s, long keySize) {
  while (!s.empty()) {
    // Get a key of the proper size:
    BigInteger key = getNextKey(keySize);
    // Use key to update globalMap
    synchronized(globalMap) {
      Value value = s.next();
      globalMap.put(key, value);
    }
  }
}
```

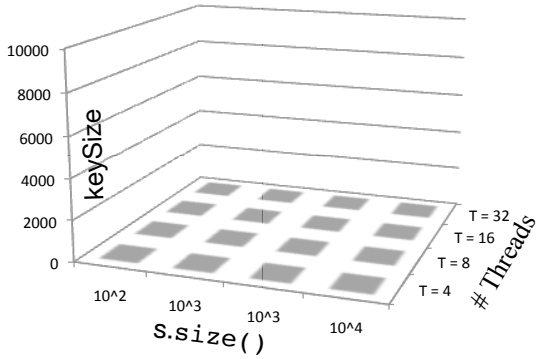

Fig. 1. (Left) A program whose behavior can be computation-heavy or synchronization-heavy, depending on its inputs. (Right) The search space formed by the program's inputs. (Taken from da Silva et al. (2020)).

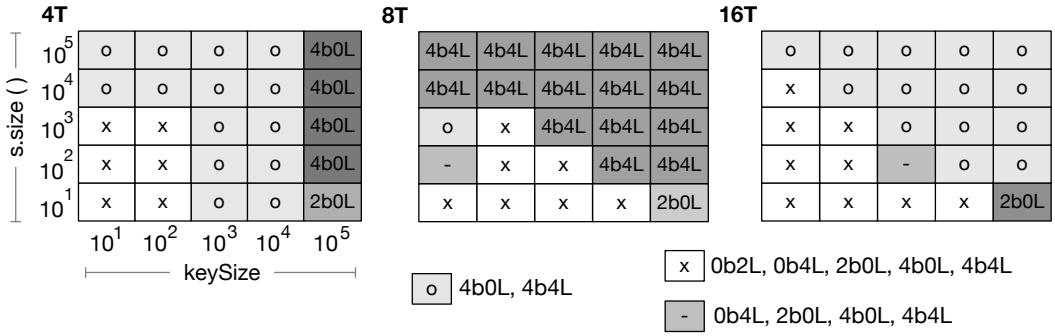

Fig. 2. Ideal configuration for different parameters of the TASK function (Fig. 1), for 4, 8 and 16 threads, measured on an Odroid XU4 with the *userspace* governor. Default configuration is 4b4L. Names in boxes indicate the best configuration(s) for that input. 'X' indicates setups with three or more configurations tied as best. To produce these charts, we followed a methodology to be described in Section 4.4. Even considering 4 threads, there is benefit to enable more than four processors, as the Java virtual machine creates threads for garbage collection and JIT compilation, for instance (Taken from da Silva et al. (2020)).

Notice that allowing a program to use more threads does not necessarily mean that this program will draw larger benefit from configurations with more cores. This counterintuitive behavior happens because more cores might provoke more thread conflicts in synchronization-heavy settings. Example 2.5 elaborates on this observation.

Example 2.5. Figure 2 shows that when the TASK function runs with eight threads, the configuration with full resources (4b4L) is often chosen. However, once 16 threads are employed to generate keys, sometimes not all the cores are used in the fastest configuration. Figure 3 provides numbers to explain this behavior. For small key sizes, collisions between threads happen frequently. In this setting, 1B0L is the best configuration, regardless of the number of threads considered (even though running times are short, the confidence level of this result is above 99%). Once key size increases, time spent on synchronization becomes less important, and extra cores start to be advantageous. Notice that past certain point (after keys with 10,000 bits), just-in-time compilation triggers, and the

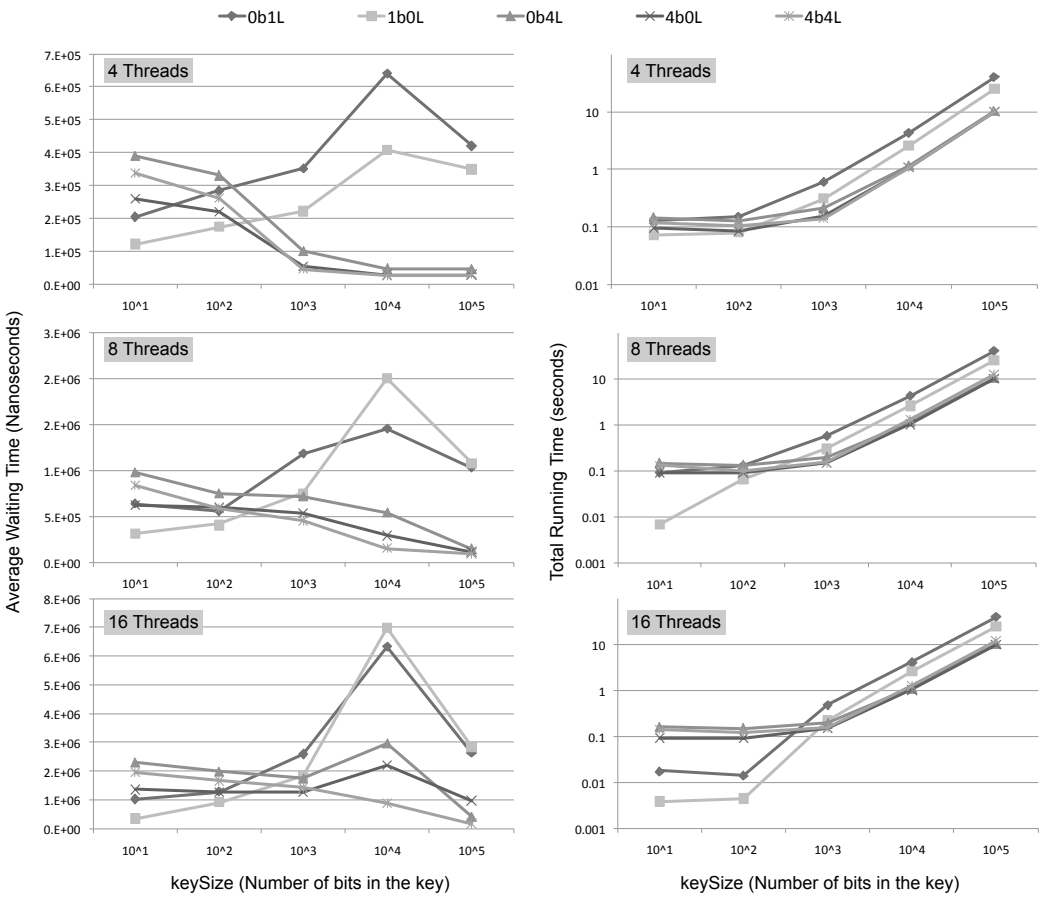

Fig. 3. Average waiting time and total running time of function Task (Figure 1) considering different key sizes, and different numbers of threads. To perform this experiment, we have modified Task to use Java's ReentrantLock class. This new version of Task is slower, due to the need to log events; hence, we present total running times to give the reader some perspective on the program's behavior.

average waiting time decreases by approximately 50%. Yet, the total application time still increases, due to the excess of calculations required by the large keys.

Example 2.5 shows that two facts increase the average time that threads spent on critical sections: the total number of working threads that the program uses, and the total number of cores where these threads execute. Therefore, adding more resources to thread rich programs can cause them to run more slowly due to extra conflicts. Such complex tradeoffs make the problem of choosing ideal hardware configurations a non-trivial endeavor.

2.2 Accounting for Energy Efficiency

Choosing good hardware configurations becomes more challenging, once we consider energy as a dimension of efficiency. Low-frequency cores tend to be more power-efficient than high-frequency processors; hence, there is incentive to use them to save energy. However, low-frequency cores

tend to take longer to finish tasks; possibly, using more energy to perform a job. This observation is critical in battery-powered devices, such as smartphones. The next example analyzes such power-performance tradeoffs. In this experiment, we are measuring the actual power consumed in the entire board, which includes not only its CPUs but also its peripherals, such as memory and cooling. To this end, we use the apparatus described by da Silva et al. (2018), which samples power at 20KHz.

Example 2.6. We have used the power measurement apparatus shown in Figure 4(a) to plot runtime and energy consumption for the function TASK earlier seen in Fig. 1, considering two different input sets. Figure 4(b) shows the power profile of TASK for a synchronization-free set of inputs (top) and for a synchronization heavy set (bottom). Following Silva *et al.* (da Silva et al. 2018), we call the chart relating runtime and energy a *constellation*. The constellation in Figure 4(c) shows the behavior of TASK for the *synchronization-free* input. In this case, the size of keys is very large, and the number of insertions in the GLOBALMAP is very low, thus conflicts seldom happen. On the other hand, if we make the size of keys very small, and the size of the stream very large, then we obtain a rather different constellation, which Figure 4(d) outlines. This constellation shows how TASK performs in a *synchronization-heavy* environment.

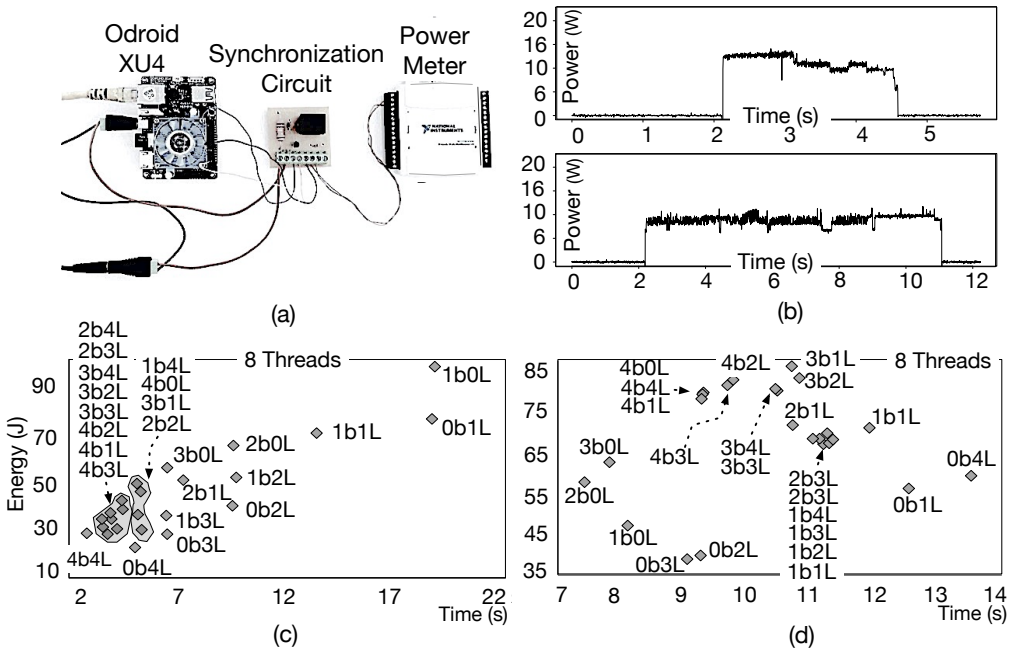

Fig. 4. (a) The energy measurement apparatus adopted in this paper. (b) Power charts for configuration 4b4L with synchronization-free inputs (top) and synchronization-heavy inputs (bottom). (c) Time vs energy constellation for the synchronization-free input set. (d) The constellation for the synchronization-heavy input set. Big cores run at 2.0*GHz* and LITTLE cores run at 1.5*GHz*. (Taken from da Silva et al. (2020))

Example 2.6 shows how changes in inputs modify the disposition of hardware configurations in the constellations. The best energy and time configuration in the CPU-heavy setting, 4b4L, is one of the worst configurations in the synchronization-heavy setting. Such dramatic changes make

it very difficult for a completely static approach to find energy-efficient hardware configurations for program parts. The size and type of program inputs are only known at runtime. To handle the lack of information at compile time, existing prior work (David et al. 2014; Nishtala et al. 2017; Petrucci et al. 2015) resorts to online monitoring; however, this may pose a potential overhead on the system as the number of programs and hardware configuration increase.

3 MULTIVARIATE LINEAR REGRESSION OF PROGRAM INPUTS

We apply statistical regression on the arguments of a function to determine the ideal hardware configurations for different inputs of that function. The pipeline in Figure 5 summarizes how code is modified in order to implement this idea. To ease our presentation, we shall be using source code in our examples, as seen in Figure 5. However, our solution works at the Java bytecode level and our interventions happen within the compiler –more precisely in the program's intermediate representation. Our techniques could have been applied directly onto Java sources or even onto a different programming language. Nevertheless, working at the bytecode level lets us optimize programs written in different languages that run on the Java Virtual Machine. In Section 4 we shall validate our techniques using Java and Scala benchmarks.

3.1 Multiple Linear Regression

The key ingredient of our work is the application of multivariate regression onto the arguments of functions. Linear regression empowers a prediction model that matches function parameters with resource-efficient hardware configurations. We extend our regression model to a multivariate system, as the output is a vector (of ideal configurations). In this model, we define a number of *dependent* variables, grouped into a matrix C, plus a number of *independent* variables, grouped into a matrix A. The goal of the regression model is to determine a matrix Θ that approximates the product $C = \sigma(A\Theta)$. In this case, σ is the *softmax* function, applied on the rows of the matrix product $A\Theta$. If Z is a $1 \times n$ vector, e.g., a line of $A\Theta$, then $\sigma(Z)$ is also an $1 \times n$ vector, whose j^{th} element is defined as: $\sigma(Z)_j = e^{Z_j} / \sum_1^n e^{Z_k}$. The softmax function receives a vector of real numbers, and produces a vector of the same size normalized over a probability distribution. Every $\sigma(Z)_j$ is a number between 0.0 and 1.0, and the sum of all the elements within $\sigma(Z)$ is 1.0.

Example 3.1. Figure 6 presents a formula for regression involving a function f that has three formal parameters. We assume a universe of five valid configurations (0B1L, 1B0L, 1B1L, 2B0L and 2B1L). The frequency level is immaterial for this example: big and LITTLE cores run at a certain fixed frequency, which is not necessarily the same for the two clusters. In this example we have a training set containing four samples, each one representing a different invocation of function f, ideally with different actual arguments.

