Mapping Computations in Heterogeneous Multicore Systems with Statistical Regression on Inputs

Junio C R da Silva, Lorena Leão, Vinicius Petrucci, Abdoulaye Gamatié, Fernando Pereira

▶ **To cite this version:**

Junio C R da Silva, Lorena Leão, Vinicius Petrucci, Abdoulaye Gamatié, Fernando Pereira. Mapping Computations in Heterogeneous Multicore Systems with Statistical Regression on Inputs. SBESC 2020 - 10th Brazilian Symposium on Computing Systems Engineering, Nov 2020, Virtual, Brazil. 10.1109/SBESC51047.2020.9277863 . lirmm-03018543

HAL Id: lirmm-03018543

https://hal-lirmm.ccsd.cnrs.fr/lirmm-03018543

Submitted on 22 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Mapping Computations in Heterogeneous Multicore Systems with Statistical Regression on Inputs

Junio C. R. da Silva*, Lorena Leão*, Vinicius Petrucci†, Abdoulaye Gamatié‡, Fernando M. Q. Pereira*
*Universidade Federal de Minas Gerais
†University of Pittsburgh & UFBA
‡LIRMM, CNRS & Université de Montpellier, France

***Abstract*—Heterogeneous multicore systems, such as ARM big.LITTLE, use different types of processors to conciliate high performance with low energy consumption. A question that concerns such systems is how to find the best hardware configuration (type and frequency of processors) for a program. Current solutions are either completely dynamic, based on in-vivo profiling, or completely static, based on supervised machine learning. Whereas the former approach can bring unwanted runtime overhead, the latter fails to account for diversity in program inputs. In this paper, we design and evaluate a compilation strategy, JINN-C, that perform statistical regression on function arguments, so as to match parameters with ideal hardware configurations at runtime. We show that JINN-C, implemented in the Soot compiler, can predict the best configuration for a suite of Java and Scala programs running on an Odroid XU4 board, while outperforming prior techniques such as ARM's GTS and CHOAMP, a recently released static program scheduler.**

***Index Terms*—Regression, Function, Heterogeneous Multicore Architectures, Scheduling, big.LITTLE**

I. INTRODUCTION

Modern multicore platforms provide developers with a suite of technologies to produce code that is more energy-efficient [1]. Among these technologies, two stand out today: dynamic voltage & frequency scaling [2] and heterogeneous architectures in which different processors are combined into the same chip. The ARM big.LITTLE design, typically found in smartphones, exemplifies the latter technology [3]. As an example, the Samsung Exynos 5422 chip has eight processors, four fast, but power-hungry (the "big" cores), and four slow, but more power efficient (the"LITTLE" cores). Additionally, processors have up to 19 different frequency levels, going from 200MHz to 1.5GHz in LITTLE cores, and up to 2.0GHz in big cores [4]. The combination of heterogeneous processors, each one featuring multiple frequency levels, gives programmers a vast suite of configurations to choose from when running their applications. However, performing this choice is challenging [5]–[10].

A recent solution to this problem is CHOAMP, a compilation technique designed by Sreelatha *et al* [11]. CHOAMP uses supervised machine learning to map program functions to the configuration that best fits them. Sreelatha *et al.* try to capture characteristics of the target architecture's runtime behavior. They use this knowledge to predict the ideal configuration to a program, given its syntactic characteristics. The beauty of Sreelatha *et al.*'s approach is the fact that it is fully static: interventions on the program remain confined into the compiler, and no extra runtime support is required from the hardware. Such *modus operandi* has been made popular by Shelepov *et al* [12]'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 scheduling decisions. As we explain in Section II, there exist programs for which the best hardware configuration for a given function varies depending on the function's inputs. Program inputs, thus, have the potential to be explored in determining good mappings between programs and hardware configurations on heterogeneous multicore systems.

Our Solution. In this paper, we introduce a compilation approach to map program parts to hardware configurations that can optimize resource usage. In contrast to prior work, our technique explicitly takes function inputs into consideration when deciding which hardware configurations to schedule. As we discuss in Section III, our idea is based on statistical regression. 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 onto SOOT [13], a bytecode optimizer, and have evaluated it onto an Odroid XU4 big.LITTLE architecture. SOOT lets us use the knowledge built during training to generate code that, at runtime, changes the hardware configuration per program function. We call this code generator the JINN-C compiler, a tool that reads and outputs Java bytecodes. Although we work at the granularity of functions, nothing hinders our approach from being applied onto smaller (or larger) program parts. As we explain in Section IV, we have evaluated JINN-C on the subset of the Program Based Benchmark Suite [14] used by Acar *et al* [15], and on programs from Renaissance –a benchmark suite introduced in 2019 [16]– that we have been able to port to the embedded board that we use. 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 [17].

