Scheduling in Heterogeneous Architectures via Multivariate Linear Regression on Function Inputs

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

Scheduling in Heterogeneous Architectures via Multivariate Linear Regression on Function Inputs

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

Heterogeneous multicore systems, such as the ARM big.LITTLE, feature a single instruction set with different types of processors to conciliate high performance with low energy consumption. An important question concerning such systems is how to determine the best hardware configuration for a particular program execution. The hardware configuration consists of the type and the frequency of the processors that the program can use at runtime. Current solutions are either completely dynamic, e.g., based on in-vivo profiling, or completely static, based on supervised machine learning approaches. Whereas the former solution might bring unwanted runtime overhead, the latter fails to account for the diversity in program inputs. In this paper, we show how to circumvent this last shortcoming. To this end, we provide a suite of code transformation techniques that perform numeric regression on function arguments, which can have either scalar or aggregate types, so as to match parameters with ideal hardware configurations at runtime. We have designed and implemented our approach on top of the Soot compilation infrastructure, and have applied it onto programs available in the PBBS and Renaissance suites. We show that we can consistently predict the best configuration for a large class of programs running on an Odroid XU4 board, outperforming other techniques such as ARM's GTS or 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. 2017. Scheduling in Heterogeneous Architectures via Multivariate Linear Regression on Function Inputs. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 33 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern multicore platforms provide developers with a suite of technologies to produce code that is more energy-efficient (Orgerie et al. 2014). Among these technologies, two stand out today: dynamic voltage & frequency scaling (Semeraro et al. 2002) and single-ISA heterogeneous architectures in which different processors are combined into the same chip. The ARM big.LITTLE design exemplifies the latter technology (Hähnel and Härtig 2014). Processors using both these technologies are today commonly found in smartphones and embedded systems. As an example, the Samsung Exynos 5422 chip has eight processors, four fast, but power hungry (the so called "big" cores), and four slow, but more power parsimonious (thus called "LITTLE" cores). Each processor has up to 19 different frequency levels, going from 200MHz to 1.5GHz in the LITTLE processors, and from 200MHz to 2.0GHz in the big cores (Greenhalgh 2011). The combination of fast and slow processors, each one featuring multiple frequency levels, gives programmers a vast suite of

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

configurations to choose from when running their applications. However, performing this choice is a challenging task (Jundt et al. 2015; Nishtala et al. 2017; Petrucci et al. 2015).

The craft of compilers that try to map program parts to different hardware configurations (combinations of cores and frequencies) is a research topic that has been receiving considerable attention in the last decade, in part due to the increasing popularity of systems formed by CPUs and GPUs (Garland and Kirk 2010; Nickolls and Dally 2010). In the CPU-GPU case, the problem of mapping program parts to hardware involves dealing with at least two instruction sets, such as x86 for the host CPU, and TASS for the hosted GPU, for instance. If the same function is allowed to run onto both processors, it must be cloned at the binary level (Poesia et al. 2017). In this paper, we focus on the same problem: scheduling of computations; however, on a different setting: same-ISA architectures. In this context, the same program might run in different kinds of processors.

The current state-of-the-art solution to this problem is CHOAMP, a compilation technique invented by Sreelatha et al. (2018). 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. In their words, "*static schedulers scale better with the number of cores as well as program complexity*". Such view has been made popular by Shelepov et al. (2009) through the success enjoyed by HASS, a scheduler for same-ISA heterogeneous systems that leverages architectural signatures (e.g., cache miss rates) generated offline.

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 2, it is regularly possible to find out programs for which the best hardware configuration for a given function varies depending on the function's inputs. Some programs used in the original description of CHOAMP, such as integer sort, bear this property. We make the case that inputs are key to determine good matchings between programs and configurations supported by the evidence that such matchings 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 that optimize resource usage. In contrast to prior work, our technique explicitly takes inputs into consideration when deciding which hardware configurations to use. As we discuss in Section 3, our idea is based on offline training and 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 unseen inputs. One of the contributions of this work is an empirical demonstration that this universe of solutions tends to be convex. As we show in Section 4.7, 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. The consequence of this observation is that derivative-based search methods are expected to converge to an optimal result, and linear regression tends to accurately predict this optimum.

Our Results. We have implemented our technique onto Soot (Vallée-Rai et al. 1999), a bytecode optimizer, and have tested 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 4, we have evaluated Jinn-C on the subset of the Program Based Benchmark Suite (Shun et al. 2012) used by Acar et al. (2018), and on the benchmarks from Renaissance –a collection introduced in 2019 (Prokopec et al. 2019)– that we have been able to port to the embedded board that we use. An interesting aspect of our approach is that the type of regression that we advocate in this paper is agnostic to the objective function. In particular, we show how Jinn-C is able to reduce either the execution time or the energy consumption of programs. The ideal hardware configuration is the one that optimizes for such a particular objective function. We measure energy for the entire board using physical probes (Bessa et al. 2017), and, even if we consider all the power overhead of the peripherals, our results are easy to reproduce. Below we summarize the benefits of our solution in the context of the existing literature:

Adaptive: contrary to previous purely static solutions to the problem of finding ideal hardware configurations to program parts, our technique is able to take input data –information known at runtime only– into consideration when choosing configurations.

Simple: we show that, for typical benchmarks used in high-performance computing, either the value of scalar inputs, or the size of aggregate inputs already yield enough information to effectively feed linear regression models.

Effective: in most of our benchmarks, only a few different 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 is linear on the number of inputs, not on their sizes. This computational complexity is $O(1)$ per hot function.

Automatic: our approach requires a minimum of interference from developers. Developers annotate which functions must be adapted. We chose this approach for simplicity. For zero programming overhead, we could discover hot functions via profiling, for instance.

Easily-deployable: our solution does not require runtime monitoring; thus, it can be deployed in any hardware and operating system, independent on them providing performance counters. We require only the capability to change the hardware configuration at runtime.

2 OVERVIEW

Heterogeneous multi-core architectures exist in a number of flavors. Architectures combining processors that run different instruction sets are called *multiple-ISA*. Typically, some cores address vector/data-level parallelism, whereas others benefit more from instruction-level parallelism. Examples include the CELL processor (Donaldson et al. 2008), and the CPU-GPU systems (Sorensen et al. 2018). In contrast, architectures featuring different processors that run the same instruction set are called Single-ISA (Kumar et al. 2005). Single-ISA systems move from the compiler towards the runtime environment (the operating system or the hardware itself) the responsibility of mapping the program code to hardware configurations. Nevertheless, as previous work has demonstrated, there are benefits to bringing this awareness back into the code generation phase (Sreelatha et al. 2018). Such is also the position of this paper. Because *hardware configuration* is an expression used with different meanings by different researchers, 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 $\{L_0, L_1, L_2, L_3\}$. Each big core has 19 frequency levels ($200MHz, 300MHz, \ldots, 1.9GHz, 2.0GHz$). Each LITTLE core has 14 frequency levels ($200MHz, 300MHz, \ldots, 1.4GHz, 1.5GHz$). This SoC supports any number of active processors; however, big cores must always use the same frequency level. The same holds true for LITTLE cores. In this setting, an example of hardware configuration would be $(b_0, 2.0GHz)$, $(b_2, 2.0GHz)$, $(L_1, 1.3GHz)$, $(L_2, 1.3GHz)$, $(L_3, 1.3GHz)$.

Example 2.2 describes a big.LITTLE architecture: a design introduced by ARM to denote architectures that combine high and low frequency clusters of cores. This design is today very popular in the implementation of smartphones, being used in models produced by Allwinner, HiSilicon, LG, MediaTek, Qualcomm, Samsung and Renesas, for instance. Yet, in spite of its rising popularity, big.LITTLE is far from being the only single-ISA heterogeneous architecture available today at a relatively low cost. ARM itself, in partnership with NVIDIA, has designed technologies such as Tegra (Ditty et al. 2014), which came before the big.LITTLE model, and DynamicIQ[1], which makes it more granular, allowing clusters of cores with different performance and power characteristics.