The matrix A of independent variables. As Example 3.1 illustrates, the matrix A encodes known values of function arguments. These values are called the *training set* of our regression. If we are analyzing a function with n arguments, and our training set contains m function calls, then A is a matrix with m rows and $n+1$ columns. The extra column is the all-ones vector 1^m, which represents *intercepts* – constants that allow us to handle a scenario in which the training set contains only null values. This all-ones column is the first column of matrix A in Figure 6.

Example 3.2. Figure 7 shows how ten different samples of function Task, from Fig. 1, are organized into a matrix A of independent variables.

The matrix C of dependent variables. C represents the ideal hardware configuration for each input in the training set. If we admit k valid configurations, and our training set has m samples, then C is an $m \times k$ matrix. Each line of C is a unitary vector e_i, which has all the components set to

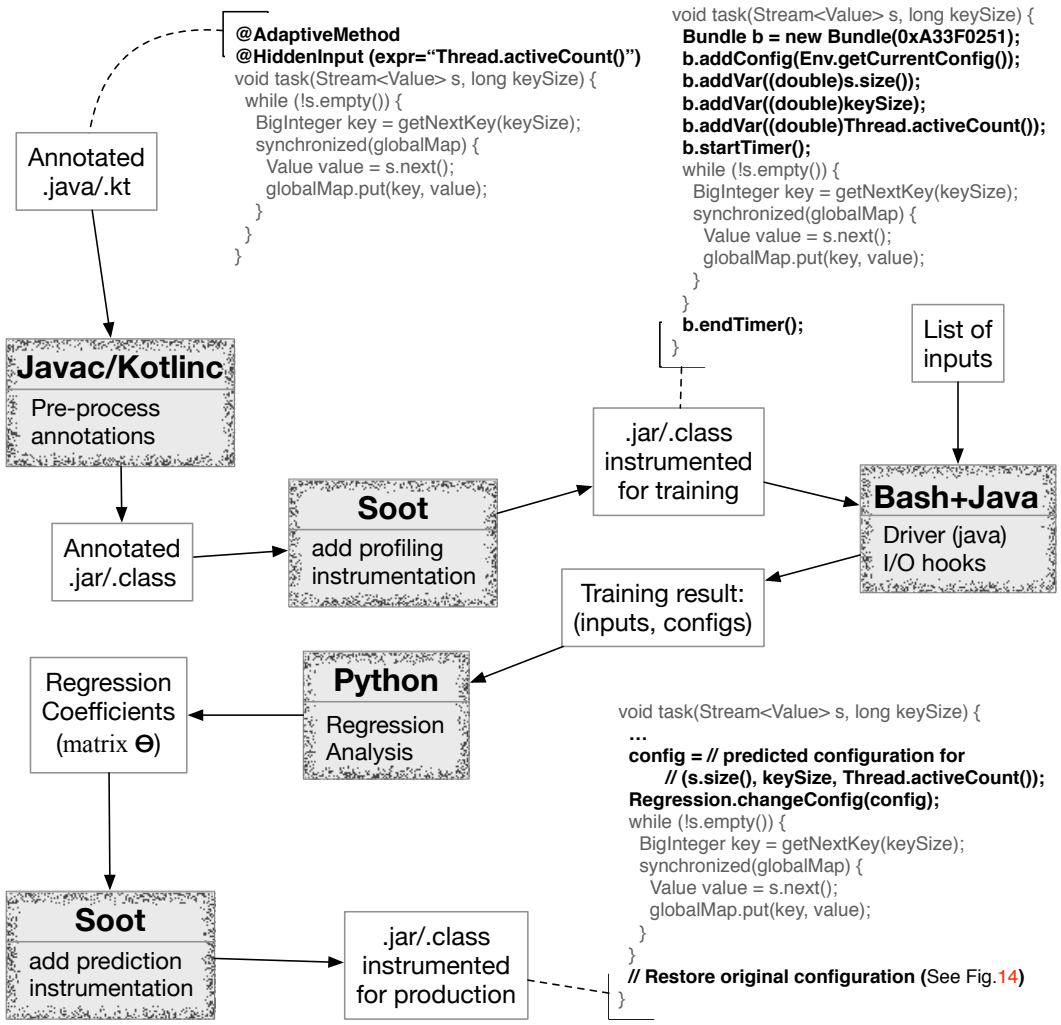

Fig. 5. The execution pipeline of Jinn-C, as shown in our previous work (da Silva et al. 2020).

zero, except its i^{th} index, which is set to one. If $C_{ji} = 1$, then i is the best configuration for input j. The next example illustrates these notions with actual data.

Example 3.3. Figure 8 reuses the ten samples seen in Example 3.2 to build the matrix of dependent variables. This matrix has one line per sample, and one column per configuration of interest. This example considers only 10 out of the 4,654 possible configurations of the Odroid XU4 board. This need for bounding the search space might prevent us from discovering good optimization opportunities; however, it ensures that our methodology is practical. Section 4 discusses the criteria used to build the search space of allowed configurations.

Finding the parameter matrix Θ. The problem of constructing a predictor based on multivariate linear regression consists in finding a matrix Θ that maximizes correct predictions on the training set. The underlying assumption is that if Θ approximates the behavior of the training set, then it is

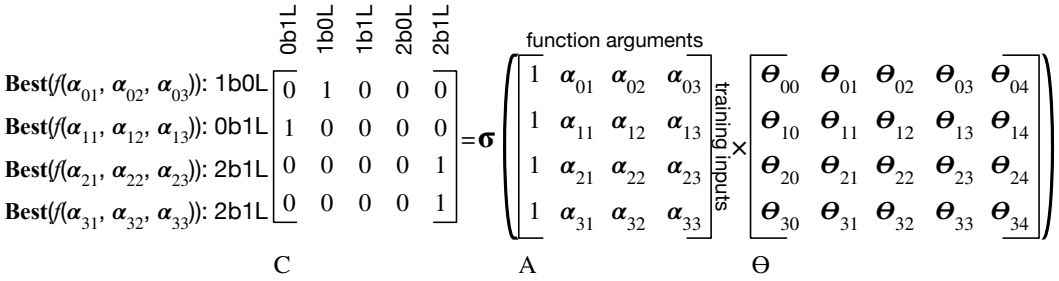

Fig. 6. Formula to train a 3-ary function $f(\alpha_0, \alpha_1, \alpha_2)$. The goal of multivariate linear regression is to find the coefficients Θ that approximate the product $C = \sigma(A\Theta)$. Training set contains four samples.

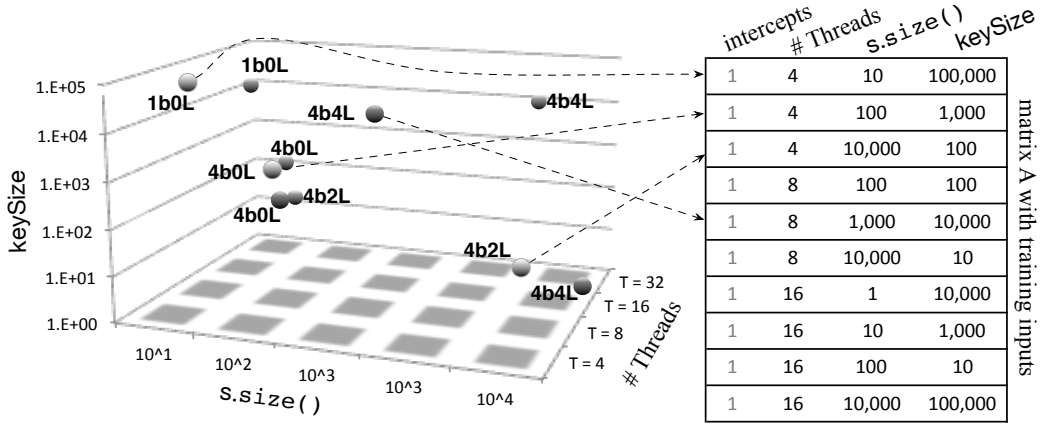

Fig. 7. Training set for the Task method (Fig. 1). The table on the right is matrix A of independent variables (taken from da Silva et al. (2020)).

likely to yield good results on the test set. There exist efficient techniques to find Θ – *gradient descent* being the best well-known (Cauchy 1847). Our model is based on multivariate linear regression; thus, searches happen over a linear space. By a linear search space, we mean that, for each element (i, j) in C, we have that: $C_{ij} = \Theta_{0j} + \alpha_{i1}\Theta_{1j} + \ldots + \alpha_{im}\Theta_{mj}$. In other words, non-linear expressions such as $\alpha_{ip}\alpha_{iq}$ do not contribute to the value of C_{ij}. Because our model involves only searches over a linear space, gradient descent converges to a global optimum (Shi 2004).

Example 3.4. Figure 9 shows a possible matrix Θ that gradient descent finds for the Task function, when given the training set seen in Figures 7 and 8. Once we apply the softmax function onto the product $A\Theta$ we obtain a predicted matrix C', which approximates the target matrix C, e.g., $C' = \sigma(A\Theta)$. Each line of C' adds up to[1] 1.00. The largest value in each line i of C' determines the ideal configuration for the input set A_i. The matrix Θ seen in Figure 9 led us into a C' that correctly matches the target C in all but two inputs. Some misses are expected. If we resort to more complex regression models, for instance, with non-linear components, then we might find a Θ that correctly

[1]We are using only two decimal digits; hence, rounding errors prevent us from obtaining 1.00 in every line.

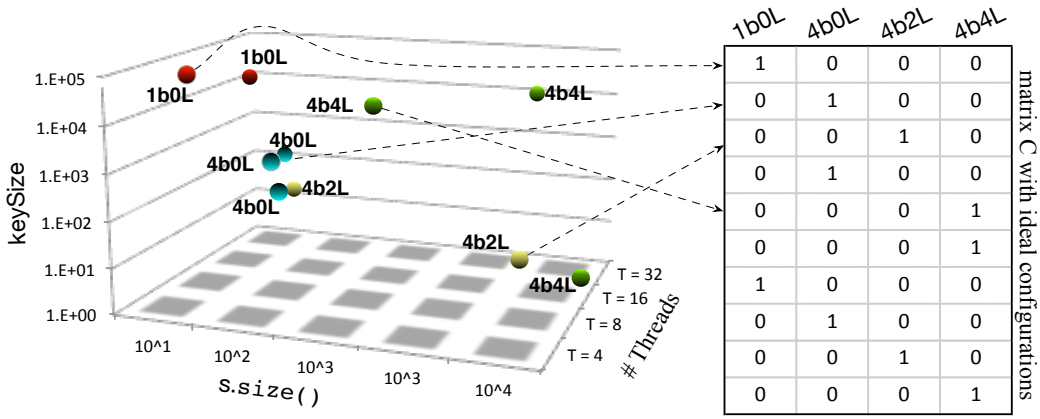

Fig. 8. Matrix of independent variables built for ten different invocations of function Task in Figure 1 (taken from da Silva et al. (2020)).

predicts every row of C. However, this matrix, which fits too well the training set, might not yield good predictions on unseen inputs.

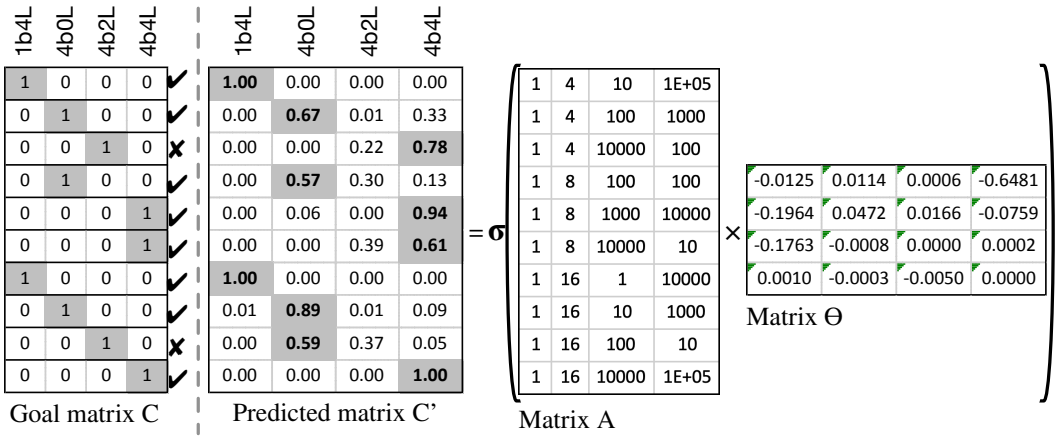

Fig. 9. The result of multivariate linear regression produced by the training set seen in Examples 3.2 and 3.3 (taken from da Silva et al. (2020)).

Using Θ to carry out predictions. The single output of regression is the matrix Θ. Once we find a suitable Θ, we can use it to predict the ideal configuration for inputs that we have not observed during training. To this effect, as we shall better explain in Section 3.3, the constants in Θ are hardcoded into the binary text that we generate for the function f under analysis. If f is invoked with a set of inputs A_i, then the expression $\sigma(A_i\Theta)$ is computed on-the-fly. The result of this evaluation determines the active configuration.

Example 3.5. Figure 10 uses the matrix Θ found in Figure 9 to guess the best configuration for four unseen input sets. These inputs appear as dark spheres in Figure 10. In this example, Θ

correctly predicts the ideal configuration for three out of four samples. In one case, the last input in Figure 10, we wrongly predict the best configuration as 4b2L, whereas empirical evidence suggests that it should be 4b4L.