II. Overview

The term *hardware configuration* is used with different meanings by different researchers, thus we shall restrict ourselves to the following definition. 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 $h = \{(\pi, f) \mid \pi \in \Pi, f \in \mathit{Freq}(\pi)\}$. If $(\pi_i, f_j) \in h$, for some $f_j \in \mathit{Freq}(\pi_i)$, then processor π_i is said to be *active* in h with frequency f_j, otherwise it is said to be *inactive*.

For example, the 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 System-on-Chip (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 $\{b_0, b_2\} \times 2.0GHz, \{L_1, L_2, L_3\} \times 1.3GHz$.

A. 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 on both cores. Nevertheless, we know of research artifacts that take these differences into consideration; for example, CHOAMP is a recent technique in this direction [11].

The CHOAMP scheduler matches program features, such as branches, barriers, reductions and memory access operations with the ideal configuration for each function. After CHOAMP trains a regression model, the same configuration decision applies for a function, regardless of its actual inputs. Purely static approaches are known to fall short in adapting to a variety of user inputs, which can negatively impact the program execution [18]. Example 1 illustrates this point by showing that it is possible to find different programs for which the ideal hardware configuration varies according to their inputs.

Example 1: 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 threads we have, the less they benefit from the

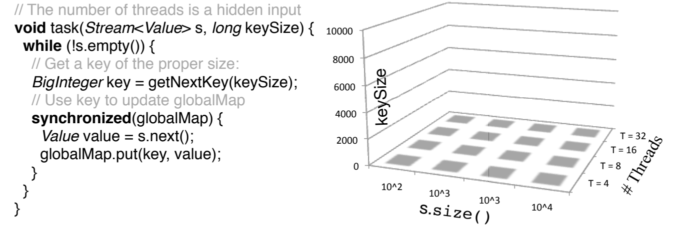


Fig. 1. A program, and its input space.

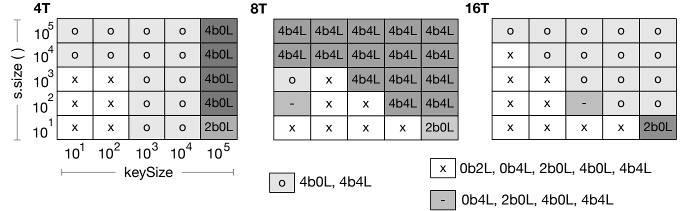


Fig. 2. The ideal configuration for different parameters of the TASK function seen in Figure 1, for 4, 8 and 16 threads, measured on an Odroid XU4 with the *userspace* governor, and default configuration 4b4L. Names inside boxes indicate the best configuration(s) for that input. 'X' indicates setups with three or more configurations tied as best. Notice that 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.

big cores. Indeed, as already observed by Kim *et al* [19], context switches are more expensive in the big than in the LITTLE cores. So are memory accesses: on the Odroid XU4, L2 latency for big cores is 21 cycles while for LITTLE cores it is 10 [4]. 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 1 in algorithms like Integer Sort, a benchmark used by Sreelatha *et al* [11], which we re-evaluate in Section IV.

B. Accounting for Energy Efficiency

If we consider energy as a dimension of efficiency, then choosing good hardware configurations becomes more challenging. Because low-frequency cores tend to be more power-efficient than high-frequency processors, we end up having more incentive to use them. 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 measurement apparatus described by Silva *et al* [20], which samples power at 20KHz.

Example 2: We have used the power measurement apparatus shown in Figure 3(a) to plot runtime and energy consumption for the function TASK earlier seen in Fig. 1, considering two different input sets. Figure 3(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* [20], we call the chart relating runtime and energy a *constellation*. The constellation in Figure 3(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 3(d) outlines. This constellation shows how TASK performs in a *synchronization-heavy* environment.