Adaptive Compilation. The notion of hardware configuration naturally leads to an interesting problem in the field of *adaptive compilation*. In the words of Cooper et al., "*an adaptive compiler uses a compile-execute-analyze feedback loop to find the combination of optimizations and parameters that minimizes some performance goal, such as code size or execution time*". In this paper we are interested in solving the adaptive compilation problem that we define below:

Definition 2.3. Input-Aware Scheduling in Single-ISA Heterogeneous Architectures (Isha) **Input**: a program P, its input i, a set of hardware configurations $H = \{h_1, \ldots h_n\}$, and a cost function $O_P^i : H \mapsto \mathbb{R}$, which determines the cost of running P 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_P^i.

We believe that this paper provides the first solution to Isha. However, this problem is part of a more general family of compiler-related problems, henceforth called *Scheduling of Programs in Heterogeneous Architectures* (Spha). Given a program P, Spha asks for a new version P' of it, which uses the hardware configuration that best suits different *runtime conditions*. The program input is a type of runtime condition, but other conditions exist. Examples include number of resident processes, ratio of cache misses, quantity of context switches, etc. Solutions to Spha run aplenty in the literature. Section 5 explains how our work stands among them.

2.1 Core Configuration in Single-ISA Heterogeneous Architectures

Mainstream compilers, such as Gcc or Clang, which generate code for the systems previously mentioned, 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 –CHOAMP being the most recent technique in this direction (Sreelatha et al. 2018). The compiler technique proposed by CHOAMP tries to match program features, such as syntax denoting branches, barriers, reductions and memory access operations with the ideal configuration for each function. CHOAMP has been tried on the OpenMP version of the NAS benchmark suite (Bailey et al. 1991) with great benefits: on average, it could produce code that was 65% more energy-efficient than its counterparts.

[1] https://developer.arm.com/technologies/dynamiq

```
1  // The number of threads is a hidden input
2  void task(Stream<Value> s, long keySize) {
3    while (!s.empty()) {
4      // Get a key of the proper size:
5      BigInteger key = getNextKey(keySize);
6      // Use key to update globalMap
7      synchronized(globalMap) {
8        Value value = s.next();
9        globalMap.put(key, value);
10     }
11   }
12 }
```

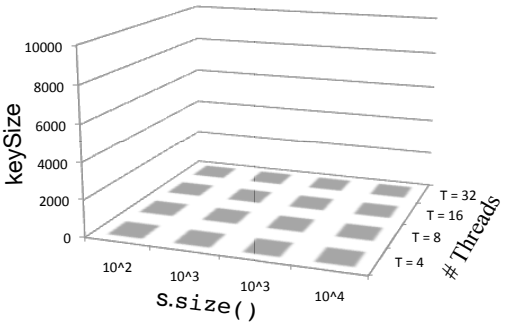


Fig. 1. A program, and its input space.

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, even before the advent of CHOAMP and similar techniques. 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 a *a hidden 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.

Example 2.4 is interesting because the ideal configuration for TASK varies even for very large values of S.SIZE() and KEYSIZE. 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 big cores. Indeed, as already observed by Kim et al. (2014), 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 cycles (Greenhalgh 2011). Furthermore, the larger the size of the input streams, the more often we access the synchronized region between lines 7 and 10 of Figure 1. It is worth noting that we can observe results similar to those seen in Example 2.4 in algorithms like Integer Sort, a benchmark used by Sreelatha et al.. We evaluate a Java implementation of this algorithm in Section 4.

2.2 Accounting for Energy Efficiency

Today, optimizing a program for energy is as important as optimizing for performance (Cao et al. 2012; Kambadur and Kim 2014; Pinto et al. 2014). Such importance comes with extra difficulties: once we add in energy efficiency alongside runtime as another optimization dimension, the impact

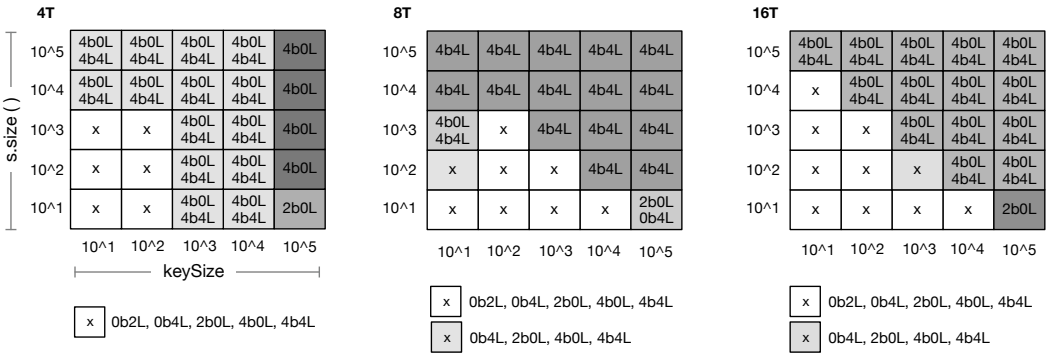


Fig. 2. The ideal configuration for different parameters of the TASK function seen in Figure 1, for 4, 8 and 16 threads, measured on an Odroid XU4 with the *userspace* governor, and default configuration 4b4L. The name(s) inside each box indicate the best configuration(s) for that input. 'X' indicates setups with three or more configurations tied as best. To produce these charts, we followed a methodology yet to be described in Section 4.7. 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.

of program inputs onto the choice of the ideal configuration becomes much higher. Because low-frequency cores tend to be more power efficient than high-frequency processors, we end up having more incentive to use them. However, these low-frequency cores also tend to take longer to finish tasks; consequently, using more energy to perform a job. This observation is critical in battery-powered devices, such as smartphones. The next example analyzes such tradeoffs.

Example 2.5. 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 da Silva et al., 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.

We found the results shown in Example 2.5 rather unexpected, given how drastically changes in inputs modify the disposition of hardware configurations in the constellations. The best energy and time configuration in the CPU-heavy setting, 4b4L, happens to be 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. As a typical way to handle the lack of information at compile time, researchers have been resorting to online monitoring. In this case, an *in-vivo* profiler, à la FREELUNCH (David et al. 2014), constantly verifies hardware state, and takes core configuration decisions based on dynamic information. This approach has been adopted in systems such as OCTOPUSMAN (Petrucci et al. 2015) and HIPSTER (Nishtala et al. 2017). Yet, the same problems pointed by Nie and Duan already in 2012 persist: "*online monitoring approaches had to trace threads' execution on all core types, which is impractical as the number of core types grows.*" This observation,

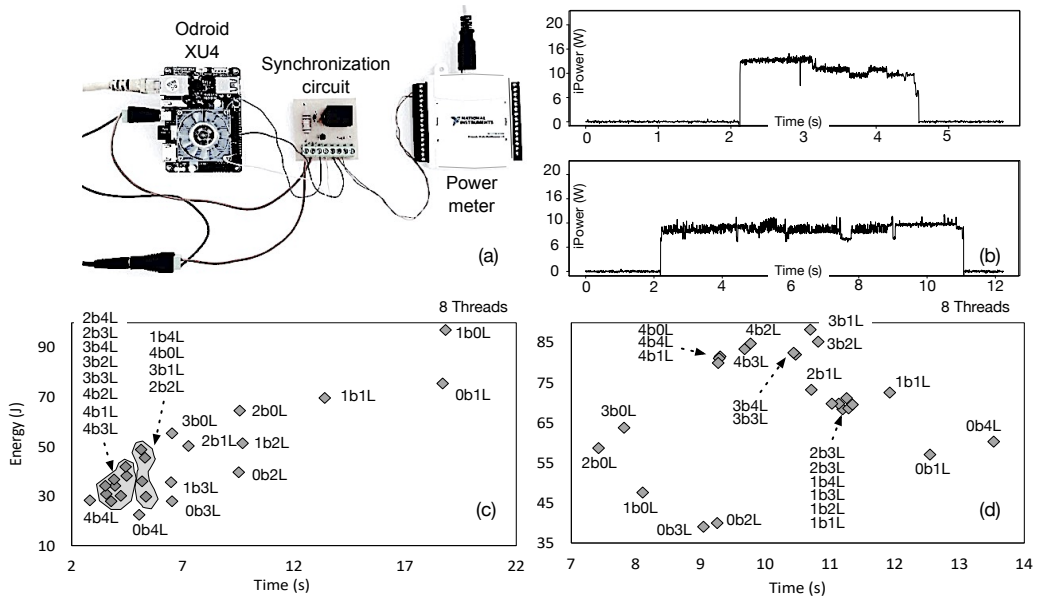


Fig. 3. (a) The energy measurement apparatus. (b) Instantaneous power charts for configuration 4b4L when running with different inputs. (c) Constellation for synchronization-free input set. (d) Constellation for synchronization-heavy input set. Frequencies are set to 2.0*GHz* for big, and 1.5*GHz* for LITTLE cores.

together with the examples discussed in this section, has motivated the contributions of this paper, which we shall detail in Section 3.