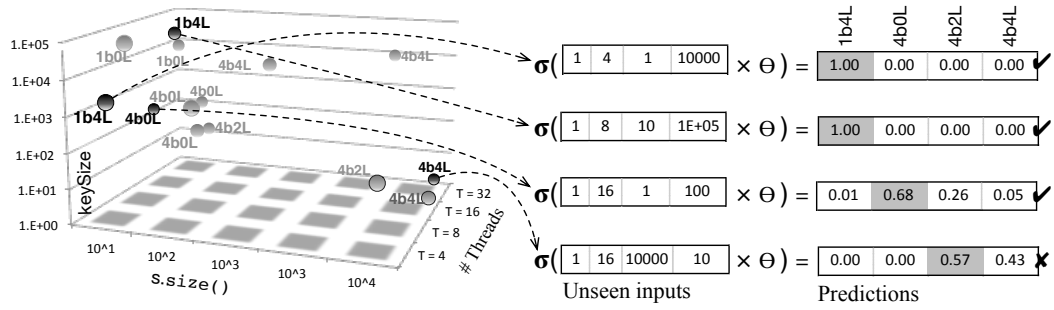

Fig. 10. The matrix Θ (see Figure 9) used to predict ideal configurations for unseen inputs. Light-grey points form the training set. Inputs in the test set are dark-grey. (taken from da Silva et al. (2020))

3.2 Engineering the Training Phase

Users of Jinn-C specify which methods must be optimized. For each one of these methods, Jinn-C singles out its inputs, and instrument them to produce regression data. Are considered inputs: the formal parameters of methods, the global variables used within these methods and the number of active threads. Regression data consists of the size of these inputs. The technique used to obtain these sizes depends on the type of input. Currently, we use the following heuristics:

Primitive types: the size of a primitive type is its own value.
Wrappers: types such as Integer or Double, which work as wrappers of primitive types, give us a size through their value() methods, e.g., intValue() for Integer.
Arrays and *Strings*: we derive the size of such types via the length property.
Collections: we derive the size of collections by invoking their size() method.
Other classes: we search within the declaration of the type, or in any of its super-types, for a method called size(); otherwise, we search for a property called length. If such names are not to be found, an error ensues.

Example 3.6. Figure 11 shows two instrumented programs. Profiling code is inserted in the programs' intermediate representation –source code is used only for readability. Instrumentation is performed by a singleton object Instrumenter, which stores "bundles" of data. Each bundle contains an identifier, a hardware configuration, the independent variables of the adaptive method, and the runtime for those variables. Identifiers map methods to bundles. Multiple invocations of the same method will produce one bundle per call.

3.2.1 Profiling, Logging and Training. Currently, we use a profiling infrastructure written as a combination of Java code and bash scripts. The part implemented in Java consists of a service that runs the program that we want to optimize in a controlled environment. This driver has two responsibilities: warming up the target program and changing hardware configurations before every profiling experiment.

```
void visit(final int NT) throws ... {
  Bundle b = new Bundle(0xFF4AC08D);
  b.addConfig(getCurrentConfig());
  b.addInt(visited.length); // array
  b.addInt(graph.size()); // class has size()
  b.addInt(NT); // primitive type
  Instrumenter.save(b);
  b.startTime();
  Vector<Visitor> bots = new Vector<Visitor>(NT);
  for (int i = 0; i < NT; i++) {
    bots.add(new Visitor(graph, i));
  }
  for (Visitor v : bots) { v.start(); }
  for (Visitor v : bots) { v.join(); }
  b.stopTime();
}
```

```
void count(final int START, final int END) {
  Bundle b = new Bundle(0xFF4AC08E);
  b.addConfig(getCurrentConfig());
  b.addInt(START); // primitive type
  b.addInt(END); // primitive type
  b.addInt(forkJoinPool.getActiveThreadCount());
  Instrumenter.save(b);
  b.startTime();
  for (int j = START; j <= END; j++) {
    SingleCounter aux = counters[elements[j]];
    synchronized (aux) {
      aux.value += 1;
    }
  }
  b.stopTime();
}
```

Fig. 11. Instrumented version of two programs. Grey code is from the original method. (Left) Breadth-first search. (Right) Sorting application. (taken from da Silva et al. (2020))

Jinn-C receives an annotated program P, a set of different inputs $I = \{\iota_1, \iota_2, \ldots, \iota_m\}$ of P, and a set of acceptable hardware configurations $H = \{h_1, h_2, \ldots, h_n\}$. It will test the program a predetermined number of times for each pair $(h, \iota), h \in H, \iota \in I$. The best configuration for each input ι is chosen among the most frequent winner. The objective function that determines the winner is configurable. Currently, we consider time, energy consumption and energy-delay product. In case of ties, we choose the configuration with the least resources. Resources are ordered according to the number of big cores, the number of LITTLE cores, the frequency of the big cores and the frequency of the LITTLE cores, in this sequence.

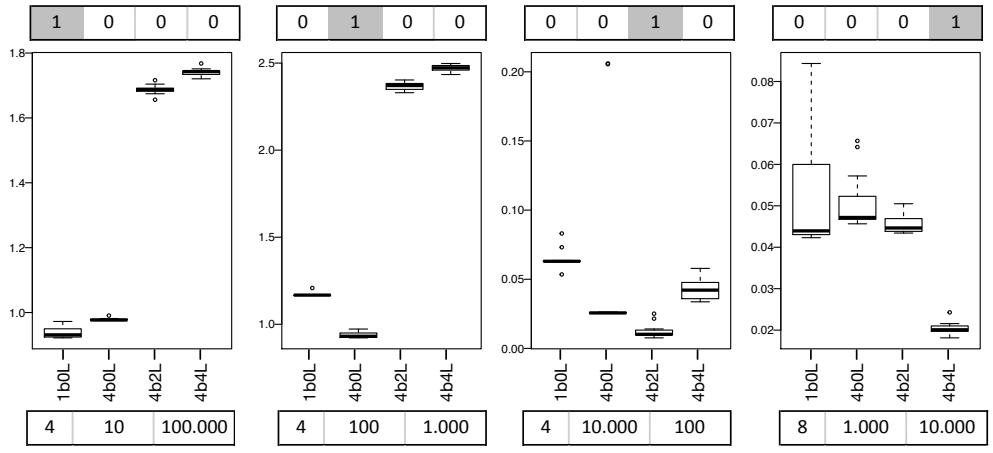

Fig. 12. Training output produced by the driver on a few inputs seen in Figure 7. Y-axis is runtime.

Example 3.7. Figure 12 shows a typical output produced during Jinn-C's training phase, considering runtime as the objective function. In this experiment, each pair formed by a hardware

configuration and an input is sampled ten times. Vectors at the bottom of Figure 12 are the inputs passed to function Task (Fig. 1). These vectors are the independent variables in the regression model. Vectors at the top of Figure 12 are the best configurations. These vectors will give us the dependent variables used in the regression.

Complexity of the Training Phase. Training involves running the target program in every valid hardware configuration with every available input. Therefore, the asymptotic complexity of this phase is determined by the number of valid hardware configurations, the number of input sets available to run the program and the complexity of the program itself. In other words, the instrumented program P will have to be profiled $|H| \times |I|$ times; H being the set of valid hardware configurations and I the set of available inputs. The third column of Table 1 (Page 17) reports training times observed when evaluating the benchmarks used in this paper. The longest time was 225 minutes; the shortest 21 minutes. Section 4.1 provides more details about these costs.

3.3 Generation of Adaptive Code

The product of training is a matrix Θ of floating-point numbers. Θ is hardcoded into the production code that we want to optimize. Such step happens in the phase labeled "add prediction instrumentation" in Figure 5. The instrumentation that we add into a function f of interest evaluates the expression $\sigma(A_i\Theta)$, where A_i is a $1 \times n$ vector. The size of A_i is one plus the number of inputs of the target function. The expression $\sigma(A_i\Theta)$ yields a $1 \times k$ vector of probabilities, whose elements add up to 1.0, where k is the number of hardware configurations considered as targets. The largest element within $\sigma(A_i\Theta)$ determines the next configuration that will be used during the current invocation of f. Therefore, the complexity of choosing a hardware configuration is proportional to the number of function arguments (n) times the number of valid hardware configurations (k). As we shall discuss in Section 4, this time tends to be too short to be of practical consequence.

Example 3.8. Figure 13 shows the production version of our running example, the function Task, originally seen in Figure 1. The dashed box outlines the code that we add to Task to change the current hardware configuration. We show, on the right of the figure, the key methods used to change and restore the current hardware configuration. The matrix Θ seen in the production version of function Task was found after training, as Example 3.4 explains.

4 EVALUATION

This section demonstrates the effectiveness of our technique when optimizing bytecodes that run on top of the Java Virtual Machine. To this end, we shall provide answers to the following research questions:

RQ1 – Training: what is the training time required by Jinn-C, and how does it compare to CHOAMP's?
RQ2 – Speed: what is the speed gain obtained by Jinn-C when compared to competing techniques?
RQ3 – Energy: what is the energy improvement that Jinn-C delivers on top of existing techniques?
RQ4 – Convexity: is convexity a common property of the space formed by the mapping of function arguments to ideal hardware configurations?

We compare Jinn-C with two approaches: Sreelatha *et al.* (Sreelatha et al. 2018)'s CHOAMP, and ARM's GTS (Jeff 2013). GTS, short for *Global Task Scheduling*, is Linux' heterogeneity-aware scheduler in our big.LITTLE system. Yet, before delving into numbers, we introduce the runtime environment we have used to evaluate Jinn-C.

The Hardware. Experiments were performed in an Odroid XU4 development board. This device is powered by a Samsung Exynos 5422 SoC with four ARM Cortex A15 cores, running at up to

```
void task(Stream<Value> s, long keySize) {
 double Theta[][] = {{-0.0125, 0.0114, 0.0006, -0.6481},
                     {-0.1964, 0.0472, 0.0166, -0.0759},
                     {-0.1763, -0.0008, 0.0000, 0.0002},
                     {0.0010, -0.0003, -0.0050, 0.0000}};
 double A[] = {1.0, s.size(), keySize, Thread.activeCount()};
 double P[] = Regression.softmax(Regression.mul(A, Theta));
 int i = indexLargestElement(P);
 Config originalConfig = Regression.getCurrentConfiguration();
 Config config = Regression.getConfiguration(i);
 Regression.changeConfiguration(config);
  while (!s.empty()) {
    BigInteger key = getNextKey(keySize);
    synchronized(globalMap) {
      Value value = s.next();
      globalMap.put(key, value);
    }
  }
 changeConfiguration(originalConfig);
}
```

```
// Returns the product A×θ
double[] mul(double[] A, double[][] θ);

// Applies the σ function onto d
double[] softmax(double[] d);

// Returns the index that holds the largest
// value within vector pred
int indexLargestElement(double[] pred);

// Get the i-th hardware configuration
Config getConfiguration(int i);

// Get the configuration currently in use
Config getCurrentConfiguration();

// Change the configuration currently in use
// to the new configuration g
void changeConfiguration(Config g);
```

Fig. 13. The production version of function Task, first seen in Figure 1.

2.0GHz, and four Cortex A7 cores running at up to 1.5GHz. The board features 2GB of LPDDR3 RAM. To measure the energy consumed exclusively by specific functions, we send signals to the synchronization circuit seen in Figure 4-a through one of the board's GPIO pin. Code to emit the signal is inserted into Java bytecodes via Soot, immediately before the invocation of a function of interest, and immediately after that function returns. We use the energy measurement framework proposed by Bessa *et al.* (Bessa et al. 2017). Power is measured by a National Instruments *DAQ* USB 6009 device, at a rate of 12,000 samples per second.

The Software Stack. We use Oracle's OpenJDK/JRE 11 LTS and Soot 3.2.0 to process bytecodes. No modifications have been made in the Java Virtual Machine –transformations performed by either Jinn-C or CHOAMP happen at the bytecode level, and are carried out via Soot. To mitigate the effect of JIT compilation in the execution time of benchmarks, each application has a warm-up stage before actual execution (details in Table 1). We have used Python 3.4 and Scikit Learn (Pedregosa et al. 2011) to implement regression. Python was also used, in addition to GNU Bash 4.4.19, to generate the suite of micro-benchmarks used to train CHOAMP. The Operating System in the Odroid XU4 used in our experiments is the GNU/Linux Ubuntu 18.04 LTS with kernel 4.17.

The Benchmark Suite. This paper uses the 18 benchmarks shown in Table 1. Eight of them were taken from Acar *et al.* (Acar et al. 2018), who had selected nine programs from *Problem Based Benchmark Suite* (PBBS) (Shun et al. 2012) to evaluate concurrency models. The version of PBBS used by Acar *et al.* (Acar et al. 2018) was implemented in C/C++, so we had to re-implement all the benchmarks in Java. We removed DelaunayTriangulation from our collection, because we could not ensure that its parallel implementation always produces the same output. The triangulation varies depending on how threads are scheduled; hence, the output of different versions of this benchmark might not be equal.

We chose six programs from the *Renaissance* benchmark collection, which was recently released by Prokopec *et al.* (Prokopec et al. 2019). Renaissance contains 21 benchmarks. All the programs in that collection come with only one set of input values. We chose only six benchmarks because we had to understand and augment each program with more inputs. We have also added verification

code to these benchmarks to check execution correctness. The six benchmarks that we chose are implemented in Scala. Our criterion when picking up programs was simplicity: we selected benchmarks that were easy to modify. We have opted for Scala programs to demonstrate that Jinn-C can deal well with languages other than Java.

In addition to PBBS and Renaissance, Jinn-C is distributed with four typical parallel algorithms. CollinearPoints and randomNumComp were taken from public repositories; HashSync, was adapted from Butcher's book; and insertAndAdd was adapted from Zhang et al. (2020)'s Figure 3. Table 1 presents an overview of the benchmarks, as well as basic characteristics of their code.

We recall an observation already made by Prokopec et al., when introducing the Renaissance Benchmark Suite: the source code of the benchmarks is relatively small; however, they cause the invocation of hundreds of different methods, potentially millions of times. Many of these methods belong to the Java Standard Library (e.g., `java.util.*`, `java.lang.System`, etc). Three of the Renaissance programs: textscAls, ChiSquare and DecTree also rely heavily on Apache's `spark` engine. The two last columns of Table 1 report the static and dynamic count of methods invoked by each benchmark. To derive these numbers, we use Oracle's `VisualVM` profiler on the default inputs of each benchmark. Notice that numbers produced through this profiling technique cannot be exactly reproduced: small variations are expected due to internal operations of the Java Virtual Machine, concerning, for instance, the activation of the garbage collector or the just-in-time compiler. Example 4.1 clarifies this observation for SpanningForest. The choice of benchmark in this case is arbitrary, because the other programs that we evaluate show similar behavior.