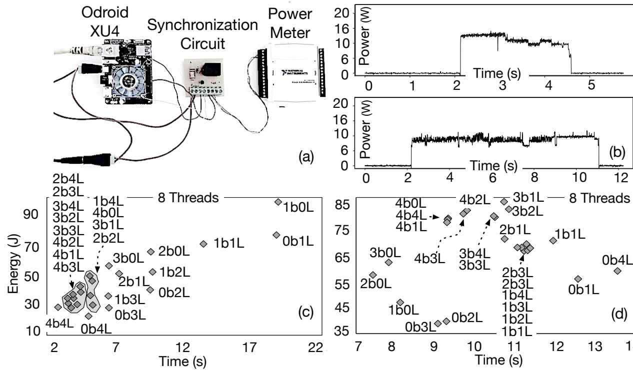


Fig. 3. (a) The energy measurement apparatus. (b) Power charts for configuration 4b4L when running with different inputs. (c) Constellation for synchronization-free input set. (d) Constellation for synchronization-heavy input set. Big cores run at 2.0*GHz* and LITTLE cores run at 1.5*GHz*.

Example 2 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 good 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 [9], [10], [21] resorts to online monitoring; however, this may pose a potential overhead on the system as the number of programs and hardware configuration increase.

III. Solution

We apply statistical regression on the arguments of a function to determine the ideal hardware configurations for different inputs of that function. The implementation of this idea asks for the modification of programs. The pipeline in Figure 4 summarizes our code transformation techniques. To ease our presentation, we shall be using source code in our examples, as seen in Figure 4. However, our solution works at the Java bytecode level and our interventions happen within the compiler; more precisely in the program's intermediate representation. Working at the bytecode level lets us optimize programs written in different languages that run on the Java Virtual Machine. In Section IV we shall validate our techniques using Java and Scala benchmarks.

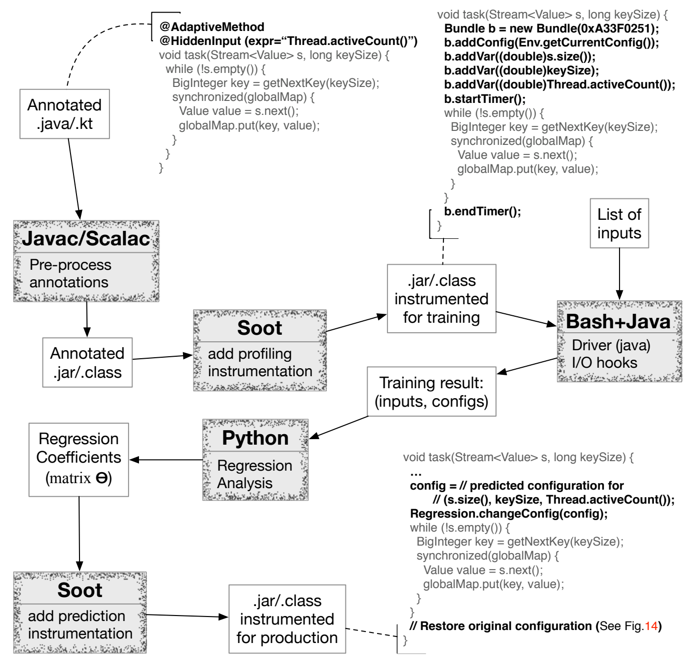


Fig. 4. The execution pipeline of JINN-C.

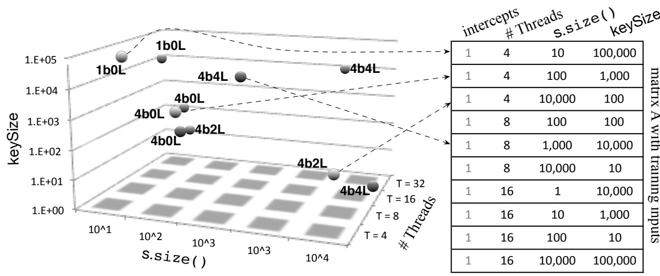
A. Multiple Linear Regression

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 lines 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.

The matrix A of independent variables. 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 lines, 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.

Example 3: Figure 5 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

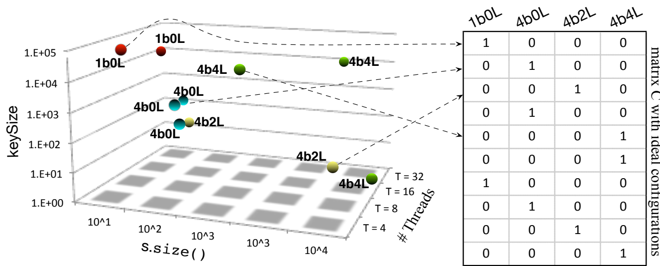


intercepts	# Threads	s.size()	keySize
1	4	10	100,000
1	4	100	1,000
1	4	10,000	100
1	8	100	100
1	8	1,000	10,000
1	8	10,000	10
1	16	1	10,000
1	16	10	1,000
1	16	100	10
1	16	10,000	100,000

Fig. 5. Training set for the TASK method (Fig. 1). The table on the right is matrix A of independent variables.