3 SOLUTION

We apply statistical regression on the arguments of a function to determine the ideal hardware configurations for different inputs of that function. The effective implementation of this idea asks for the parsing and modification of programs. The pipeline in Figure 4 provides an overview of our code transformation techniques. To ease our presentation, we shall be using source code in all our examples, as seen in that figure. However, our solution works at the Java bytecode level and all our interventions on the program 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 brings one major advantage: we can optimize programs written in different languages that run on the Java Virtual Machine. Indeed, 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. We explore linear regression to build a prediction model that can match actual function parameters with resource-efficient hardware configurations. Because a function might have several parameters, we use multiple linear regression when building predictors. 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*

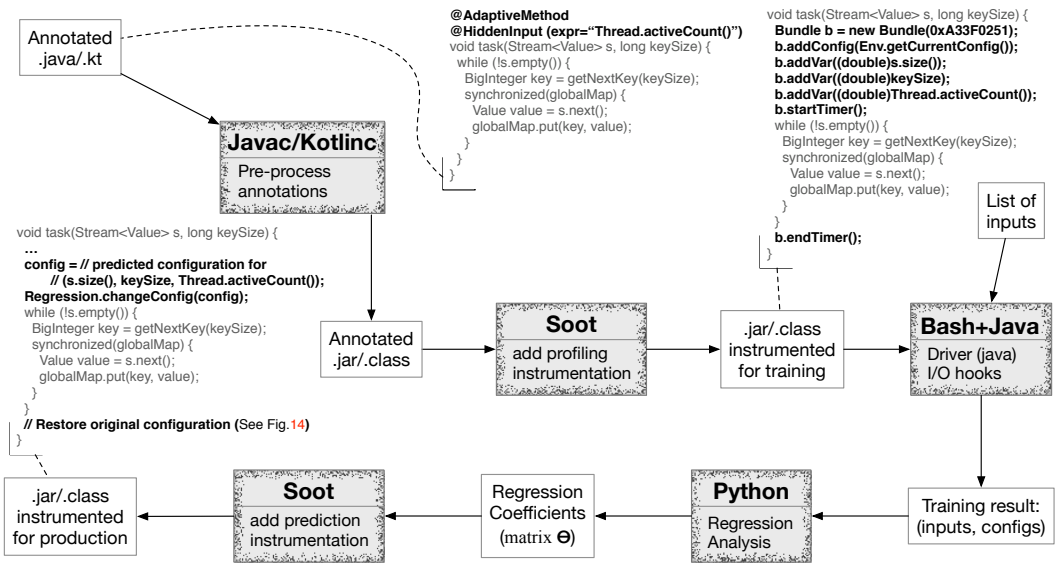


Fig. 4. The execution pipeline of Jinn-C.

$$
\begin{array}{l}
\\
\textbf{BestConfig}(f(\alpha_{01},\alpha_{02},\alpha_{03})) = \text{1b0L} \\
\textbf{BestConfig}(f(\alpha_{11},\alpha_{12},\alpha_{13})) = \text{0b1L} \\
\textbf{BestConfig}(f(\alpha_{21},\alpha_{22},\alpha_{23})) = \text{2b1L} \\
\textbf{BestConfig}(f(\alpha_{31},\alpha_{32},\alpha_{33})) = \text{2b1L}
\end{array}
\begin{array}{c}
\begin{array}{ccccc} \text{0b1L} & \text{1b0L} & \text{1b1L} & \text{2b0L} & \text{2b1L} \end{array} \\
\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
C
\end{array}
= \sigma \left(\begin{array}{c}
\overbrace{\begin{bmatrix} 1 & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ 1 & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ 1 & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ 1 & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix}}^{\text{function arguments}} \\
A
\end{array}
\times
\begin{array}{c}
\overbrace{\begin{bmatrix} \Theta_{00} & \Theta_{01} & \Theta_{02} & \Theta_{03} & \Theta_{04} \\ \Theta_{10} & \Theta_{11} & \Theta_{12} & \Theta_{13} & \Theta_{14} \\ \Theta_{20} & \Theta_{21} & \Theta_{22} & \Theta_{23} & \Theta_{24} \\ \Theta_{30} & \Theta_{31} & \Theta_{32} & \Theta_{33} & \Theta_{34} \end{bmatrix}}^{\text{training inputs}} \\
\Theta
\end{array}
\right)
$$

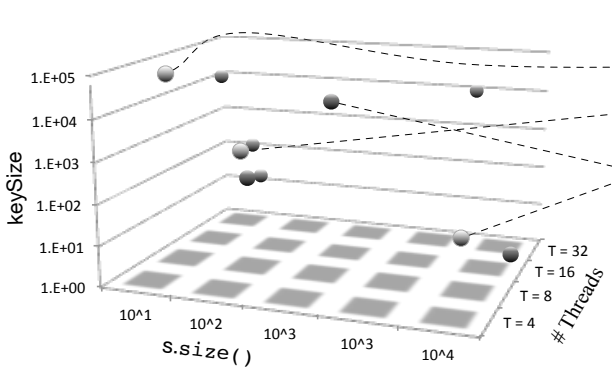
Fig. 5. Formula to train a 3-ary function $f(\alpha_0, \alpha_1, \alpha_2)$. The goal of multivariate linear regression is to find the coefficients Θ that approximate the product $C = \sigma(A\Theta)$. Training set contains four samples.

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 an $1 \times n$ 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 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 5 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 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. This all-ones column is the first column of matrix A in Figure 5.

Example 3.2. Figure 6 shows how ten different samples of function Task, from Fig. 1, are organized into a matrix A of independent variables.



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

matrix A with training inputs

Fig. 6. Training set for the Task method (Fig. 1). The table on the right is 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 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 7 reuses the ten samples earlier discussed in Example 3.2 to show how we build the matrix of dependent variables. Notice that this matrix has one line per sample, and one column per configuration of interest. Because a typical heterogeneous architecture might support thousands of different configurations, usually we separate a few when doing regression. For instance, in Section 4, to render our approach practical, we shall consider only 10 out of the 4,654 possible configurations of the Odroid XU4 board. This need for bounding the search space might, of course, 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. As previously mentioned, 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 most well-known of them (Cauchy 1847). 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

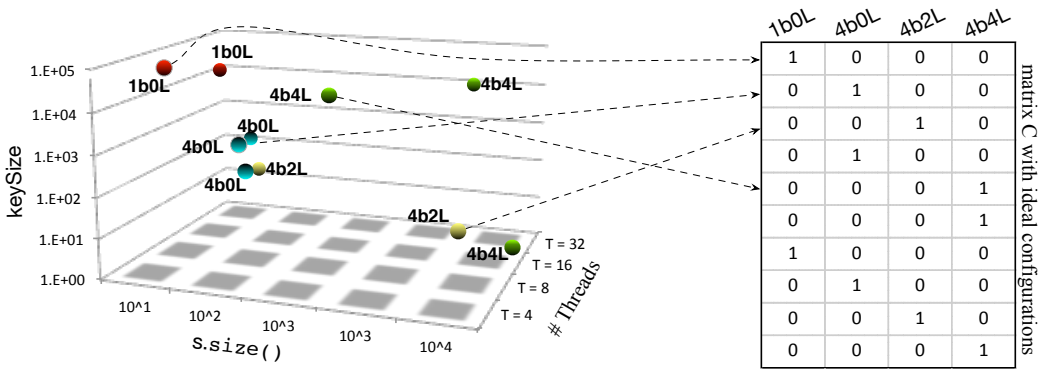


Fig. 7. Matrix of independent variables built for ten different invocations of function Task in Figure 1.