Example 4.1. The benchmark called SpanningForest is part of the Problem Based Benchmark Suite. It computes a series of minimum spanning trees –one per connected component– of undirected graphs. This benchmark's source code contains 410 lines of code, grouped into 31 methods organized in five `java` files. We adapt the method called "`benchmark`", which receives a list of edges and an integer denoting the number of threads that will be created to process these edges. By default, this benchmark receives a graph with 120 nodes and 1,000 edges. With these default inputs, "`benchmark`" causes the invocation of 1,905 different methods. Out of this lot, 1,874 are either part of the Java Standard Library (in classes like `java.util.ArrayList` or `java.util.concurrent.Executors`), or are part of the Java Virtual Machine. In a single run of this benchmark with default inputs, these methods were invoked 106,462 times. These numbers, 1,874 and 106,462, might vary due to internal operations of the JVM.

Available Inputs. Each benchmark comes with a default input set. We have augmented every one of them with 13 additional inputs. We tried, as best as our knowledge of the benchmark allowed it, to maximize the diversity of inputs, having data of different sizes. Ten of these inputs, randomly chosen, are used for training. When evaluating the model trained for a benchmark, we use four unseen and randomly chosen inputs. Sections 4.2 and 4.3 further discuss the impact of different inputs in the execution time and energy consumption of the applications. The separation of inputs into training and evaluation sets is random.

Choice of regression target. We optimize one method per benchmark: the routine invoked by the benchmark's `main` function. This approach is equivalent to doing regression on inputs of the whole program. In other words, in practice, we are tuning the entire program; not a single method of it. As previously mentioned, each of these adapted functions will, during a normal execution of the benchmark, cause the invocation of many methods external to the benchmark's source code. All these activations will run in the hardware configuration determined by the regression model. To give the reader some perspective on this approach, Table 1 contains in the last two columns the number of different functions that are affected by the change in hardware configuration (*stCalls*), plus the number of times that these functions are dynamically invoked (*dyCalls*).

Source	*Benchmark*	*TTime*	*Lang.*	*LoC*	*W*	*stCalls*	*dyCalls*
Jinn-C	collinearPoints	32m1	J	565	3	576	3,724
Jinn-C	hashSync	94m7s	J	73	3	649	3,838
Jinn-C	insertAndAdd	47m30s	J	130	4	645	47,978
Jinn-C	randomNumComp	26m7s	J	89	6	631	3,399
(Shun et al. 2012)	bfs	42m33s	J	353	4	736	13,713,590
(Shun et al. 2012)	radixSort	20m51s	J	501	4	2,045	13,821
(Shun et al. 2012)	sampleSort	26m17s	J	414	3	625	3,173
(Shun et al. 2012)	suffixArray	30m12s	J	316	3	288	162,976,109
(Shun et al. 2012)	removeDuplicates	30m31s	J	174	4	1,204	12,341
(Shun et al. 2012)	convexHull	56m30s	J	499	5	246	28,488,450
(Shun et al. 2012)	nearestNeighbors	30m29s	J	715	3	1,335	3,664,153
(Shun et al. 2012)	spanningForest	21m40s	J	410	4	1,905	106,462
(Prokopec et al. 2019)	als	80m12s	S/J	97	1	165,108	2,123,294
(Prokopec et al. 2019)	philosophers	21m15s	S/J	146	1	10,446	96,129
(Prokopec et al. 2019)	futureGenetic	26m8s	S/J	115	1	909	8,426
(Prokopec et al. 2019)	finagleHTTP	225m10s	S/J	119	1	785	1,554
(Prokopec et al. 2019)	chiSquare	27m15s	S/J	101	1	185,202	3,523,326
(Prokopec et al. 2019)	decTree	64m22s	S/J	129	1	836,283	5,717,648

Table 1. Benchmarks used for evaluating Jinn-C. Column *TTime* shows time to train each benchmark, which will be further explained in Section 4.1. *Lang.* contains the source language of benchmarks: *J* stands for Java and *S* for Scala. Column *W* shows the number of *warm-up* executions performed by each application. *StCalls* approximates the number of methods invoked by a benchmark running with default inputs; *dyCalls* approximates the number of times that those functions are invoked. These numbers were produced with Oracle's `VisualVM` profiler; hence, small variations are to be expected. collinearPoints finds three points on the same line; hashSync inserts in a concurrent table; randomNumComp has several long sequences of branches that are hard to predict; and insertAndAdd implements parallel operations on a Database.

On the Choice of Hardware Configurations. When training Jinn-C and CHOAMP, we follow the methodology proposed by Sreelatha *et al* (Sreelatha et al. 2018). Thus, we consider a universe of six configurations: 4b4L (4 big and 4 LITTLE cores), 4b0L, 0b4L, 2b2L, 2b0L and 0b2L. LITTLE cores run at maximum frequency: 1.5GHz. Big cores are statically set to run at either 1.6GHz or 1.8GHz (instead of using the maximum 2.0GHz frequency level) due to known thermal issues (da Silva et al. 2019, Sec-4.2). GTS is allowed to choose among any possible hardware configuration involving big and LITTLE cores, and the different frequency levels available in the hardware. For the sake of reproducibility and to better understand the impacts of our technique, we have disabled *Dynamic Voltage and Frequency Scaling* (DVFS) when using either Jinn-C or CHOAMP, but not GTS. Regardless of the methodology used to choose hardware configurations, thread scheduling uses Linux' default *Heterogeneous Multi-Processor* (HMP) scheduler (Rezki and Wool 2015), which is integrated into the kernel's *Completely Fair Scheduler* (CFS). Thus, threads might switch between big and LITTLE cores. The higher the CPU utilization of a thread, the higher the likelihood that it will run on a big core.

On the Implementation of CHOAMP. CHOAMP is a system that, different from our approach, relies on the syntax of the program text –and on its implied semantics– to predict ideal hardware configurations. CHOAMP represents this text of code as a set of characteristics that are useful for training and prediction. Such characteristics, also called *prime features* by its authors, are split into two different groups: language dependent and independent. Language independent features, such as number of branches or memory accesses, are easier to identify and port, as they tend to appear in

Prime Feature	*Lang.Dep.*	*OpenMP*	*Java VM*
Branch operations	No	-	-
Memory operations	No	-	-
Atomic operations	Yes	omp atomic	atomic
Barriers	Yes	omp barrier	CyclicBarrier, Phaser
Critical Sections	Yes	omp critical	Synchronized
False Sharing	No	-	-
Flush operations	Yes	omp flush	not used

Table 2. Prime features and their correspondent Java VM implementation.

most languages. On the other hand, features that depend on a specific programming language need to be adapted when porting the technique to new environments. CHOAMP was initially designed to work with OpenMP applications implemented in C; therefore, some of the prime features used by Sreelatha *et al.* depend on OpenMP constructs. Our re-implementation of CHOAMP targets Java applications running on Hotspot; thus, some of its features had to be adapted to our needs. Table 2 presents the list of program characteristics originally used by CHOAMP for OpenMP and the new version of them, adapted to the JVM.

Most language dependent features find correspondents in the Java standard library, as is the case of the *omp atomic* pragma, which we derived from classes in the package `java.util.concurrent.atomic`. For instance, the occurrence of method `incrementAndGet()`, from the `AtomicInteger` class, would add an "Atomic Operation" to the feature vector of the function where `incrementAndGet()` is invoked. However, some features like *flush operations*, proposed by Sreelatha *et al.* (Sreelatha et al. 2018), were not reused in our implementation, due to a lack of correspondents.

Training and Tuning Following Sreelatha *et al.*, we have trained the probabilistic model of CHOAMP by running it on a set of generic micro-benchmarks. As the original training set was written in C and OpenMP, we had to create a new training set that suits Java. The micro-benchmarks we used were directly based on the scripts made public by Sreelatha *et al.* These scripts generate hundreds of micro-benchmarks. The user adjusts the intensity of each prime feature through command line inputs. We used the originals generator scripts at https://bitbucket.org/jkrishnavs/openmp-eigenbench, adjusting the code to Java. We also used the same range and intensity of features as used in the original work of CHOAMP. Sreelatha *et al.* have proposed three different regression models for CHOAMP. We have experimented with all of them, and end up choosing the linear fit, because, in our setup, it yields better results than the Quadratic and Gaussian predictors. This result in on par with the findings of Sreelatha *et al.*

4.1 RQ1: Training

Jinn-C and CHOAMP require training to adjust the parameters of the regression models. While this cost is paid once by CHOAMP, when performing the training over a set of generic micro-benchmarks, Jinn-C pays this cost for each application that it optimizes. CHOAMP uses micro-benchmarks for training; Jinn-C uses the application itself. The training time of CHOAMP is computed over a set of 285 micro-benchmarks over all the hardware configurations mentioned in our experimental setup. In our hardware, we took about 780 minutes to train our implementation of CHOAMP.

To train Jinn-C, we follow the methodology described in Section 3.2.1. Jinn-C's training time depends on the target application's run time, and on the number of available inputs. Table 1 shows the training time of each benchmark. Using ten inputs and ten allowed hardware states (clock speed × hardware configurations) per benchmark, we took 903 minutes to train the 18 programs

used in this section. The longest time, three hours and 45 minutes was spent in Renaissance's FINAGLEHTTP. PBBS's RADIXSORT gave us the fastest training time: 20 minutes and 51 seconds.

Once the benchmark is trained, no further pre-processing is required. The product of training, the code earlier seen in Figure 13, is embedded directly into a program's bytecode, and runs in constant time. Training's overhead bears no impact once the optimized code runs. The only overhead that is imposed onto optimized programs comes from the matrix multiplication that happens once a hot function is invoked, as we have discussed in Section 3.3. As we will see in Section 4.2, this runtime overhead is too low to be reliably measured.

4.2 RQ2: Speed

Figure 14 summarizes the comparison of the three different schedulers, when the objective function that JINN-C and CHOAMP minimize is the execution time of target applications. We have tested each benchmark with four input sets, adopting a significance level $\alpha = 0.05$; i.e., a confidence level of 95%. So, if the results reported by, for instance, JINN-C and CHOAMP cannot be distinguished with a confidence of more than 95%, then we consider them as originating from the same population. Thus, we use Student's Test to measure the p-value of two populations, and consider significant results with a p-value less than 0.05. White boxes with letters in Figure 14 identify the technique which achieved the best result for a combination of benchmark and input. J stands for JINN-C, C for CHOAMP and G for GTS; X means that the winning systems have produced results statistically similar (p-value greater than 0.05). Above each input set, we show the configuration that JINN-C chose for that input. The grey box, at the right of the name of each benchmark, is the configuration that CHOAMP chooses for that benchmark.

We notice that in 26 cases, out of 72 combinations of [benchmarks × inputs], JINN-C achieved better results when compared to the other techniques. In other 42 cases, JINN-C was at least as fast as GTS or CHOAMP. CHOAMP, in turn, accounted for 3 best results, and GTS for only one, in HASHSYNC's IN4. These numbers show that JINN-C usually outperforms the two other techniques considered in this paper; however, JINN-C's performance still depends on a good choice of inputs for training. Example 4.2 highlights this fact. Further discussion about the importance of finding good training sets shall appear in Example 4.6 (Page 27).

Example 4.2. JINN-C performed rather poorly in COLLINEARPOINTS, due to an unfortunate choice of training inputs. Indeed, the 10 training inputs chosen when optimizing COLLINEARPOINTS find in 4B4L their best configuration. However, coincidentally, three of the test inputs ask for 4B0L. It suffices to switch one of the test and training inputs to put JINN-C on pair with the other schedulers.

This experiment shows that configurations impact in non-trivial ways the behavior of applications. All the winning configurations, regardless of the technique, converged to the frequency of 1.8GHz whenever at least one big core was present. The most recurring configurations were 4b4L (16x for CHOAMP and 37x for JINN-C), 0b4L (2x/11x), 4b0L (17x for JINN-C only), 2b0L (4x for JINN-C only), and 0b2L (2x for JINN-C only). Example 4.3 further emphasizes the importance of the hardware configuration in the behavior of a program.

Example 4.3. Considering CHISQUARE's last input (WORKERS = 2, SIZE = 1,023,464), JINN-C prediction of the configuration 4b0L led to a mean run time of 8.18 seconds, while CHOAMP decision led to 8.47 and GTS to 9.00. For its second input (WORKERS = 4, SIZE = 2,250,467), we observed JINN-C prediction (4b0L) leading to mean run time of 17.00 seconds, CHOAMP to 18.70 and GTS to 17.76. In every case, p-values comparing these populations were less than 0.008.