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 row of C is a unitary vector e_i, which has all the components set to zero, except its ith 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 4: Figure 6 reuses the ten samples seen in Example 3 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 IV discusses the criteria used to build the search space of allowed configurations.

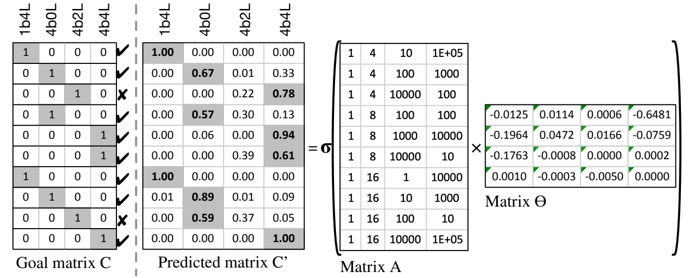


1b0L	4b0L	4b2L	4b4L
1	0	0	0
0	1	0	0
0	0	1	0
0	1	0	0
0	0	0	1
0	0	0	1
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Fig. 6. Matrix of independent variables built for ten different invocations of function TASK in Figure 1.

Finding the parameter matrix Θ. The problem of constructing a predictor based on multivariate linear regression consists in finding a matrix Θ that maximizes the quantity of correct predictions on the training set. The underlying assumption is that if Θ approximates the behavior of the training set, then it is likely to yield also good results on the test set. There exist efficient techniques to find Θ –*gradient descent* being the best well-known [22]. Because our model involves only searches over a linear space, gradient descent converges quickly to a global optimum. By a linear search space, we mean that, for each element (i, j) in C, we have that: $C_{ij} = \Theta_{0j} + \alpha_{i1}\Theta_{1j} + \ldots + \alpha_{im}\Theta_{mj}$. Therefore, non-linear expressions such as $\alpha_{ip}\alpha_{iq}$ bear no impact on C_{ij}.

Example 5: Figure 7 shows a possible matrix Θ that gradient descent finds for the TASK function, when given the training set seen in Figures 5 and 6. 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 1.00. We are using only two decimal digits; hence, rounding errors prevent us from obtaining 1.00 in every line. 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 7 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 predicts every row of C. However, this matrix, which fits too well the training set, might not yield good predictions on unseen inputs.



Goal matrix C:

1b4L	4b0L	4b2L	4b4L	
1	0	0	0	✓
0	1	0	0	✓
0	0	1	0	✗
0	1	0	0	✓
0	0	0	1	✓
0	0	0	1	✓
1	0	0	0	✓
0	1	0	0	✓
0	0	1	0	✗
0	0	0	1	✓

Predicted matrix C':

1b4L	4b0L	4b2L	4b4L
1.00	0.00	0.00	0.00
0.00	0.67	0.01	0.33
0.00	0.00	0.22	0.78
0.00	0.57	0.30	0.13
0.00	0.06	0.00	0.94
0.00	0.00	0.39	0.61
1.00	0.00	0.00	0.00
0.01	0.89	0.01	0.09
0.00	0.59	0.37	0.05
0.00	0.00	0.00	1.00

Matrix A:

1	4	10	1E+05
1	4	100	1000
1	4	10000	100
1	8	100	100
1	8	1000	10000
1	8	10000	10
1	16	1	10000
1	16	10	1000
1	16	100	10
1	16	10000	1E+05

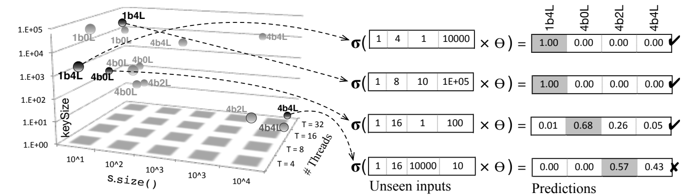
Matrix Θ:

-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

Fig. 7. The result of multivariate linear regression produced by the training set seen in Examples 3 and 4.

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 III-C, 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 6: Figure 8 uses the matrix Θ found in Figure 7 to guess the best configuration for four unseen input sets. These inputs appear as dark spheres in Figure 8. In this example, Θ correctly predicts the ideal configuration for three out of four samples. In one case, the last input in Figure 8, we wrongly predict the best configuration as 4b2L, whereas empirical evidence suggests that it should be 4b4L.