(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}. Henceforth we shall assume that Θ can be efficiently approximated for any training set. In Section 4.4 we shall demonstrate that such is the case.

Example 3.4. Figure 8 shows a possible matrix Θ that gradient descent finds for the Task function, when given the training set seen in Figures 6 and 7. 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[2] 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 8 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.

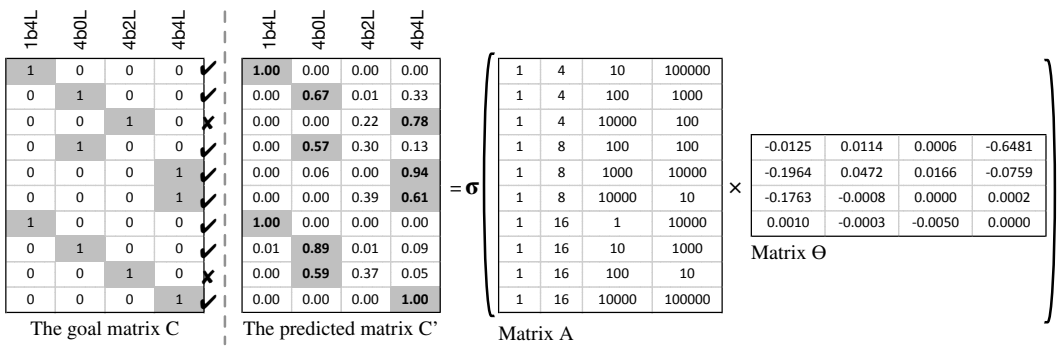


Fig. 8. The result of multivariate linear regression produced by the training set seen in Examples 3.2 and 3.3.

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

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

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 configuration that will be active during the invocation of f.

Example 3.5. Figure 9 shows how the matrix Θ found in Figure 8 supports prediction. We use it to guess the best configuration for four different input sets. These unseen invocations of Task are marked as the dark spheres in Figure 9. In this example, Θ lets us correctly predict the ideal configuration for three out of the four samples. In one case, the last input in Figure 9, we wrongly predict the best configuration as 4b2L, whereas empirical evidence suggests that it should be 4b4L.

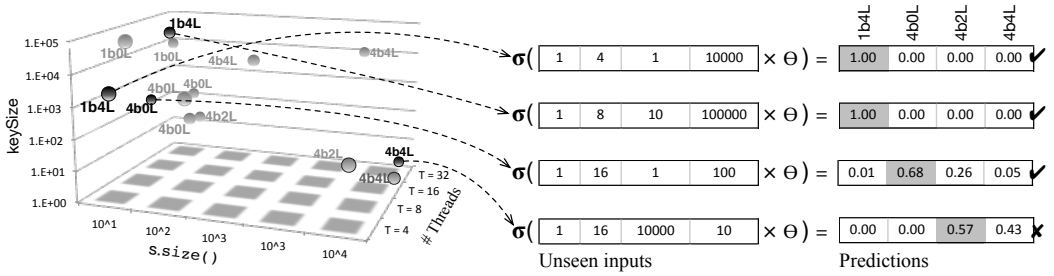


Fig. 9. The matrix Θ found in Figure 8 used to predict the ideal configuration for four unseen input sets. Inputs used in the training set are the light-grey points, whereas inputs in the test set are dark-grey.

3.2 Engineering the Training Phase

In the following subsections we describe our design decisions for the training phase.

3.2.1 Code Annotation. We use a system of annotations to tell Jinn-C what are the methods and their inputs that should be used in the multivariate regression. This can be used as either Java or Scala comments. We define three types of annotations:

@AdaptiveMethod: marks a method as the target of multivariate regression. The annotated method will go through every stage outlined in Figure 4. Unless the @Input annotation is also used, every formal parameter of the method will be used as an independent variable of the linear regression. Global variables are not considered inputs in this case.

@Input: specifies which references or primitive values are independent variables (the $\alpha's$ in Figure 5) in the regression. This annotation must be employed when Jinn-C's users know that some function arguments bear no effect onto the choice of ideal configurations for the target method. Function parameters and global variables (whose scope includes the point where the target method is declared) can be marked as inputs. If names marked as inputs are not visible within the target method, a compilation error ensues.

@HiddenInput: specifies extra information to be used as independent variables. These hidden inputs are mostly system variables, such as the number of threads; however, hidden inputs can also be global variables that are not directly used within a function, albeit they are accessed within methods called by said function. A method, chain of methods or any expression can be used to obtain a reference to a hidden input. The names used in these expressions must be visible during compilation time, otherwise an error is thrown.

Example 3.6. Figure 10 shows two examples of annotated methods. These examples were taken from actual applications. However, for the sake of readability, we have removed some boilerplate

code that, otherwise, would render the programs difficult to understand. The VISIT method, which is part of an implementation of the Breadth-First Search algorithm, contains three Input annotations. Two of them, referring to VISITED and GRAPH, were applied onto global variables. The other, on NT, refers to a method argument. The method COUNT, part of a sorting application, contains two Input annotations, all used on function arguments. These annotations are redundant in this example, because whenever an adaptive method does not present an Input annotation, all its arguments are marked as independent variables. Because this method is invoked by threads in a Java thread pool, the number of active threads in the pool is marked as a hidden input.

```
@AdaptiveMethod
@Input (global="visited")
@Input (global="graph")
@Input (param="NT")
void visit(final int NT) throws ... {
  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(); }
}
```

```
@AdaptiveMethod
@Input (param="START")
@Input (param="END")
@HiddenInput (expr="forkJoinPool.getActiveThreadCount()")
void count(final int START, final int END) {
  for (int j = START; j <= END; j++) {
    SingleCounter aux = counters[elements[j]];
    synchronized (aux) {
      aux.value += 1;
    }
  }
}
```

Fig. 10. Examples of annotated code snippets. (Left) Breadth-first search. (Right) Sorting application.

The expression (**expr="forkJoinPool.getActiveThreadCount()"**) will be parsed by SOOT, which will split it into 2 parts: forkJoinPool and getActiveThreadCount(). The former, forkJoinPool, must be an object accessible from the COUNT method, so forkJoinPool needs to be global in the current class or be a class visible in the path. Notice that our annotations can only be processed if we compile the original java (or Scala) file with debug information. For instance, if we use JAVAC to produce bytecodes, then we must pass the -g flag to it.

3.2.2 Extracting Sizes from Annotated Terms. Annotations tell JINN-C to build expressions denoting the size of the annotated names. The technique used to obtain these sizes depends on the type of the target input. Currently, we can reconstruct sizes for the following patterns:

- *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. Notice that, in this case, users can still use the **HiddenInput** annotation to specify an expression that yields the size of the target type.

Example 3.7. Figure 11 shows the instrumented version of the annotated programs discussed in Example 3.6. We remind the reader that such profiling interventions are inserted in the intermediate representation of these programs –source code is used only for readability. Instrumentation is performed by a singleton class 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. The identifier associates a method with a bundle. 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. 11. Instrumented version of programs seen in Figure 10. (Left) Breadth-first search. (Right) Sorting application.

3.2.3 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 *driver* –a service that runs the program that we want to optimize in a controlled environment. The driver has two responsibilities. First, it is in charge of warming up the target program. We call *warm-up* an execution of the target program performed before profiling starts. The warm-up phase tends to put the virtual machine into a steady state; thus, ensuring the consistency of the results that we produce during the training phase. Jinn-C's users must determine the number of warm-up rounds. Barrett *et al.* (Barrett et al. 2017) have shown that it is very difficult to ensure that a given virtual machine will always reach a steady state of peak performance. Nevertheless, in the experiments that we report in Section 4 using Java Hotspot, a steady state is reached. The second responsibility of the driver is to change hardware configurations before every profiling experiment takes place. To this end, the driver goes over a range of pre-defined configurations, repeating the same experiment a number of times for each of them.

```
void warmUp() {
  setWarmUp(true);
  for (int i = 0; i < WARM_UP_RUNS; i++) {
    // Use reflexion to call user code.
    // …
    runBench();
  }
  setWarmUp(false);
  // Next execution will be actual profiling…
}
```

```
static void setCoreConfig(Config config) throws ... {
  Runtime r = Runtime.getRuntime();
  // Build the command string for the system call:
  String configStr = configStr(config.numBig, config.nLITTLEs);
  final int pid = getProcessID();
  String cmd = "taskset -pa " + configStr + " " + pid;
  // Set the hardware configuration:
  Process p = r.exec(cmd);
  // Check for errors ...
}
```

Fig. 12. Example of functionalities provided by the driver. (Left) simplified version of the warm-up code. (Right) library code that changes the number of cores visible to the target program.