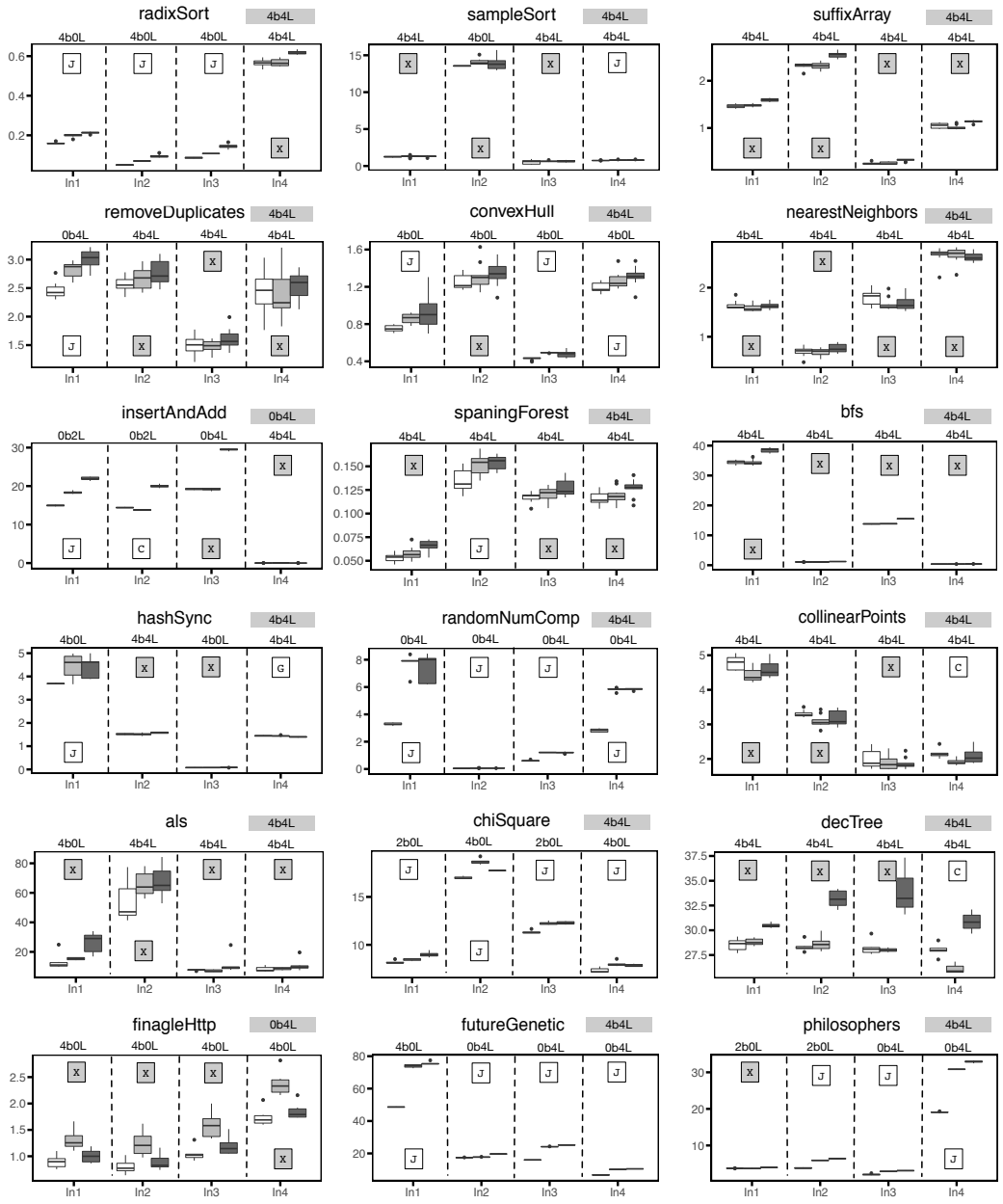

Fig. 14. Execution time of benchmarks from Table 1. Y-axis shows time in seconds. X-axis shows different experiments; each experiment uses different inputs. Boxplots are ordered by Jinn-C, CHOAMP and GTS.

4.3 RQ3: Energy

Figure 16 compares CHOAMP, GTS and Jinn-C regarding energy consumption. The clock speed of 1.6GHz was the most common among all the schedulers, except for one input set of radixSort,

when CHOAMP chose to use 1.8GHz. GTS can choose any possible frequency levels from 200MHz to 1.8GHz in the big cluster and from 200MHz to 1.5GHz in the little one. This flexibility may lead to performance degradation because GTS increases frequency gradually, even in computation-intensive programs. Even with several warm-up rounds, GTS might take an excessively long time to achieve maximum frequency levels for some applications. Thus, Jinn-C outperforms GTS mostly due to its ability to choose high-performance hardware configurations, such as 4b4L at 1.6GHz, immediately. GTS, in turn, needs a warm-up period to arrive at them.

Figure 16 shows that Jinn-C achieved the best results in 20 experiments (out of 72); GTS won in 2, and CHOAMP in 6. In 44 experiments there was no clear winner. This difficulty to pinpoint a best technique is due to the fact that we measure energy for the entire board. Therefore, peripherals like the fan and the memory bus increase the variance of results. This decision was no accident: we believe that measuring energy for the entire boards provides a more realistic assessment of the behavior of our techniques if adopted in production. CHOAMP has chosen the 0b4L configuration at 1.6GHz for almost all the samples in this evaluation. This behavior is due to some features, such as branching and memory operations, dominating the others in most of the functions that constitute a benchmark. It is possible to improve this behavior by scaling the relative importance of the features; however, this optimization is out of the scope of this work.

On the Influence of Execution History. Execution history impacts the energy consumed by different programs. Take as an example the entry corresponding to hashSync in Figure 16. Although Jinn-C and CHOAMP predicted the same configuration for the second input set (In2), the former consumed marginally more energy. This behavior is even more surprising once we consider that Jinn-C's and CHOAMP's code run within about the same time, as Figure 14 reveals. The culprit of this counter-intuitive result is the board state at the time measurement started. The warm-up phase, in this case, is responsible for giving Jinn-C's and CHOAMP's code different starting states. In the discussion that follows, we shall separate the execution of a benchmark into two parts: *warm-up*, when the target routine is called a number of times to stabilize the Java Virtual Machine; and *measurement*, when the behavior of the benchmark is actually gauged.

Hysteresis. Figure 15 shows the power profile of hashSync, including warm-up and measurement phases. Invocations in the warm-up stage use different inputs than in the measurement stage. As a result, our technique predicted the configuration 4b4L for the last warm-up invocation, which is different than 0b4L, the configuration predicted at measurement. The use of big cores during warm-up increased the amount of energy consumed by the board in the measurement step, due to a well-known phenomenon: the hysteresis of power dissipation.

The mean power dissipated by Jinn-C's version of hashSync in Figure 15(a) was 4.68W. CHOAMP's was 4.09W, as seen in Figure 15(b). Jinn-C's program consumes more energy (9.47J vs 8.22J). However, if we fix the hardware configuration in the warm-up phase of Jinn-C's code, then the average dissipation goes down to 3.95W. Figure 15(c) reports the power profile of this setup. The only difference between the executions of Jinn-C in Figures 15 (a) and (c) is the configuration used in the warm-up stage. There is no statistically significant difference between the amount of energy consumed by CHOAMP and Jinn-C when both warm-up with the same hardware configuration.

This behavior caused by differences between configurations chosen at warm-up and measurement phases only affects Jinn-C. CHOAMP always chooses the same hardware configuration per function, and GTS increases frequency gradually. The only further impact that this difference had in Jinn-C's behavior was observed in decTree and collinearPoints. In both cases, only for the last input set (In4), and only when measuring running time (Fig. 14). The need to change configuration when moving from warm-up to measurement has cost Jinn-C's code some time. Nevertheless, when

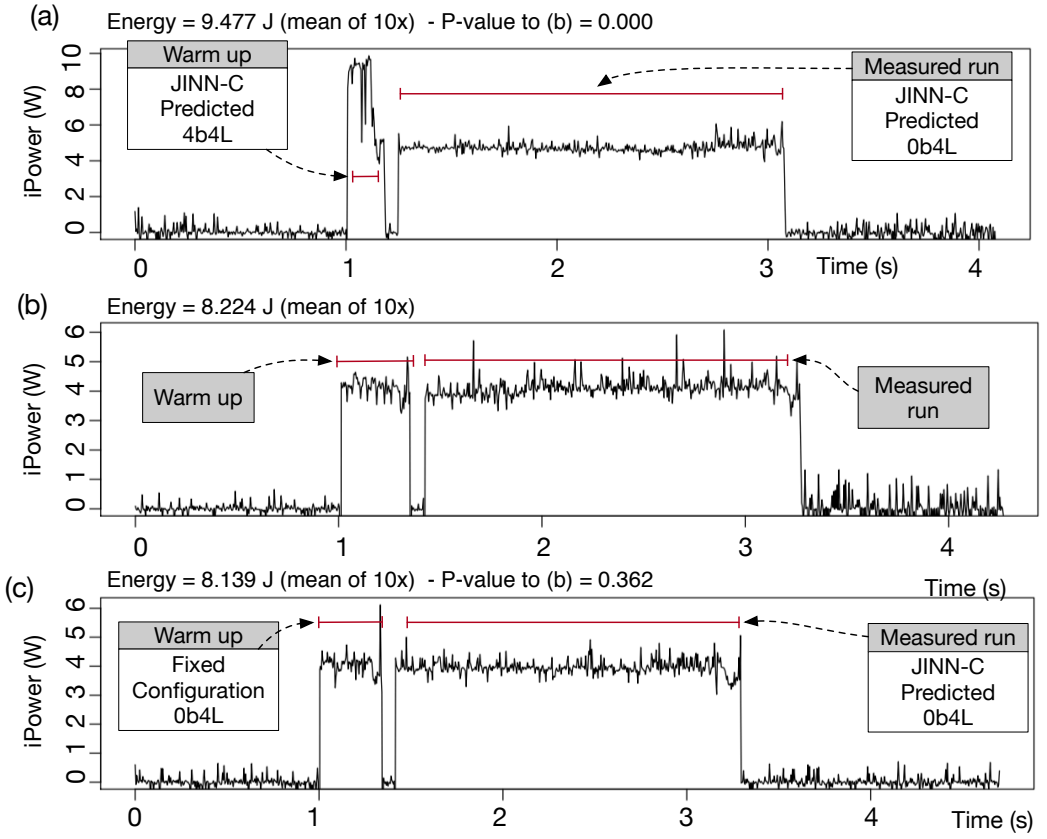

Fig. 15. Power consumption of hashSync with (a) JINN-C, (b) CHOAMP, and (c) JINN-C with fixed configuration at warm-up. P-values below 0.05 indicate that executions of JINN-C's and CHOAMP's code are statistically different. For this benchmark, CHOAMP (b) predicted 0b4L as the best configuration for the parallel kernel. This configuration is used in all warm-up stages and in the measurement phase.

reporting the results in this paper, we have opted to let the hardware configuration fluctuate during warm-up, as this is the expected behavior of JINN-C, once it is deployed in production.

4.4 RQ4: Convexity

A *convex space* is a region within a Euclidean Space whose intersection with any line results in a continuous line segment. If the convex space can be described by a function, then said function is also called *convex*. Convex functions are very important in classification problems. Firstly, because exploration methods based on derivatives, like gradient descent, converge to the optimal solution when applied on them (Boyd and Vandenberghe 2004). Secondly, as we discuss in Section 4.4.2, disjoint convex sets are linearly separable. Thus, linear techniques, like the one employed in this paper, tend to yield good results when used to classify such sets.

In our setting, the search space is a function that maps program inputs to hardware configurations. This function is discrete, because its image is a finite set of hardware configurations. As a consequence of convexity, if we fix all the program inputs and vary one of them, then every

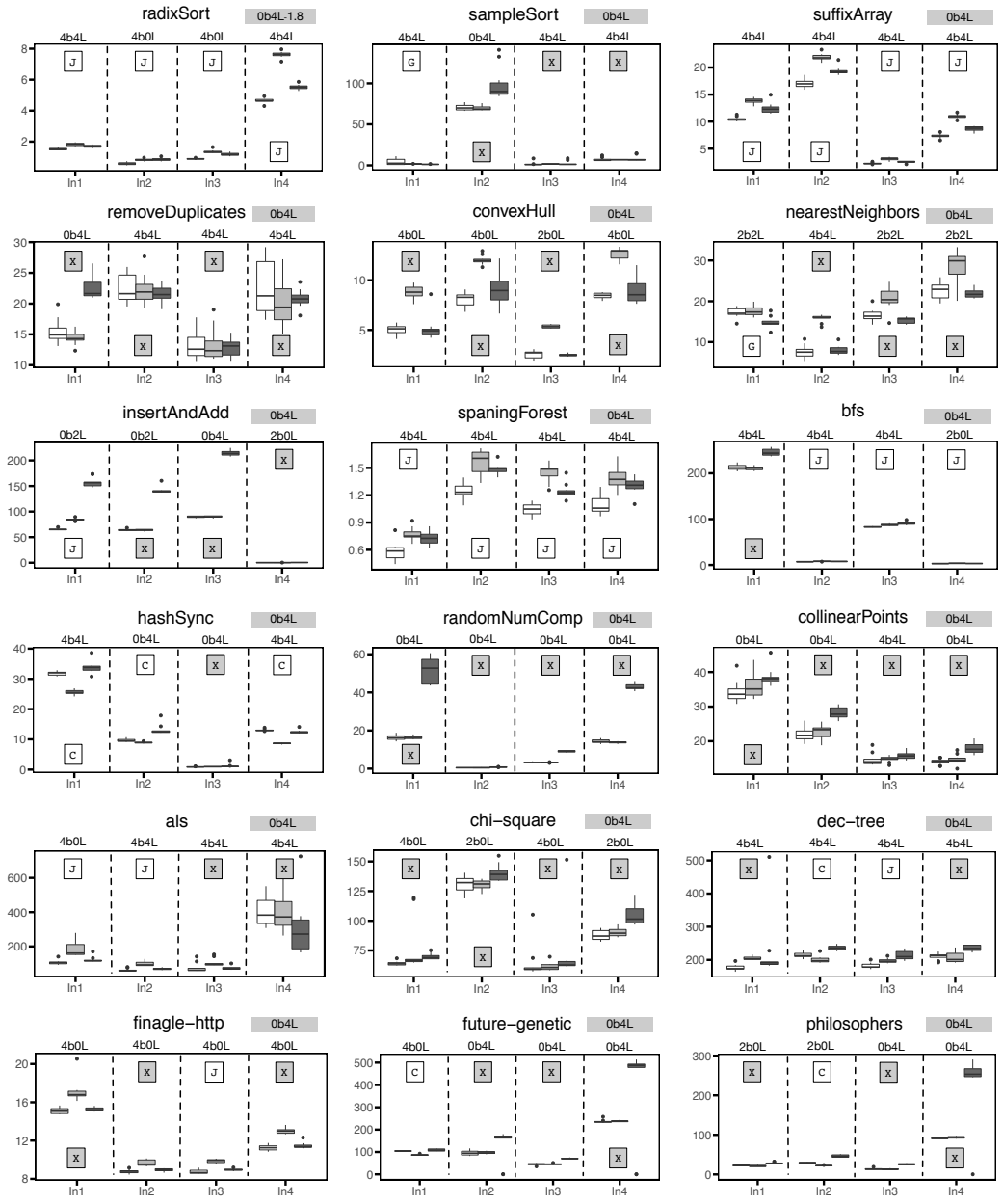

Fig. 16. Energy consumed by the benchmarks in Table 1. Y-axis shows energy in Joules. X-axis shows different experiments. Boxplots are sorted as in Figure 14.

region covered by the same optimal configuration is continuous. In other words, while varying this single input monotonically, we will not leave a region r where a certain configuration h is the best, find a new region r' governed by a different configuration h', only to find h again later,

once we cross the boundary between r' and a third region r''. Notice that convexity is a tendency, not a principle: it is possible to implement programs whose space of optimal configurations is not convex, as Example 4.4 demonstrates.