Unseen inputs					Predictions 1b4L	4b0L	4b2L	4b4L	
σ(1	4	1	10000	× Θ) =	1.00	0.00	0.00	0.00	✓
σ(1	8	10	1E+05	× Θ) =	1.00	0.00	0.00	0.00	✓
σ(1	16	1	100	× Θ) =	0.01	0.68	0.26	0.05	✓
σ(1	16	10000	10	× Θ) =	0.00	0.00	0.57	0.43	✗

Fig. 8. The matrix Θ found in Figure 7 used to predict the ideal configuration for four unseen input sets. Inputs used in the training set are the light-grey points, whereas inputs in the test set are dark-grey.

B. 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. The following 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. We do not allow annotations on booleans and characters, as their values do not have a direct conversion to a real (e.g., a double) number.

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, doubleValue() for Double, etc.

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 7: Figure 9 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.

```
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. 9. Instrumented version of two programs. Grey code is from the original method. (Left) Breadth-first search. (Right) Sorting application.

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. 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 pre-determined 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.

C. Code Generation

The product of training is a matrix Θ of floating-point numbers. The matrix Θ is hardcoded into the production code that we want to optimize. Such step happens in the phase labeled "add prediction instrumentation" in Figure 4. 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. The largest element within $\sigma(A_i\Theta)$ determines the next configuration that will be used during the current invocation of f.

IV. EVALUATION

This section demonstrates the effectiveness of our technique when optimizing bytecodes that run on top of the Java Virtual Machine. We compare JINN-C with two approaches: Sreelatha *et al* [11]'s CHOAMP, and ARM's GTS [23]. GTS, short for *Global Task Scheduling*, is Linux' heterogeneity-aware scheduler in our big.LITTLE system.

A. Experimental Setup

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 2.0GHz, and four Cortex A7 cores running at up to 1.5GHz. The board features 2GB of LPDDR3 RAM. We use the energy measurement framework proposed by Bessa *et al* [17]. 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 analyze, instrument and run bytecodes. No modifications have been made in the Java Virtual Machine. Code 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 (see Figure 10). We have used Python 3.4 and Scikit Learn [24] to implement regression. 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 Figure 10. Eight of them were taken from Acar *et*

Source	*Benchmark*	*TTime*	*Lang.*	*LoC*	*W*
[14]	bfs	42m33s	J	353	4
[14]	radixSort	20m51s	J	501	4
[14]	sampleSort	26m17s	J	414	3
[14]	suffixArray	30m12s	J	316	3
[14]	removeDuplicates	30m31s	J	174	4
[14]	convexHull	56m30s	J	499	5
[14]	nearestNeighbors	30m29s	J	715	3
[14]	spanningForest	21m40s	J	410	4
[16]	als	80m12s	S/J	97	1
[16]	philosophers	21m15s	S/J	146	1
[16]	futureGenetic	26m8s	S/J	115	1
[16]	finagleHTTP	225m10s	S/J	119	1
[16]	chiSquare	27m15s	S/J	101	1
[16]	decTree	64m22s	S/J	129	1
JINN-C	collinearPoints	32m1	J	565	3
JINN-C	hashSync	94m7s	J	73	3
JINN-C	insertAndAdd	47m30s	J	130	4
JINN-C	randomNumComp	26m7s	J	89	6

Fig. 10. Benchmarks used for evaluating JINN-C. Column *TTime* shows time to train each benchmark. To train JINN-C, we follow the methodology described in Section III-B1. JINN-C's training time depends on the target application's run time, and on the number of available inputs. *Lang.* contains the source language of benchmarks, where *J* stands for Java and *S* stands for Scala. The *W* column shows the number of *warm-up* executions performed by each application. 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.