Figure 12 shows part of the driver's implementation. The code is organized as a framework: users must implement one method called runBench, which is then invoked a preset number of

times by the WarmUp function in Figure 12. Any implementation of runBench must invoke the target program once over a particular set of inputs. Users must specify the code that reads and loads inputs. Implementing runBench is one, out of the two tasks, that we expect from Jinn-C's users. The other task is to provide inputs for training. We use a suite of bash scripts to traverse and organize the program inputs, changing hardware configurations between experiments. Our framework provides functions to setup the hardware configuration. As an example, Figure 12 shows function setCoreConfig, which determines the number of big and LITTLE cores available on the Odroid XU4 board that we use in this paper.

Jinn-C receives an annotated program P, a set of different inputs $I = \{\iota_1, \iota_2, \ldots, \iota_m\}$ of P, a set of acceptable hardware configurations $H = \{h_1, h_2, \ldots, h_n\}$, and the implementation of the runBench method. It will then invoke runBench 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, according to some objective function, such as time or energy consumption. In case of ties, we choose the configuration with the smallest quantity of 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.

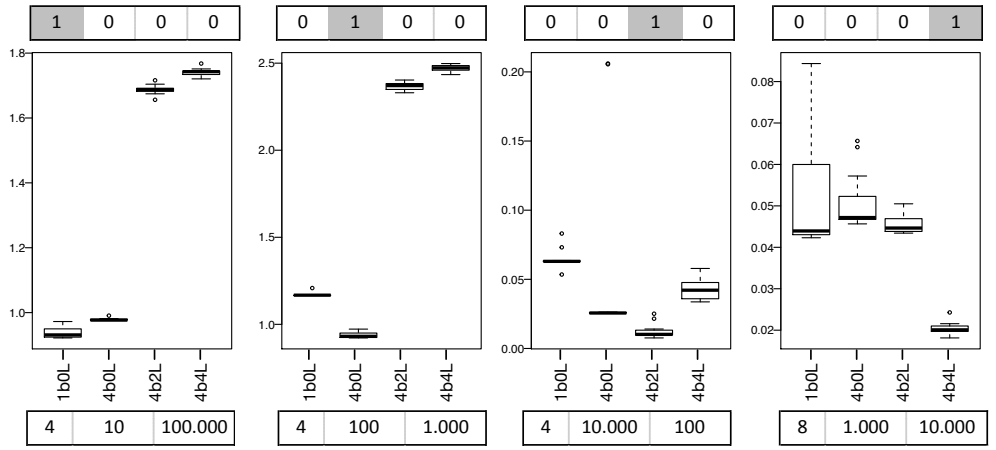


Fig. 13. Training output produced by the driver on a few inputs seen in Figure 6. Y-axis is runtime in seconds.

Example 3.8. Figure 13 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 13 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 13 are the best configurations. These vectors will give us the dependent variables used in the regression.

3.3 Generation of Adaptive Code

The product of training is a collection of floating-point constraints, organized into a matrix Θ. These constraints are 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 an $1 \times n$ vector. The size of A_i is one plus the number of inputs of the target function. The expression $\sigma(A_i\Theta)$ yields an $1 \times k$

vector of probabilities, whose sum adds 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.