Example 4.4. The space formed by ideal hardware configurations associated with `i` in `unlikely`, below, is non-convex:

```
void unlikely(int i) {
  if (10 <= i && i <= 100) sync_intensive();
  else comp_intensive();}
```

The `unlikely` routine receives one input, namely, the integer `i`. If `i` is less than 10 or greater than 100, it invokes a computationally intensive procedure; otherwise, it invokes a synchronization intensive one. The optimal configurations for these two pieces of code differ. The space occupied by the optimal hardware configuration for `comp_intensive`, i.e., $[-\infty, 10[\cup]100, +\infty]$, is non-continuous; hence, concave.

4.4.1 Space Exploration. The program discussed in Example 4.4 is unlikely to exist in the real-world. To support this statement, Figure 17 provides a glimpse of the best hardware configurations for different inputs of four benchmarks in our collection. Each matrix in Figure 17 associates a pair of inputs with the hardware configurations that yielded the shortest execution times for those inputs. In this experiment, we chose the two benchmarks from our collection that contain two inputs. We have augmented this set with HASHSYNC and FUTUREGENETIC, to fit the figure into a 2×2 matrix (for aesthetic reasons). However, to avoid having to draw a 3D-figure, we have fixed one input for each benchmark: `number_of_Generations` for FUTUREGENETIC, and `number_of_workers`, for HASHSYNC. The computational time involved in the production of each figure follows: RADIXSORT: 1 hour and 37 minutes; PHILOSOPHERS: 2 hours and 24 minutes; FUTUREGENETIC: 55 hours and 53 minutes; and HASHSYNC: 58 hours and 12 minutes. The four tables in Figure 17 show convex spaces: any sequence of rows or columns traverses a continuous region, as Example 4.5 illustrates:

Example 4.5. Consider HASHSYNC in Figure 17(c). If we fix the value for keySize in 10^6, and vary s.size() in the set $\{10, 10^2, 10^3, 10^4, 10^5\}$, we observe that each one of the continuous intervals $[10, 10]$, $[10^2, 10^2]$ and $[10^3, 10^5]$ is governed by the same set of optimal hardware configurations.

Dealing with variance: To arrive at this result, we had to account for small time variations. To generate the data in every table seen in Figure 17, we considered 5×5 combinations of inputs, and the hardware configurations used in the previous sections. We run each pair of inputs with every configuration of interest 20 times. To reduce variance, we removed the four fastest and the four slowest samples; hence, considering 12 executions per input per configuration. Nevertheless, this expedient only would not be enough to mitigate the problem of high variance, mostly when considering input settings with small runtimes.

To mitigate variance for small inputs, we consider not the best, but the set of best hardware configurations per input. For each input, we fitted our linear regression model using the least squares method to estimate the model parameters. We analyze the differences among group means with standard analysis of variance (ANOVA) (Fisher 1918); hence, generalizing the T-test beyond two means. In the context of this work, we consider groups of hardware configuration; and the null hypothesis states that samples from different hardware configurations came from the same probability distribution. Thus, the null hypothesis means that there is no statistical difference between the execution time of different hardware configurations. We checked if the data were statistically significant considering a confidence of 95%, i.e., a P-value less than 0.05. ANOVA is

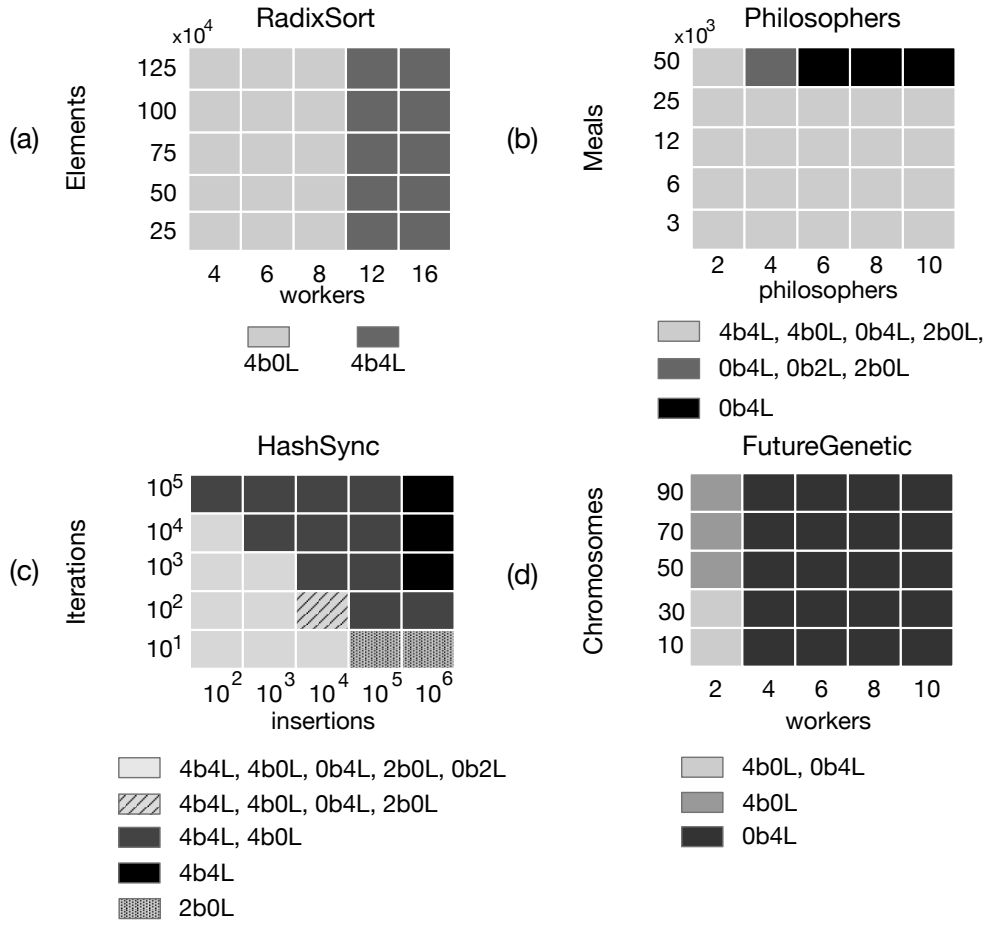

Fig. 17. Best configurations for 4 benchmarks used in our evaluation. The charts exemplify the convex space over benchmarks inputs. HashSync and FutureGenetic receive 3 inputs each, but for this experiment we fixed the number of workers in HashSync to 16 and the number of generations in FutureGenetic to 5000.

an omnibus test –it analyzes the data as a whole; hence, we performed a post-hoc test to find out where the differences among the groups were.

The post-hoc test consists of a series of T-tests between each pair of configurations. The significance level is adjusted to avoid spurious positives. To that end, we used the Bonferroni correction (Bonferroni 1936; Dunn 1958). Each individual hypothesis is tested with a threshold of α/n, where α is the significance level for the entire set of comparisons, e.g., 0.05, and n is the number of statistical tests performed. Thus, analogously to the ANOVA test, if the resulting P-value is lower than the significance level given by the Bonferroni correction, then the null hypothesis can be rejected. Rejection of the null hypothesis is equivalent to assume that the two groups of configurations present a statistically significant difference.

Monotonicity and Convexity. As Example 4.4 shows, Convexity is a tendency, not a rule. Nevertheless, we believe that convexity is common because the asymptotic behavior of most

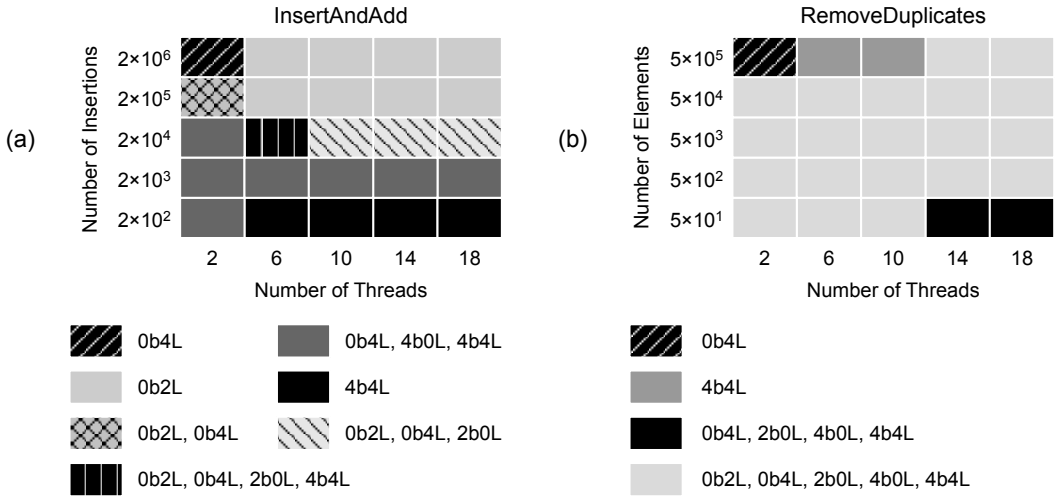

Fig. 18. Best hardware configuration, considering running time as the objective function. InsertAndDouble receives four inputs. We fixed the "initial capacity" in 5,000 and the "change threshold" in 500. RemoveDuplicates receives three inputs. We fixed the "size of the keys" in 16 bytes. This choice of values is arbitrary, and was necessary to give us 2D figures.

algorithms is described by monotonic –therefore convex– functions. The running time of algorithms tend to be governed by convex functions ranging on their inputs. This observation can also be projected onto parallel algorithms, once we consider the number of threads as an input. In other words, variations in the running time of a parallel algorithm tend to be described by (typically decreasing) monotonic functions ranging on the number of available threads (Keller et al. 2000). As an example, Figure 18 shows the best hardware configuration for two of our benchmarks, considering running time as the objective function. In this case, we are varying, in addition to some input, the number of available threads. It is possible to observe the convex shape of the space delimited by the best hardware configurations for these benchmarks.

4.4.2 Separability. In the context of classification problems, the main benefit of convexity is *separability*. The *Hyperplane Separation Theorem* (Boyd and Vandenberghe 2004, Ch.4) states that two disjoint convex sets are linearly separable. In other words, if C_1 and C_2 are closed convex sets such that $C_1 \cap C_2 = \emptyset$, then there exists a linear function $g(x) = w^T x + w_0$ such that $g(x) > 0$ for every $x \in C_1$ and $g(x) < 0$ for every x in C_2. Function g is called a *linear discriminant*, because it can distinguish points from C_1 and C_2 (Chan 2020).

In this paper, we use multivariate linear regression to build linear discriminants. As already mentioned in Section 2.1, these functions range on the space formed by program inputs. Completely chartering this space would involve applying the methodology discussed in Section 3.2 to every possible combination of inputs that a program might use. This task is impractical—if at all possible; thus, in Section 4.4.1 we have evaluated a handful of inputs for a few benchmarks. This evaluation indicates that, at least for these benchmarks, the regions in the search space covered by similar hardware configurations tend to be convex.

Figure 17 shows that the regions in which specific hardware configuration excel are not disjoint. In other words, the same set of function arguments could be mapped to different hardware configurations with similarly good effects. Nevertheless, the linear discriminants that we build

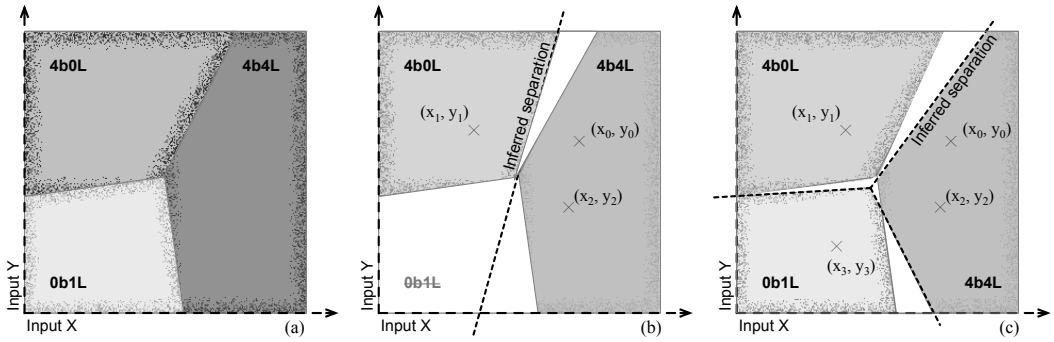

Fig. 19. (a) An ideal representation of the space of best hardware configurations, with three convex sets. (b) Approximation of this space after training with three inputs covering two convex sets. (c) Approximation of this space after training with inputs that cover all the convex sets. White regions are misclassified zones.

are disjoints, for, in case of ties, we choose the configuration with less resources as explained in Section 3.2.1. This said, the accurate classification of regions into the right hardware configurations still depends on a good assortment of inputs. Example 4.6 illustrates this fact.

Example 4.6. Figure 19 (a) shows an idealized representation of the search space for a hypothetical program[2]. This space is formed by three convex sets mapping inputs to configurations 4B0L, 0B4L and 4B4L. A perfect classification model would find the three lines that separate these regions. In practice, classification approximates these lines by maximizing the distance between the known inputs. In this example, we have three of them. The white regions in each figure show points for which classification would give wrong answer. The more inputs are used in training, the smaller these regions tend to be, as the comparison between Figures 19 (b) and (c) illustrates. For a concrete discussion about the impact of training inputs on the quality of the classification model, we refer the reader to Example 4.2 (Page 19).

5 RELATED WORK

Our work explores a type of machine learning technique –multivariate linear regression– to solve an instance of program scheduling in heterogeneous architectures. There exists vast literature about uses of machine learning in compilers (Wang and O'Boyle 2018) and Ashouri *et al.* (Ashouri et al. 2018). Equally abundant is the material about scheduling in heterogeneous multi-core systems. For a comprehensive overview on this topic, we recommend the survey recently carried out by Singh et al. (2020). In the rest of this section, we discuss some of this research, focusing on scheduling, with the intention to explain how our work stands within the contemporary literature.

5.1 A General Overview on Program Scheduling in Heterogeneous Systems

The general problem of scheduling computations in heterogeneous architectures has attracted much attention, as Mittal and Vetter have discussed (Mittal and Vetter 2015). Table 3 provides a taxonomy of previous solutions to this problem. We group them according to the level at which they are implemented, and to the way they answer each of the following four questions:

- **Architecture**: do they apply to Single or Multi-ISA systems?