al [15], who had selected nine programs from *Problem Based Benchmark Suite* (PBBS) [14] to evaluate concurrency models. The version of PBBS used by Acar *et al* [15] was implemented in C/C++, so we had to reimplement all the benchmarks in Java. We used six programs from the *Renaissance* benchmark collection, which was recently released by Prokopec *et al* [16]. Our criterion when picking up the six 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 extra benchmarks. These programs are typical parallel algorithms. Three of them were taken from public repositories; the fourth, HASHSYNC, was adapted from Butcher *et al* [25]'s book. Figure 10 presents an overview of the benchmarks, as well as basic characteristics of their code.

The Available Inputs. We have augmented every one of our benchmarks with 14 inputs. We have separated 10 of these inputs for training. When evaluating the trained model, for each application we used four new, unseen, and randomly chosen inputs. Sections IV-B and IV-C further discuss the impact of different inputs in the execution time and energy consumption of the applications.

Choice of Regression Target. We optimize one method per benchmark. This method is the routine invoked by the benchmark's driver. This approach is equivalent to doing regression on inputs of the whole program.

Choice of Hardware Configurations. For the sake of reproducibility, when training JINN-C and CHOAMP, we follow the methodology proposed by Sreelatha *et al* [11]. 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 max 2.0GHz) due to thermal issues [26] found in our board. For more details, see our technical report [27]. Therefore, the two adaptive approaches that we use might choose from a pool of ten different hardware configurations: 4b4L at either 1.6 or 1.8GHz (plus LITTLE cores at 1.5GHz), 0b4L at 1.5GHz, 4b0L at either 1.6 or 1.8GHz, etc. The GTS algorithm, however, is allowed to choose among any possible hardware configuration involving big and LITTLE cores, and the different frequency levels available in the hardware.

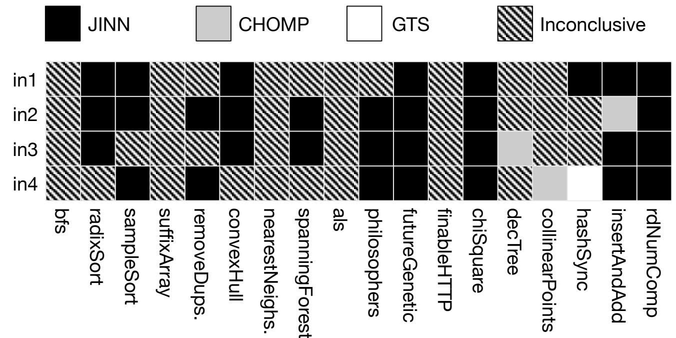


Fig. 11. Summary of runtime comparisons. Boxes indicate best approaches per test input with confidence level of 95%.

B. Results of Performance

Figure 11 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. Figure 12 shows four selected samples used to build Figure 11. 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.

We notice that in 26 cases, out of 72 combinations of [benchmarks $\times$ 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. All the winning configurations, regardless of the technique, featured 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). JINN-C performed rather poorly in COLLINEARPOINTS. Such bad results were due to the fact that we have not chosen good inputs for training. Indeed, the 10 training inputs chosen when optimizing COLLINEARPOINTS find in 4B4L their best

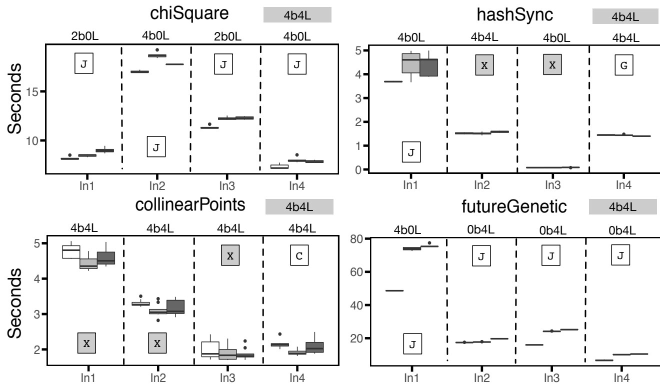


Fig. 12. Execution time of selected benchmarks (full data is available in technical report [27]). Y-axis shows time in seconds. X-axis shows different experiments; each experiment uses different inputs. Boxplots are ordered by JINN-C, CHOAMP and GTS. White boxes with letters identify the technique which 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.

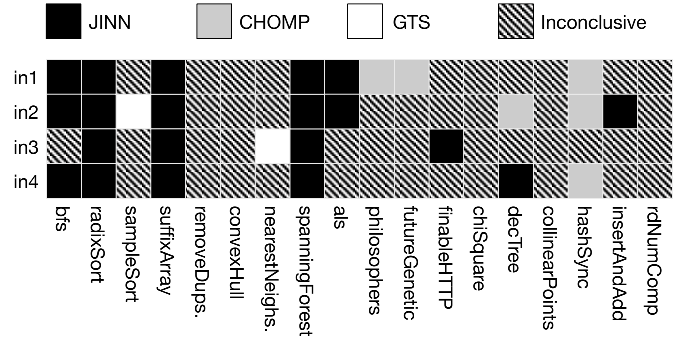


Fig. 13. Summary of energy comparisons (boxes indicate best approaches).

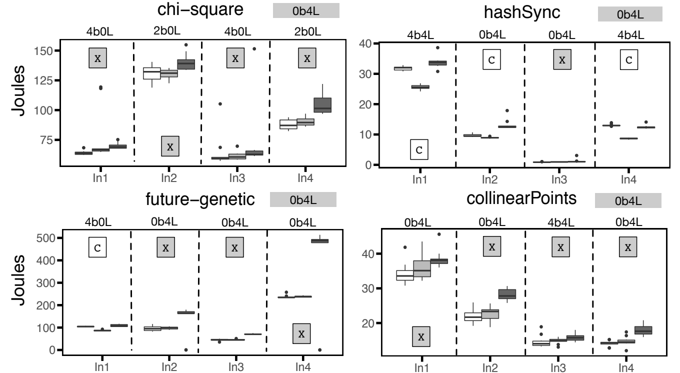


Fig. 14. Energy consumed by the benchmarks in Figure 10 (full data is available in our technical report [27]). Y-axis shows energy in Joules. X-axis shows different experiments. We follow the notation seen in Figure 12.

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. For instance, in CHISQUARE's first input (WORKERS $= 2$, SIZE $= 1023464$), 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, with all values for the p-value less than 0.008. For its second input (WORKERS $= 4$, SIZE $= 2250467$), we observed JINN-C prediction (2b0L) leading to mean run time of 17.00 seconds, CHOAMP to 18,70 and GTS to 17,76.