Example 3.9. Figure 14 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.

```
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()) {
    // 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);
    }
  }
  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. 14. The production version of function Task (Fig.1).

4 EVALUATION

The goal of this section is to demonstrate the effectiveness of the technique presented in this work 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 – Speed:** what is the speedup that can be obtained by Jinn-C when compared to scheduling techniques of similar goals?
- **RQ2 – Energy:** what is the improvement that Jinn-C delivers on top of other tools, in terms of energy consumption?
- **RQ3 – Training:** what is the training time of Jinn-C, and how does it compare to the training time of similar tools?
- **RQ4 – Convexity:** how is the space of best configurations that Jinn-C explores when trying to optimize programs?

We compare Jinn-C with two state-of-the-art approaches: Sreelatha et al.'s CHOAMP, and ARM's GTS (Jeff 2013). GTS, short for *Global Task Scheduling*, is the default scheduler for big.LITTLE systems running the Linux Kernel. Before delving into numbers, in Section 4.1 we introduce the runtime environment we have used to carry out the evaluation of Jinn-C.

4.1 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. To measure the energy consumed exclusively by specific functions, we send signals to the synchronization circuit seen in Figure 3-a through one of the board's GPIO pin. We use the energy measurement framework proposed by Bessa et al.. 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[3] and Soot 3.2.0[4] to analyze, instrument and run bytecodes. No modifications have been made in the Java Virtual Machine or its Just-in-Time compilers –all the interventions 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 its actual execution. The exact number of warm-up runs is specific for each benchmark and was manually tuned for each one of them (details in Table 1). Tuning is made possible by the JVM flag `-XX:+PrintCompilation`, which allows us to see when JIT compilation kicks in during the execution of an application. Thus, we can change the number of warm-up rounds, to minimize the amount of compilation taking place during the final –metered– run of a given benchmark. 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 by CHOAMP during its training stage (details in section 4.3). 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. (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. was implemented in C/C++, so we had to reimplement all the benchmarks in Java. We had to remove 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. We have replaced it with BFS, which is also part of PBBS, but was not in Acar et al.'s suite.

We also chose six benchmarks from the *Renaissance* benchmark collection, which was recently released by 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 and verification code. The extra inputs enable profiling, and the verification code is necessary to check execution correctness. The six benchmarks that we chose are implemented in Scala; however, they rely on a variety of Java libraries, such as Twitter's Finagle (Twitter 2019), Java Jenetics (Wilhelmsttter 2019), the Spark Machine Learning Library (Meng et al. 2016), and the standard Java library. Our criterion when picking up programs was simplicity: we selected benchmarks that were easy to extend with more inputs. 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's book. We shall refer to these four programs as part of Jinn-C's *test suite*. All the 18 benchmarks used in this paper share a similar running environment:

[3]https://jdk.java.net/11/

[4]https://github.com/Sable/soot/releases/tag/3.2.0

Source	*Benchmark*	*TTime*	*Lang.*	*LoC*	*W*	*Class*
Shun et al.	bfs	42m33s	J	353	4	graph manipulation
Shun et al.	radixSort	20m51s	J	501	4	sorting algorithm
Shun et al.	sampleSort	26m17s	J	414	3	sorting algorithm
Shun et al.	suffixArray	30m12s	J	316	3	string manipulation
Shun et al.	removeDuplicates	30m31s	J	174	4	sequence manipulation
Shun et al.	convexHull	56m30s	J	499	5	geometry and graphics
Shun et al.	nearestNeighbors	30m29s	J	715	3	geometry and graphics
Shun et al.	spanningForest	21m40s	J	410	4	graph manipulation
Prokopec et al.	als	80m12s	S/J	97	1	matrix factorization
Prokopec et al.	philosophers	21m15s	S/J	146	1	synchronization algorithm
Prokopec et al.	futureGenetic	26m8s	S/J	115	1	genetic algorithm
Prokopec et al.	finagleHTTP	225m10s	S/J	119	1	server-client exchanges
Prokopec et al.	chiSquare	27m15s	S/J	101	1	statistical algorithm
Prokopec et al.	decTree	64m22s	S/J	129	1	random forest algorithm
Jinn-C	collinearPoints	32m1	J	565	3	geometry and graphics
Jinn-C	hashSync	94m7s	J	73	3	sequence manipulation
Jinn-C	insertAndAdd	47m30s	J	130	4	database manipulation
Jinn-C	randomNumComp	26m7s	J	89	6	system exploration

Table 1. Benchmarks used for evaluating Jinn-C. The *TTime* column shows the time required to train each benchmark, which will be further explained in Section 4.4. *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. among Jinn-C's benchmarks, 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 predicted; and InsertAndAdd implements parallel operations on a DataBase.

an execution driver that is responsible for warming them up, preparing the inputs and collecting time and energy values. The time and energy used by the driver itself is never considered in our experiments. Table 1 presents an overview of the used benchmarks, as well as basic characteristic 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 4.5 and 4.6 further discuss the impact of different inputs in the execution time and energy consumption of the applications.

4.2 On the Choice of Hardware Configurations

When training Jinn-C and CHOAMP, we consider a universe of six core configurations: 4b4L (4 big and 4 LITTLE cores), 4b0L, 0b4L, 2b2L, 2b0L and 0b2L. The LITTLE cores run always at maximum frequency: 1.5GHz; for the big cores, we let them run at either 1.6GHz or 1.8GHz. 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. GTS runs on 4b4L by default, meaning it is allowed to choose among any possible hardware configuration involving big and LITTLE cores. We coupled GTS with the *on-demand* frequency governor, meaning that the runtime system is free to choose any frequency level 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.