[2]We say "idealized" because, in practice, we would not be able to determine the best hardware configurations for every input set, due to imprecisions in the measurement apparatus. Imprecisions are especially present when considering small inputs, which tend to cause short running times.

Work	*Level*	*Arch.*	*Source*	*Input*	*Auto*	*Runtime*	*Learn*
(Poesia et al. 2017)	C	Multi	Yes	No	Yes	No	Yes
(Barik et al. 2016)	C	Multi	Yes	No	Yes	Yes	No
(Rossbach et al. 2013)	C/L	Multi	Yes	No	No	Yes	No
(Luk et al. 2009)	C/L	Multi	Yes	No	No	Yes	No
(Joao et al. 2012)	A/L	Multi	Yes	No	No	No	No
(Lukefahr et al. 2016)	A	Multi	No	No	Yes	No	No
(Van Craeynest et al. 2012a)	A	Multi	No	No	Yes	No	No
(Imes et al. 2015)	L	Multi	Yes	No	No	Yes	Yes
(Nishtala et al. 2017)	O	Single	No	No	Yes	Yes	Yes
(Petrucci et al. 2015)	O	Single	No	No	Yes	Yes	No
(Reddy et al. 2020)	O	Multi	No	No	Yes	Yes	No
(Delimitrou and Kozyrakis 2014)	O	Multi	No	No	Yes	Yes	Yes
(Augonnet et al. 2011)	L	Multi	Yes	No	No	No	No
(Piccoli et al. 2014)	O/C	Single	Yes	No	Yes	Yes	No
(Tang et al. 2013)	O/C	Multi	Yes	No	Yes	Yes	No
(Cong and Yuan 2012)	O/C	Multi	Yes	No	Yes	Yes	No
(Sreelatha et al. 2018)	C	Single	Yes	No	Yes	No	Yes
Jinn-C	C	Single	Yes	Yes	Yes	No	Yes

Table 3. Different solutions to the problem of finding ideal hardware configurations. We consider the following levels: Architecture (A), Operating System (O), Compiler (C) or Library/Programming model (L).

- **Source**: is the program's code modified?
- **Input**: is the approach input-aware?
- **Auto**: is user intervention required to choose configuration?
- **Runtime**: is runtime information exploited?
- **Learn**: is there any adaptation to runtime conditions?

Perhaps the most important difference among the several strategies proposed to find ideal hardware configurations 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 (Cong and Yuan 2012; Jain et al. 2016; Luk et al. 2009; Poesia et al. 2017; Rossbach et al. 2013; Sreelatha et al. 2018; Tang et al. 2013), or not. In the latter case, users can use annotations (Mendonça et al. 2017), domain specific programming languages (Luk et al. 2009; Rossbach et al. 2013) or library calls (Augonnet et al. 2011) 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 3, 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 (Joao et al. 2012; Lukefahr et al. 2016; Rangan et al. 2009; Van Craeynest et al. 2012a; Yazdanbakhsh et al. 2015), or at the virtual machine (VM)/OS level (Barik et al. 2016; Gaspar et al. 2015; Nishtala et al. 2017; Petrucci et al. 2015; Somu Muthukaruppan

et al. 2014; Zhang and Hoffmann 2016). By leveraging runtime information, the system can use environment information, unknown at compilation time. 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 weighing 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*. Examples of hybrid solutions to scheduling include works from Piccoli et al. (2014), Cong and Yuan (2012), and Tang et al. (2013). Piccoli *et al* (Piccoli et al. 2014) have used a compiler to instrument a program with guards that determine, based on input sizes, where each loop should run. Cong and Yuan (Cong and Yuan 2012), in turn, use the compiler to partition a program in regions of similar behavior, and rely on runtime information to schedule computation so as to minimize the energy consumed by each region. Finally, Tang *et al.* (Tang et al. 2013) 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 tune the behavior of the scheduler, as Table 3 indicates in the column *Learn*. Guards, once created, behave always on the same way.

5.2 Scheduling in Single-ISA Heterogeneous Systems

Much attention has been dedicated to the problem of finding good placements of computation on heterogeneous multicore systems, as Mittal *et al.* (Mittal 2016) has summarized. However, we emphasize that a large part of this literature concerns the design of scheduling heuristics implemented at the level of the hardware or the operating system (Cai et al. 2016; Garcia-Garcia et al. 2018; Mascitti et al. 2020; Mittal 2016; Neto et al. 2018; Park et al. 2018; Van Craeynest et al. 2012b). This section describes works that, like Jinn-C, can be auto-tuned to characteristics of the runtime environment, and that have been specifically designed for big.LITTLE architectures. By characteristics of the runtime environment we mean the nature of the inputs or the behavior of the hardware. We leave out of this comparison scheduling algorithms that rely on worst-case execution estimates of tasks, such as Mascitti et al. (2020)'s or Roeder et al. (2021). In other words, no assumptions are made on the time or energy budget of a given task.

Table 4 categorizes these techniques along the following lines:

- **Granularity**: what is the data used for training? Most of the techniques use runtime information (R) –available via performance counters. CHOAMP relies on features mined from the program's syntax (S). We use the program's inputs to perform predictions (I).
- **Training**: when does learning occur? Off-line (off) systems calibrate prediction before the target program runs; on-line (on) systems do it while the program executes.
- **Data**: what is the source of training data? OS-based off-line systems usually rely on micro-benchmarks (μ-benchs) to perform calibration. CHOAMP uses features of the program, which it extracts from its syntax. Techniques used in servers can rely on the target program itself as the source of training data, for said program is bound to run for a long time.
- **Target**: in which scenario is the technique meant to be used? Most of the papers that deal with Isha (Definition 2.3), ours included, present solutions for embedded devices and smartphones (clients). Sloop, SaC, Octopus-Man and Hipster guarantee QoS in servers.
- **Level**: the different adaptive techniques that we list in Table 4 either run on the operating system (OS), or are implemented at the compiler's level (CP).

Approach	*Gran.*	*Tr.*	*Data*	*Target*	*Level*
Oct-Man (Petrucci et al. 2015)	R	on	self	server	OS
SaC (Azhar et al. 2019)	R	on	self	server.	OS
Sloop (Azhar et al. 2017)	R/S	on	self	server	OS/CP
Sparta (Donyanavard et al. 2016)	R	off	μ-bench	client	OS
DyPO (Gupta et al. 2017)	R	off	μ-bench	client	OS
Tzilis *et al* (Tzilis et al. 2019)	R	off-line	μ-bench	client	OS
Hispter (Nishtala et al. 2017)	R	off/on	μ-bench+self	server	OS
CHOAMP (Sreelatha et al. 2018)	S	off	μ-bench	client	CP
SIAM (Krishna and Nasre 2018)	S+I	off	self	client	CP
Jinn-C	I	off	self	client	CP

Table 4. Recent solutions to Isha. Granularity (Gran.) uses R for runtime, S for syntax and I for input. Training (Tr.) uses on for on-line and off for off-line. Level uses OS for operating system and CP for compiler.

Two systems that solve Isha (Definition 2.3) in big.LITTLE architectures at the compiler level are Sreelatha *et al.* (Sreelatha et al. 2018)'s CHOAMP, and Krishna *et al.* (Krishna and Nasre 2018)'s SIAM. We have compared Jinn-C with CHOAMP extensively in this paper. SIAM, in turn, is a system that targets specifically graph algorithms parallelized via OpenMP. It consists of a prediction model that, given a particular shape of a graph, determines the best data-structure format and hardware configuration for that shape. We could, in principle, adapt it to implement some of our benchmarks, such as spanningForest and BFS –graph-based algorithms. However, this implementation would involve providing each algorithm with different graph representations –a task to be paid at a non-negligible programming cost.

5.3 Input-Aware Program Autotuning

Our work is centered around the idea that characteristics of the input can be used to determine the behavior of a program. Autotuning techniques that take decisions based on inputs are well-known. Even relatively old libraries such as `FFTW` (Frigo and Johnson 2005) provide code that is optimized for different input sizes (Guarrasi et al. 2013). And more recent work (Esper et al. 2021; Teich et al. 2021) has demonstrated that input-awareness can be used to keep programs running in multi-core systems within the limits of predetermined energy or time requirements. Along these lines, a direction of research that has been much explored concerns the matching of inputs with data-structures (Costa and Andrzejak 2018; Jung et al. 2011; Schiller et al. 2016). We emphasize that none of these previous techniques embed regression code into the compiled program, in order to bestow on said program the ability to choose hardware configurations based on input value.

In this regard, recent work by Oliveira et al. (2021) have shown that the implementation of data structures bear much impact upon the energy consumption of mobile applications. From their observations, Oliveira et al. provide developers with a number of recommendations to code energy efficient software, following a methodology previously proposed by Couto et al. (2020). This line of work aims at building tools that work like code linters: tools that point out potential power inefficiencies to programmers. This *modus operandi* is made clear by Melfe et al. (2018). This paper, in contrast, describes an automatic optimization: our code transformation works at the compiler level, and does not require intervention from users.

6 CONCLUSION

This paper presented an end-to-end code generation technique that matches programs to hardware configurations in heterogeneous multicore systems. This paper is centered around the thesis that the values of a function's arguments provide enough information to predict the best hardware configuration for that function. Our technique is able to outperform, be it in energy consumption, be it in execution time, the default Linux scheduler for ARM (the Global Task Scheduler), and CHOAMP, a recently released tool that predicts the best hardware configuration to a parallel program based on its syntax.

Limitations and Future Work. As any profiler-based technique, Jinn-C has limitations. Its effectiveness depends on having adequate training inputs for each program that it optimizes. However, the more inputs are used, the longer the training time, as observed in Section 3.2. We speculate that it might be possible to remove the need of training per application if we use static-profiling techniques (Ball and Larus 1993; Wu and Larus 1994) to infer, at compilation time, how the inputs of a program might influence that program's behavior. In terms of engineering, we believe that the techniques that we advocate in this paper could be ported to programming environments other than the JDK's ecosystem. For instance, to port our technique to C/OpenMP, Jinn-C could be reimplemented in LLVM (Lattner and Adve 2004). In this regard, LLVM would fill in that setting the role that Soot has filled in this paper. Notice that our techniques do not depend on static program features, so feature engineering would not be necessary for portability. We leave the investigation of such possibilities as future work.

Software. Jinn-C is available at https://github.com/lac-dcc/JINN-C under the GPL-3.0 License. Details about this project can be found at https://homepages.dcc.ufmg.br/~juniocezar/intelligentDVFS

ACKNOWLEDGMENTS

This work has been made possible by the support of the following agencies: ANR (the CONTINUUM project: grant ANR-15-CE25-0007-01); CNPq (Grants PDE-202896/2017-0 and 406377/2018-9); FAPEMIG (Grant PPM-00333-18) and CAPES (Edital CAPES PrInt). While working on this project, Junio Cezar was the recipient of a scholarship generously donated by Google LLC (*Research Awards for Latin America—LARA*).

REFERENCES

Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *PLDI*. ACM, New York, NY, USA, 769–782.

Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *Comput. Surv.* 51, 5 (2018), 96:1–96:42. DOI: http://dx.doi.org/10.1145/3197978

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.

Muhammad Waqar Azhar, Miquel Pericàs, and Per Stenström. 2019. SaC: Exploiting Execution-Time Slack to Save Energy in Heterogeneous Multicore Systems. In *ICPP*. ACM, New York, NY, USA, 26:1–26:12. DOI: http://dx.doi.org/10.1145/3337821.3337865

M. Waqar Azhar, Per Stenström, and Vassilis Papaefstathiou. 2017. SLOOP: QoS-Supervised Loop Execution to Reduce Energy on Heterogeneous Architectures. *ACM Trans. Archit. Code Optim.* 14, 4, Article 41 (2017), 25 pages. DOI: http://dx.doi.org/10.1145/3148053

D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results. In *Supercomputing*. ACM, New York, NY, USA, 158–165.

Thomas Ball and James R. Larus. 1993. Branch Prediction for Free. *SIGPLAN Not.* 28, 6 (1993), 300–313. DOI: http://dx.doi.org/10.1145/173262.155119

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, New York, NY, USA, 70–81.

Tarsila Bessa, Ghristopher Gull, Pedro Quint ao, Michael Frank, José Nacif, and Fernando Magno Quint ao Pereira. 2017. JetsonLEAP: A framework to measure power on a heterogeneous system-on-a-chip device. *Science of Computer Programming* 33, 1 (2017), 1–37.

Carlo Emilio Bonferroni. 1936. Teoria statistica delle classi e calcolo delle probabilità. (1936).

Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press, New York, NY, USA.

Paul Butcher. 2014. *Seven Concurrency Models in Seven Weeks* (1st ed.). Pragmatic Bookshelf, Raleigh, NC, US.

Haoran Cai, Qiang Cao, Feng Sheng, Manyi Zhang, Chuanyi Qi, Jie Yao, and Changsheng Xie. 2016. Montgolfier: Latency-aware power management system for heterogeneous servers. In *IPCCC*. IEEE, Washington, DC, USA, 1–8.

M. Augustine Cauchy. 1847. Méthode Générale pour la résolution des systèmes d'Équations simultanées. *Comptes Rendus Hebd. Séances Acad. Sci.* 25, 10 (1847), 536–538.

Stanley Chan. 2020. Linear Separability. (2020). Lecture Notes on Machine Learning - STAT598.

Jason Cong and Bo Yuan. 2012. Energy-efficient Scheduling on Heterogeneous Multi-core Architectures. In *ISLPED*. ACM, New York, NY, USA, 345–350.

Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2005. ACME: Adaptive Compilation Made Efficient. In *LCTES*. ACM, New York, NY, USA, 69–77.