C. Results of Energy Savings

Figure 13 compares CHOAMP, GTS and JINN-C regarding energy consumption. Selected samples appear in Figure 14. 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 13 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. 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. We believe that 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.

V. RELATED WORK

Our work explores a type of machine learning technique (multivariate linear regression) to solve an instance of program scheduling in heterogeneous architectures. For an overview of the impact of machine learning onto compiler construction, we recommend surveys from Wang *et al* [28] and Ashouri *et al* [29]. For an overview of input-aware compilation techniques in general, we refer the reader to the Related Work section of our technical report [27]. The rest of this section focuses on scheduling for heterogeneous multicore systems.

Much attention has been dedicated to the problem of finding good placements of computation on heterogeneous multicore systems, as Mittal *et al* [30] has summarized in a 2016 survey. 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 [30]–[33].

At the compiler level, Sreelatha *et al* [11]'s CHOAMP, and Krishna *et al* [34]'s SIAM provide solutions to scheduling in big.LITTLE architectures. 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.

VI. CONCLUSION

This paper presented a code generation technique that matches programs to hardware configurations in heterogeneous multicore systems. The key insight of this work was the observation that the values of a function's inputs often provide enough information to predict the best hardware configuration that suits said function. From this observation, we showed how to build predictors based on linear regression on function inputs. Our technique is able to outperform, be it in energy consumption, be it in execution time, the default Linux scheduler (the Global Task Scheduler), and CHOAMP, a recently released tool that predicts the best hardware configuration to a parallel program based on its syntax.

REFERENCES

[1] A.-C. Orgerie, M. D. d. Assunção, and L. Lefevre, "A survey on techniques for improving the energy efficiency of large-scale distributed systems," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 47:1–47:31, 2014. [Online]. Available: http://doi.acm.org/10.1145/2532637

[2] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *HPCA*. Washington, DC, USA: IEEE, 2002, pp. 29–.

[3] M. Hähnel and H. Härtig, "Heterogeneity by the numbers: A study of the odroid xu+e big. little platform," in *HotPower*. Berkeley, CA, USA: USENIX Association, 2014, pp. 3–3.

[4] P. Greenhalgh, "Big.LITTLE processing with ARM cortex-A15 & cortex-A7," San Francisco, CA, US, pp. 1–8, 2011. [Online]. Available: https://www.eetimes.com/document.asp?doc_id=1279167

[5] M. W. Azhar, M. Pericàs, and P. Stenström, "SaC: Exploiting execution-time slack to save energy in heterogeneous multicore systems," in *ICPP*. New York, NY, USA: ACM, 2019, pp. 26:1–26:12.

[6] A. Jundt, A. Cauble-Chantrenne, A. Tiwari, J. Peraza, M. A. Laurenzano, and L. Carrington, "Compute bottlenecks on the new 64-bit arm," in *E2SC*. New York, NY, USA: ACM, 2015, pp. 6:1–6:7.

[7] O. Khan and S. Kundu, "A self-adaptive scheduler for asymmetric multi-cores," in *GLSVLSI*. New York, NY, USA: ACM, 2010, p. 397–400.

[8] M. Nejat, M. Pericàs, and P. Stenström, "QoS-driven coordinated management of resources to save energy in multi-core systems," in *IPDPS*. IEEE, 2019, pp. 303–313.

[9] R. Nishtala, P. M. Carpenter, V. Petrucci, and X. Martorell, "Hipster: Hybrid task manager for latency-critical cloud workloads," in *HPCA*. New York, NY, USA: IEEE, 2017, pp. 409–420.

[10] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N. A. Gazala, and S. Gobriel, "Energy-efficient thread assignment optimization for heterogeneous multicore systems," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 1, pp. 15:1–15:26, 2015.