Before presenting results, one last observation is in order: we chose 1.6GHz and 1.8GHz, instead of the highest frequency levels (1.9 and 2.0GHz), for the big cores to better deal with *thermal throttling*. Thermal throttling forces the Operating System to downscale CPUs' frequencies (Cohen et al. 2003). As stated by Mishra et al., this is a security feature that keeps the system temperature under a safe threshold. Excessive exposures to high temperatures could damage the equipment.

We have noticed empirically that thermal throttling renders experiments at 1.9 or 2.0GHz hard to reproduce. Figure 15 (a) illustrates this issue on the Odroid Xu4 board. The image displays the online values for temperature and clock frequency when executing a parallel application that performs math calculations during 15 seconds. The benchmark uses all 8 available cores and every time the temperature surpasses 176 F (80 C) the clock speed is decreased, leading to thermal values under the acceptable threshold. Such behavior happens even when DVFS is disabled. This experiment can be easily reproduced with the code in Figure 15 (b).

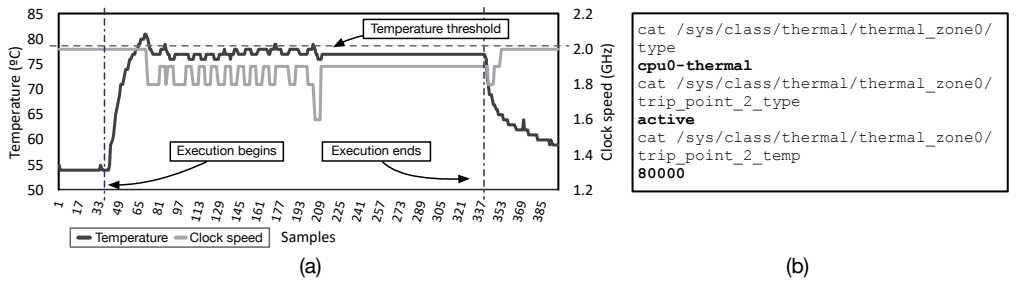


Fig. 15. Variation in CPU frequency and temperature values for the big cluster while running a sample application that uses all 8 available cores. Samples collected at each 50 ms from thermal sensors present in the Odroid Xu4 board. The code in the right side shows where such values are set in the Operating System.

4.3 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*, 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 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 scenario.

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 `incrementAnd`

Prime Feature	*Language dependent*	*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 blocks/methods
False Sharing	No	-	-
Flush operations	Yes	omp flush	not used

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

`Get()` is invoked. However, some features like *flush operations*, proposed by Sreelatha et al., were not reused in our implementation, due to a lack of correspondents in Java.

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 original generator scripts[5], 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 ended 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..

4.4 RQ1: Training time

Both techniques, Jinn-C and CHOAMP, require training. Training adjusts the parameters of the regression models to enable predictions of good hardware configurations. 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 previously described in Section 4.2. In our hardware, we took about 780 minutes to train our implementation of CHOAMP. Out of this training time, 365 minutes were spent running the micro-benchmarks with the 1.8GHz CPU frequency for the big cluster. When using the frequency of 1.6GHz, the time required for training was 415 minutes.

To train Jinn-C, we follow the methodology described in Section 3.2.3: we run the target application on the allowed hardware configurations using the inputs available for training. Jinn-C's training time, naturally, depends on the target application's run time, and on the number of available inputs. Table 1 shows the training time of each individual benchmark. Using ten inputs and ten allowed hardware states (clock speed × hardware configurations) per benchmark, we took around 903 minutes to train the 18 programs used in this section. The longest time, three hours and 45 minutes were 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, and, as we will see in Section 4.5, runtime overhead tends to be minimal. This overhead is due to the matrix multiplication

[5]https://bitbucket. org/jkrishnavs/openmp-eigenbench

that happens once a hot function is invoked, as we have discussed in Section 3.3. The product of training, the code earlier seen in Figure 14, is embedded directly into a program's bytecode. Thus, different programs adapted by JINN-C can coexist independently in the same runtime environment, for no changes are required in the operating system as a result of training.

4.5 RQ2: Optimizing for Speed

We have tested JINN-C and CHOAMP with two objective functions: speed and energy consumption. When the cost function is speed, the tools try to decrease the execution time of target applications. Figure 16 reports results observed when optimizing for speed. In Section 4.6 we discuss energy consumption. We have tested each benchmark with four input sets. Each chart within those figures shows four sets of three boxplots. Boxplots refer, in this order, to JINN-C, CHOAMP and GTS. We adopt a significance level $\alpha = 0.05$; i.e., a confidence interval 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. In practical terms, we use Student's Test to measure the p-value of two populations, and consider significant results with a p-value lower than 0.05. 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. The grey box x means that the two winning systems have produced results very similar (with a p-value greater than 0.05). Above each one of the four input sets used in each benchmark, we show the configuration that JINN-C chose for that input. We also show, in a grey box, to the right of the name of each benchmark, the configuration that CHOAMP chooses for that benchmark.

The data in Figure 16 shows 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 results are summarized in Figure 17.

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 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 par with the other schedulers. On the other hand, for some benchmark, such as CHISQUARE or FUTUREGENETIC, JINN-C's choices outperformed other scheduling techiniques, with rather different configurations for different inputs, as it is expected for the tool. In the CHISQUARE case, for example, with the 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's 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 that JINN-C's predicted configuration (2b0L) led to a mean runtime of 17.00 seconds, while CHOAMP's had a runtime of 18.70 and GTS 17.76. For the second input, all the p-values were below 0.0005, resulting in a confidence interval over 99%. These scenarios illustrate well the effectiveness of JINN-C in identifying the most suitable configuration for applications that behave differently according to the inputs fed to them.

4.6 RQ3: Energy Consumption

Figure 18 compares CHOAMP, GTS and JINN-C regarding energy consumption. When set up to

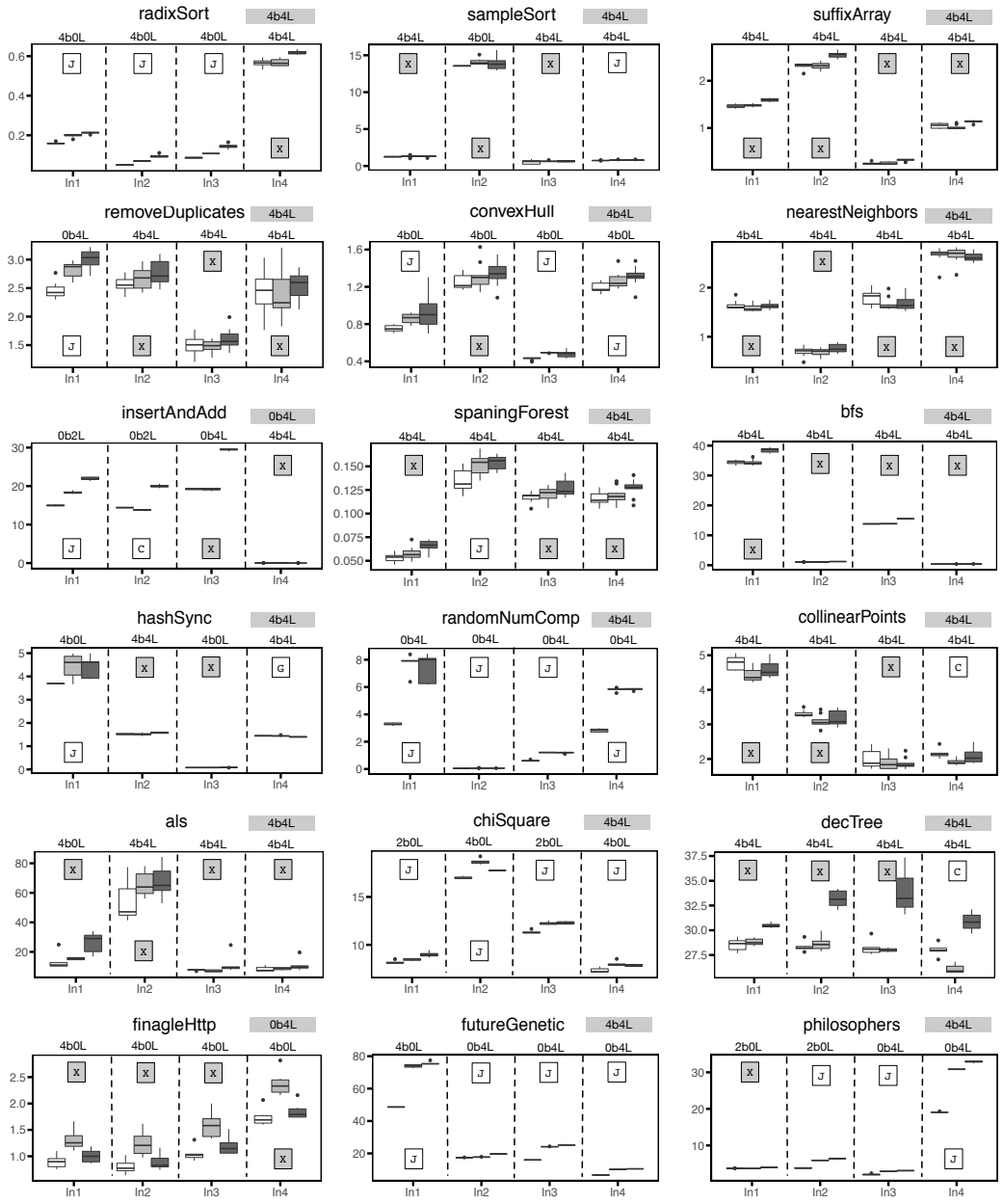


Fig. 16. Execution time of benchmarks from Table 1. Y-axis shows time in seconds. X-axis shows different experiments; each experiment uses different inputs. Boxplots are ordered by Jinn-C, CHOAMP and GTS.