Diego Costa and Artur Andrzejak. 2018. CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection. In *CGO*. Association for Computing Machinery, New York, NY, USA, 16–26. DOI: http://dx.doi.org/10.1145/3168825

Marco Couto, João Saraiva, and João Paulo Fernandes. 2020. Energy Refactorings for Android in the Large and in the Wild. In *SANER*, Kostas Kontogiannis, Foutse Khomh, Alexander Chatzigeorgiou, Marios-Eleftherios Fokaefs, and Minghui Zhou (Eds.). IEEE, New York, NY, USA, 217–228. DOI: http://dx.doi.org/10.1109/SANER48275.2020.9054858

Junio Cezar Ribeiro da Silva, Lorena Le ao, Vinícius Petrucci, Abdoulaye Gamatié, and Fernando Magno Quint ao Pereira. 2019. *Scheduling in Heterogeneous Architectures via Multivariate Linear Regression on Function Inputs*. Technical Report LIRMM-02281112. CNRS.

Junio Cezar Ribeiro da Silva, Lorena Le ao, Vinícius Petrucci, Abdoulaye Gamatié, and Fernando Magno Quint ao Pereira. 2020. Mapping Computations in Heterogeneous Multicore Systems with Statistical Regression on Inputs. In *SBESC*. IEEE, USA, 42–49.

Junio Cezar Ribeiro da Silva, Fernando Magno Quintão Pereira, Michael Frank, and Abdoulaye Gamatié. 2018. A Compiler-Centric Infra-Structure for Whole-Board Energy Measurement on Heterogeneous Android Systems. In *ReCoSoC*. IEEE, Washington, DC, USA, 1–8.

Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. 2014. Continuously Measuring Critical Section Pressure with the Free-lunch Profiler. *SIGPLAN Not.* 49, 10 (2014), 291–307.

Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS*. ACM, New York, NY, USA, 127–144.

Bryan Donyanavard, Tiago Mück, Santanu Sarma, and Nikil Dutt. 2016. SPARTA: Runtime Task Allocation for Energy Efficient Heterogeneous Many-cores. In *CODES*. ACM, New York, NY, USA, 27:1–27:10.

Olive Jean Dunn. 1958. Estimation of the Means for Dependent Variables. *Annals of Mathematical Statistics*. 29 (1958), 1095–1111. Issue 4.

Khalil Esper, Stefan Wildermann, and Jürgen Teich. 2021. A Comparative Evaluation of Latency-Aware Energy Optimization Approaches in Many-Core Systems (Invited Paper). In *NG-RES (OpenAccess Series in Informatics (OASIcs))*, Marko Bertogna and Federico Terraneo (Eds.), Vol. 87. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:12. DOI: http://dx.doi.org/10.4230/OASIcs.NG-RES.2021.1

Ronald A. Fisher. 1918. The Correlation Between Relatives on the Supposition of Mendelian Inheritance. *Philosophical Transactions* 52 (1918), 399–433.

M. Frigo and S.G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216 –231. DOI: http://dx.doi.org/10.1109/JPROC.2004.840301

Adrian Garcia-Garcia, Juan Carlos Saez, and Manuel Prieto. 2018. Contention-Aware Fair Scheduling for Asymmetric Single-ISA Multicore Systems. *IEEE Trans. Computers* 67, 12 (2018), 1703–1719. DOI: http://dx.doi.org/10.1109/TC.2018.2836418

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 (Dec. 2015), 42:1–42:25.

Peter Greenhalgh. 2011. Big.LITTLE processing with ARM cortex-A15 & cortex-A7. (2011). https://www.eetimes.com/document.asp?doc_id=1279167

Massimiliano Guarrasi, Giovanni Erbacci, and Andrew Emerson. 2013. Auto-tuning of the FFTW Library for Massively Parallel Supercomputers. (2013).

Ujjwal Gupta, Chetan Arvind Patil, Ganapati Bhat, Prabhat Mishra, and Umit Y. Ogras. 2017. DyPO: Dynamic Pareto-Optimal Configuration Selection for Heterogeneous MpSoCs. *Trans. Embed. Comput. Syst.* 16, 5s (2017), 123:1–123:20. DOI:

http://dx.doi.org/10.1145/3126530

Mark Gurman, Debby Wu, and Ian King. 2020. Apple Aims to Sell Macs With Its Own Chips Starting in 2021. (2020). Accessed on July 2021.

Marcus Hähnel and Hermann Härtig. 2014. Heterogeneity by the Numbers: A Study of the ODROID XU+E Big. LITTLE Platform. In *HotPower*. USENIX Association, Berkeley, CA, USA, 3–3.

Connor Imes, David H. K. Kim, Martina Maggio, and Henry Hoffmann. 2015. POET: a portable approach to minimizing energy under soft real-time constraints. In *RTAS*. IEEE, New York, NY, USA, 75–86. DOI: http://dx.doi.org/10.1109/RTAS.2015.7108419

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*. IEEE, New York, NY, USA, 1–12.

Brian Jeff. 2013. *big.LITTLE Technology moves towards fully heterogeneous Global Task Scheduling*. Technical Report. ARM. White paper.

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

Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. 2011. Brainy: Effective Selection of Data Structures. In *PLDI*. ACM, New York, NY, USA, 86–97. DOI: http://dx.doi.org/10.1145/1993498.1993509

Jörg Keller, Christoph Kessler, and Jesper Larsson Träff. 2000. *Practical Pram Programming*. John Wiley & Sons, Inc., USA.

J. M. Kim, S. K. Seo, and S. W. Chung. 2014. Looking into heterogeneity: when simple is faster. (2014). https://news.ycombinator.com/item?id=8714613.

Jyothi Krishna and Rupesh Nasre. 2018. Optimizing Graph Algorithms in Asymmetric Multicore Processors. *Trans. on CAD of Integrated Circuits and Systems* 37, 11 (2018), 2673–2684. DOI: http://dx.doi.org/10.1109/TCAD.2018.2858366

Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *SIGARCH Comput. Archit. News* 32, 2 (2004), 64. DOI: http://dx.doi.org/10.1145/1028176.1006707

Chris Lattner and Sarita V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *CGO*. IEEE, New York, US, 75–86. DOI: http://dx.doi.org/10.1109/CGO.2004.1281665

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

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. *Transactions on Computers* 65, 2 (2016), 535–547.

Agostino Mascitti, Tommaso Cucinotta, and Mauro Marinoni. 2020. An Adaptive, Utilization-Based Approach to Schedule Real-Time Tasks for ARM Big.LITTLE Architectures. *SIGBED Rev.* 17, 1 (2020), 18–23. DOI: http://dx.doi.org/10.1145/3412821.3412824

Gilberto Melfe, Alcides Fonseca, and João Paulo Fernandes. 2018. Helping developers write energy efficient Haskell through a data-structure evaluation. In *GREENS@ICSE*, Ivano Malavolta, Rick Kazman, and João Saraiva (Eds.). ACM, New York, NY, USA, 9–15. DOI: http://dx.doi.org/10.1145/3194078.3194080

Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. 2017. DawnCC: Automatic Annotation for Data Parallelism and Offloading. *Transactions on Architecture and Code Optimization* 14, 2 (2017), 13:1–13:25.

Sparsh Mittal. 2016. A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors. *Comput. Surv.* 48, 3 (2016), 45:1–45:38. DOI: http://dx.doi.org/10.1145/2856125

Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *Comput. Surv.* 47, 4 (2015), 69:1–69:35.

Mehrzad Nejat, Madhavan Manivannan, Miquel Pericas, and Per Stenstrom. 2020. Coordinated Management of Processor Configuration and Cache Partitioning to Optimize Energy under QoS Constraints. In *IPDPS*. IEEE, USA, 303–313. DOI: http://dx.doi.org/10.1109/IPDPS.2019.00040

Jose Leal Domingues Neto, Se-Young Yu, Daniel F. Macedo, José Marcos S. Nogueira, Rami Langar, and Stefano Secci. 2018. ULOOF: A User Level Online Offloading Framework for Mobile Edge Computing. *IEEE Trans. Mob. Comput.* 17, 11 (2018), 2660–2674. DOI: http://dx.doi.org/10.1109/TMC.2018.2815015

Pengcheng Nie and Zhenhua Duan. 2012. Efficient and Scalable Scheduling for Performance Heterogeneous Multicore Systems. *J. Parallel Distrib. Comput.* 72, 3 (2012), 353–361.

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

Wellington Oliveira, Renato Oliveira, Fernando Castor, Gustavo Pinto, and João Paulo Fernandes. 2021. Improving energy-efficiency by recommending Java collections. *Empir. Softw. Eng.* 26, 3 (2021), 55. DOI: http://dx.doi.org/10.1007/s10664-021-09950-y

Anne-Cecile Orgerie, Marcos Dias de Assunção, and Laurent Lefevre. 2014. A Survey on Techniques for Improving the Energy Efficiency of Large-scale Distributed Systems. *ACM Comput. Surv.* 46, 4 (2014), 47:1–47:31. DOI: http://dx.doi.org/10.1145/2532637

Jinsu Park, Seongbeom Park, and Woongki Baek. 2018. RPPC: A Holistic Runtime System for Maximizing Performance Under Power Capping. In *CCGRID*. IEEE, Washington, DC, USA, 41–50.

Suraj Paul, Navonil Chatterjee, Prasun Ghosal, and Jean-Philippe Diguet. 2020. Adaptive Task Allocation and Scheduling on NoC-Based Multicore Platforms with Multitasking Processors. *ACM Trans. Embed. Comput. Syst.* 20, 1, Article 4 (2020), 26 pages. DOI: http://dx.doi.org/10.1145/3408324

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

Vinicius Petrucci, Orlando Loques, Daniel Mossé, Rami Melhem, Neven Abou Gazala, and Sameh Gobriel. 2015. Energy-Efficient Thread Assignment Optimization for Heterogeneous Multicore Systems. *ACM Trans. Embed. Comput. Syst.* 14, 1 (2015), 15:1–15:26.

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.

Gabriel Poesia, Breno Campos Ferreira Guimarães, Fabricio Ferracioli, and Fernando Magno Quintão Pereira. 2017. Static placement of computation on heterogeneous devices. *PACMPL* 1, OOPSLA (2017), 50:1–50:28.

Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *PLDI*. ACM, New York, NY, USA, 31–47.

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

Basireddy Karunakar Reddy, Amit Kumar Singh, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2020. AdaMD: Adaptive Mapping and DVFS for Energy-Efficient Heterogeneous Multicores. *Trans. Comput. Aided Des. Integr. Circuits Syst.* 39, 10 (2020), 2206–2217. DOI: http://dx.doi.org/10.1109/TCAD.2019.2935065

Uladizislau Rezki and Vitaly Wool. 2015. Doing big.LITTLE right: little and big obstacles. (2015).

Julius Roeder, Sebastian Altmeyer, Benjamin Rouxel, and Clemens Grelck. 2021. Energy-aware Scheduling of Multi-version Tasks onHeterogeneous Real-time Systems. In *SAC*. ACM, New York, NY, USA, 1–10.

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, New York, NY, USA, 49–68.

Benjamin Schiller, Clemens Deusser, Jerónimo Castrillón, and Thorsten Strufe. 2016. Compile- and run-time approaches for the selection of efficient data structures for dynamic graph analysis. *Appl. Netw. Sci.* 1 (2016), 9. DOI: http://dx.doi.org/10.1007/s41109-016-0011-2

Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. 2009. HASS: A Scheduler for Heterogeneous Multicore Systems. *SIGOPS Oper. Syst. Rev.* 43, 2 (2009), 66–75.

Zhen-Jun Shi. 2004. Convergence of Line Search Methods for Unconstrained Optimization. *Appl. Math. Comput.* 157, 2 (2004), 393–405. DOI: http://dx.doi.org/10.1016/j.amc.2003.08.058

Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. In *SPAA*. ACM, New York, NY, USA, 68–70.

Amit Kumar Singh, Somdip Dey, Klaus D. McDonald-Maier, Basireddy Karunakar Reddy, Geoff V. Merrett, and Bashir M. Al-Hashimi. 2020. Dynamic Energy and Thermal Management of Multi-core Mobile Platforms: A Survey. *Des. Test* 37, 5 (2020), 25–33. DOI: http://dx.doi.org/10.1109/MDAT.2020.2982629

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

Jyothi Krishna Viswakaran Sreelatha, Shankar Balachandran, and Rupesh Nasre. 2018. CHOAMP: Cost Based Hardware Optimization for Asymmetric Multicore Processors. *Trans. Multi-Scale Computing Systems* 4, 2 (2018), 163–176.

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, New York, NY, USA, 89–100.

Jürgen Teich, Pouya Mahmoody, Behnaz Pourmohseni, Sascha Roloff, Wolfgang Schröder-Preikschat, and Stefan Wildermann. 2021. Run-Time Enforcement of Non-functional Program Properties on MPSoCs. In *A Journey of Embedded and Cyber-Physical Systems - Essays Dedicated to Peter Marwedel on the Occasion of His 70th Birthday*, Jian-Jia Chen (Ed.). Springer-Verlag, Berlin, Heidelberg, 125–149. DOI: http://dx.doi.org/10.1007/978-3-030-47487-4_9

Stavros Tzilis, Pedro Trancoso, and Ioannis Sourdis. 2019. Energy-Efficient Runtime Management of Heterogeneous Multicores using Online Projection. *TACO* 15, 4 (2019), 63:1–63:26.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *CASCON*. IBM Press, Indianapolis, US, 13–.

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

Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012b. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *ISCA*. IEEE Computer Society, Washington, DC, USA, 213–224.

Zheng Wang and Michael F. P. O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. DOI: http://dx.doi.org/10.1109/JPROC.2018.2817118

Anton Weber, Kim-Anh Tran, Stefanos Kaxiras, and Alexandra Jimborean. 2017. Decoupled Access-Execute on ARM big.LITTLE. *CoRR* abs/1701.05478 (2017), 1–25. arXiv:1701.05478 http://arxiv.org/abs/1701.05478

Youfeng Wu and James R. Larus. 1994. Static Branch Frequency and Program Profile Analysis. In *MICRO*. ACM, New York, NY, USA, 1–11. DOI: http://dx.doi.org/10.1145/192724.192725

A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh. 2015. Neural acceleration for GPU throughput processors. In *MICRO*. IEEE, New York, NY, USA, 482–493.

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

Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing Ordered Graph Algorithms with GraphIt. In *CGO*. Association for Computing Machinery, New York, NY, USA, 158–170. DOI: http://dx.doi.org/10.1145/3368826.3377909