[11] J. K. V. Sreelatha, S. Balachandran, and R. Nasre, "CHOAMP: cost based hardware optimization for asymmetric multicore processors," *Trans. Multi-Scale Computing Systems*, vol. 4, no. 2, pp. 163–176, 2018.

[12] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: A scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, 2009.

[13] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *CASCON*. Indianapolis, US: IBM Press, 1999, pp. 13–.

[14] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *SPAA*. New York, NY, USA: ACM, 2012, pp. 68–70.

[15] U. A. Acar, A. Charguéraud, A. Guatto, M. Rainey, and F. Sieczkowski, "Heartbeat scheduling: Provable efficiency for nested parallelism," in *PLDI*. New York, NY, USA: ACM, 2018, pp. 769–782.

[16] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder, "Renaissance: Benchmarking suite for parallel applications on the jvm," in *PLDI*. New York, NY, USA: ACM, 2019, pp. 31–47.

[17] T. Bessa, G. Gull, P. Q. ao, M. Frank, J. Nacif, and F. M. Q. ao Pereira, "JetsonLEAP: A framework to measure power on a heterogeneous system-on-a-chip device," *Science of Computer Programming*, vol. 33, no. 1, pp. 1–37, 2017.

[18] P. Nie and Z. Duan, "Efficient and scalable scheduling for performance heterogeneous multicore systems," *J. Parallel Distrib. Comput.*, vol. 72, no. 3, pp. 353–361, 2012.

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

[20] J. C. R. da Silva, F. M. Q. Pereira, M. Frank, and A. Gamatié, "A compiler-centric infra-structure for whole-board energy measurement on heterogeneous android systems," in *ReCoSoC*. Washington, DC, USA: IEEE, 2018, pp. 1–8.

[21] F. David, G. Thomas, J. Lawall, and G. Muller, "Continuously measuring critical section pressure with the free-lunch profiler," *SIGPLAN Not.*, vol. 49, no. 10, pp. 291–307, 2014.

[22] M. A. Cauchy, "Méthode générale pour la résolution des systèmes d'Équations simultanées," *Comptes Rendus Hebd. Séances Acad. Sci.*, vol. 25, no. 10, pp. 536–538, 1847.

[23] B. Jeff, "big.LITTLE technology moves towards fully heterogeneous global task scheduling," ARM, Tech. Rep., 2013, white paper.

[24] 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, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] P. Butcher, *Seven Concurrency Models in Seven Weeks*, 1st ed. Raleigh, NC, US: Pragmatic Bookshelf, 2014.

[26] V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. D. Smith, G.-Y. Wei, and D. Brooks, "Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling," in *MICRO*. USA: IEEE, 2010, p. 77–88.

[27] J. C. Ribeiro da Silva, L. Leão, V. Petrucci, A. Gamatié, and F. M. Quintao Pereira, "Scheduling in Heterogeneous Architectures via Multivariate Linear Regression on Function Inputs," Sep. 2019, working paper or preprint. [Online]. Available: https://hal-lirmm.ccsd.cnrs.fr/lirmm-02281112

[28] Z. Wang and M. F. P. O'Boyle, "Machine learning in compiler optimization," *Proc. IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[29] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *Comput. Surv.*, vol. 51, no. 5, pp. 96:1–96:42, 2018.

[30] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *Comput. Surv.*, vol. 48, no. 3, pp. 45:1–45:38, 2016.

[31] H. Cai, Q. Cao, F. Sheng, M. Zhang, C. Qi, J. Yao, and C. Xie, "Montgolfier: Latency-aware power management system for heterogeneous servers," in *IPCCC*. Washington, DC, USA: IEEE, 2016, pp. 1–8.

[32] A. Garcia-Garcia, J. C. Saez, and M. Prieto, "Contention-aware fair scheduling for asymmetric single-isa multicore systems," *IEEE Trans. Computers*, vol. 67, no. 12, pp. 1703–1719, 2018.

[33] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *ISCA*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 213–224.

[34] J. Krishna and R. Nasre, "Optimizing graph algorithms in asymmetric multicore processors," *Trans. on CAD of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2673–2684, 2018. [Online]. Available: https://doi.org/10.1109/TCAD.2018.2858366