reduce energy consumption, Jinn-C and CHOAMP build models to estimate the most adequate hardware configuration to save energy. 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.

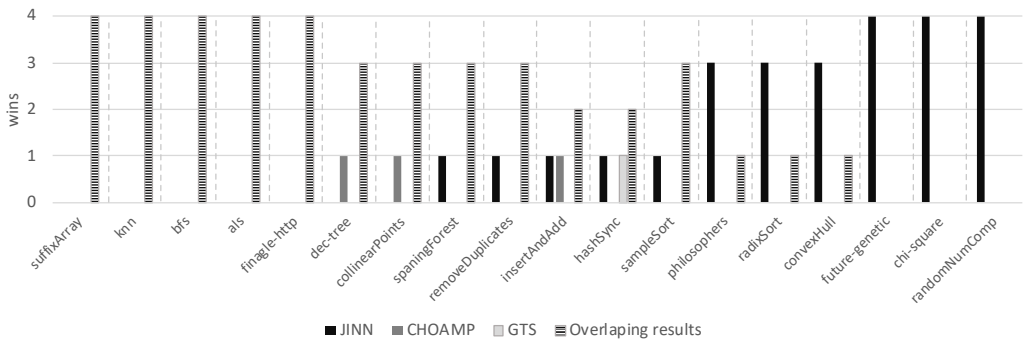


Fig. 17. Summary of the results displayed in Figure 16

When optimizing for speed and energy, GTS with the *on-demand* DVFS governor was free to choose any possible configuration with frequency levels ranging from 200MHz to 1.8GHz in the big cluster and from 200MHz to 1.5GHz in the little one. As this is the default and expected behavior for the GTS scheduling policy, we kept it as is. Observe that this scheduling technique may lead to performance degradation because GTS increases frequency gradually, until it arrives at the top levels in computation intensive programs. Additionally, even with several warm-up rounds, GTS might take an excessively long time to achieve maximum frequency levels for some applications.

Figure 18 reveals that Jinn-C achieved best results in 20 experiments (out of 72); GTS was the best approach in 2, and CHOAMP in 6. Figure 19 summarizes these results. Most of the experiments did not have a clear winner –this difficulty to pinpoint a best technique is, in part, due to the fact that we measure energy for the entire board, not only for the cores. Therefore, peripherals such as the fan and the memory bus increase the variance of our results.

We have observed that Jinn-C outperforms GTS mostly due to its ability to choose high-performance hardware configurations, such as 4b4L at 1.6GHz immediately, whereas GTS needs a warm-up period to arrive at them. Our implementation of CHOAMP has chosen the 0b4L configuration at 1.6GHz for almost all the samples in this evaluation. We speculate that this behavior happens because some features, such as branching and memory operations, tend to dominate 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.

On the Influence of Execution History. The execution history impacts the energy consumed by different programs. Take as an example the entry corresponding to HashSync in Figure 18. When analyzing the second input set (In2), we observed that, although predicting the same configuration as CHOAMP, Jinn-C led to marginally higher energy consumption. This behavior is even more surprising once we consider that Jinn-C's and CHOAMP's codes run in about the same time, as Figure 16 reveals. The culprit of this apparently 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 codes 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.

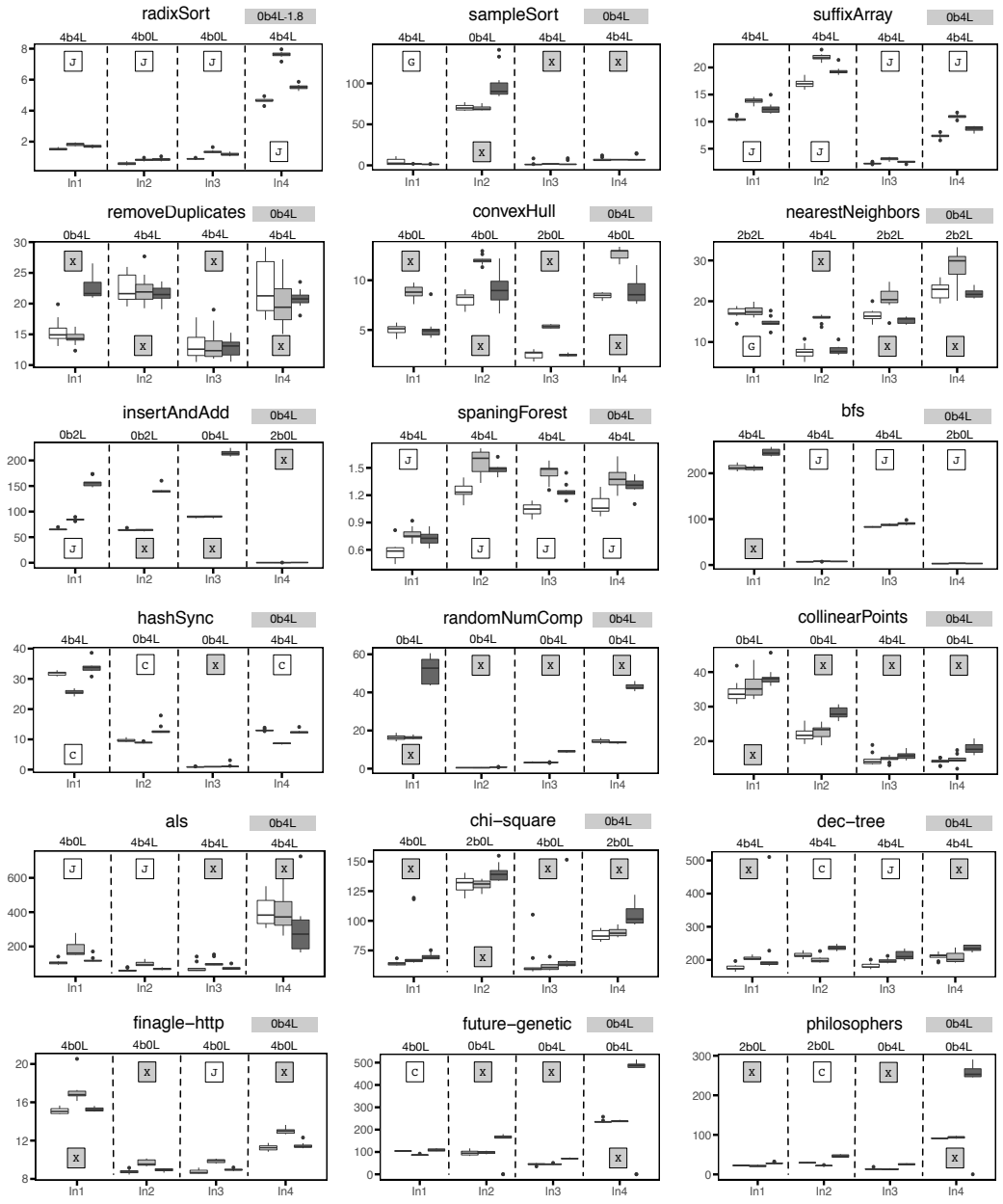


Fig. 18. Energy consumed by the benchmarks in Table 1. Y-axis shows energy in Joules. X-axis shows different experiments. Boxplots are sorted as in Figure 16.

Figure 20 shows the power profile of HashSync, including warm-up and measurement phases. The invocations of HashSync in the warm-up stage have different set of inputs compared to its invocation in the measurement stage. As a result, our technique predicted the configuration

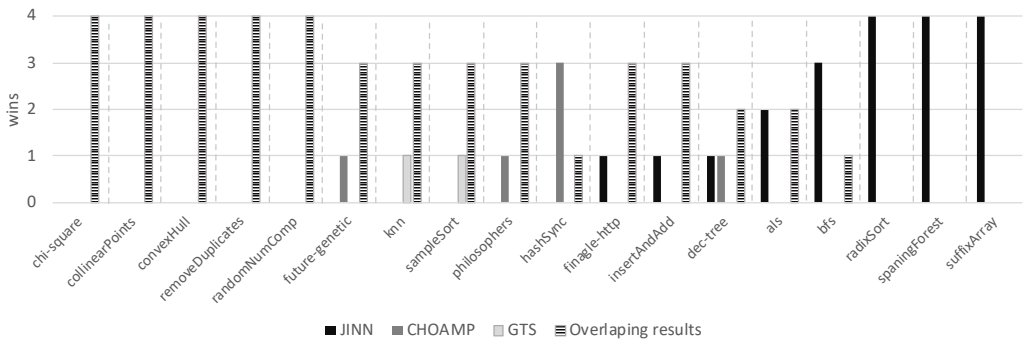


Fig. 19. Summary of the results displayed in Figure 18

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 the hysteresis of power dissipation. This inertia is a well-known phenomenon (De Leon and Semlyen 1995; Piquette et al. 2002); it may be seen as the tendency of a system to conserve an electrical deformation caused by a stimulus. In the context of this example, such stimulus is the use of the big cluster in the warm-up phase.

The mean power dissipated by Jinn-C's version of HashSync in Figure 20(a) was 4.68W. CHOAMP's mean power is 4.09W, as taken from Figure 20(b). Thus, 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 we observe that the average power dissipation goes down to 3.95W. Figure 20(c) reports the power profile of this setup. The only difference between the two executions of Jinn-C, in Figures 20 (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 once we ensure that both use the same hardware configuration during warm-up.

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 runtime (Fig. 16). The need to change configuration when moving from warm-up to measurement has costed Jinn-C's code some time. Although a small fraction of the overall execution, it let CHOAMP's program run slightly faster than Jinn-C's. When reporting the results in this paper (Figs. 16 and 18), 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.

4.7 RQ4: Convexity

A *convex space* is a region within an 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 optimization problems, because exploration methods based on derivatives, such as gradient descent, are guaranteed to converge to the optimal solution when applied on them (Boyd and Vandenberghe 2004). Therefore, we can restrict ourselves to them, as they are known to be much faster than other space exploration techniques, such as those that use quadratic or higher-order polynomials (e.g., multi-layer perceptrons). That is the reason why we chose a linear model to match hardware configurations with program inputs.

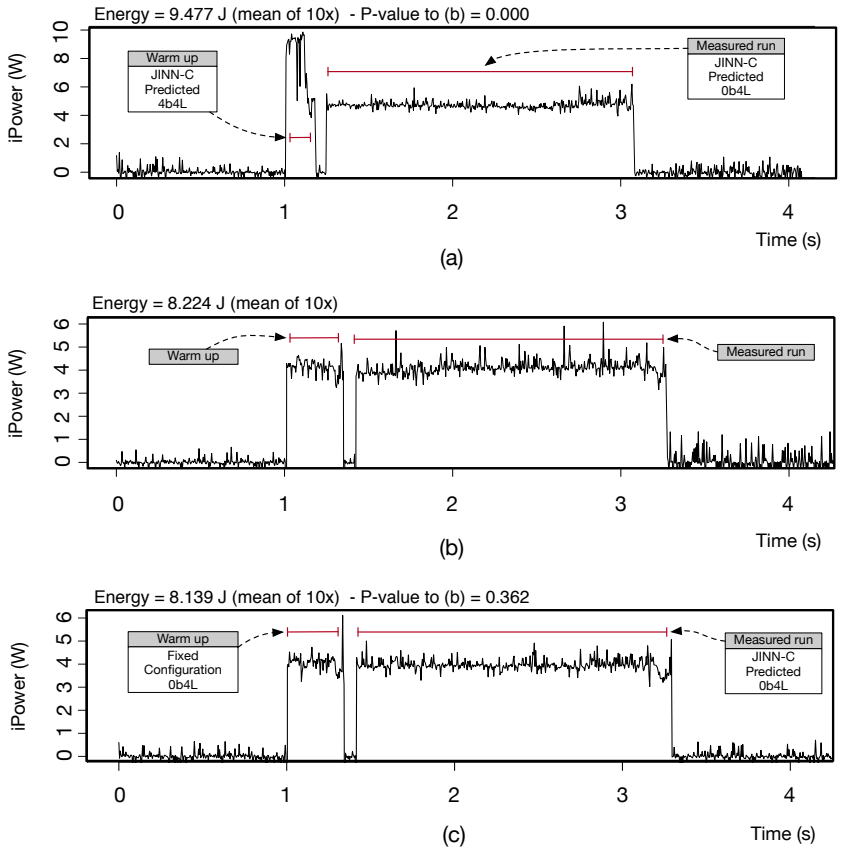


Fig. 20. Power consumption of HashSync with (a) JINN-C, (b) CHOAMP, and (c) JINN-C with fixed configuration during warm-up. P-values below 0.05 indicate that the executions of JINN-C's and CHOAMP's code are statistically different. For this benchmark, CHOAMP (b) predicted 0b4L as the best configuration for the parallel kernel. This configuration is used in all warm-up stages and in the measurement phase. Figures 16 and 18 report values for the measured run only.

In our setting, the search space is a function that maps program inputs to optimal hardware configurations. This function is discrete, because its image is a finite set of hardware configurations. Convexity, in this case, means that if we fix all the program inputs, and vary the one left, every 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". In this section, we analyze the space of optimal hardware configurations, to provide some evidence that these regions tends to be convex in practice. Notice that convexity is a tendency, not a principle. In other words, it is possible to implement programs whose space of optimal configurations is not convex. Example 4.1 shows an instance of such a program.

Example 4.1. If we build a function that associates the input `i` of the procedure `unlikely` (seen below) with optimal hardware configurations, then we obtain a non-convex (concave) space:

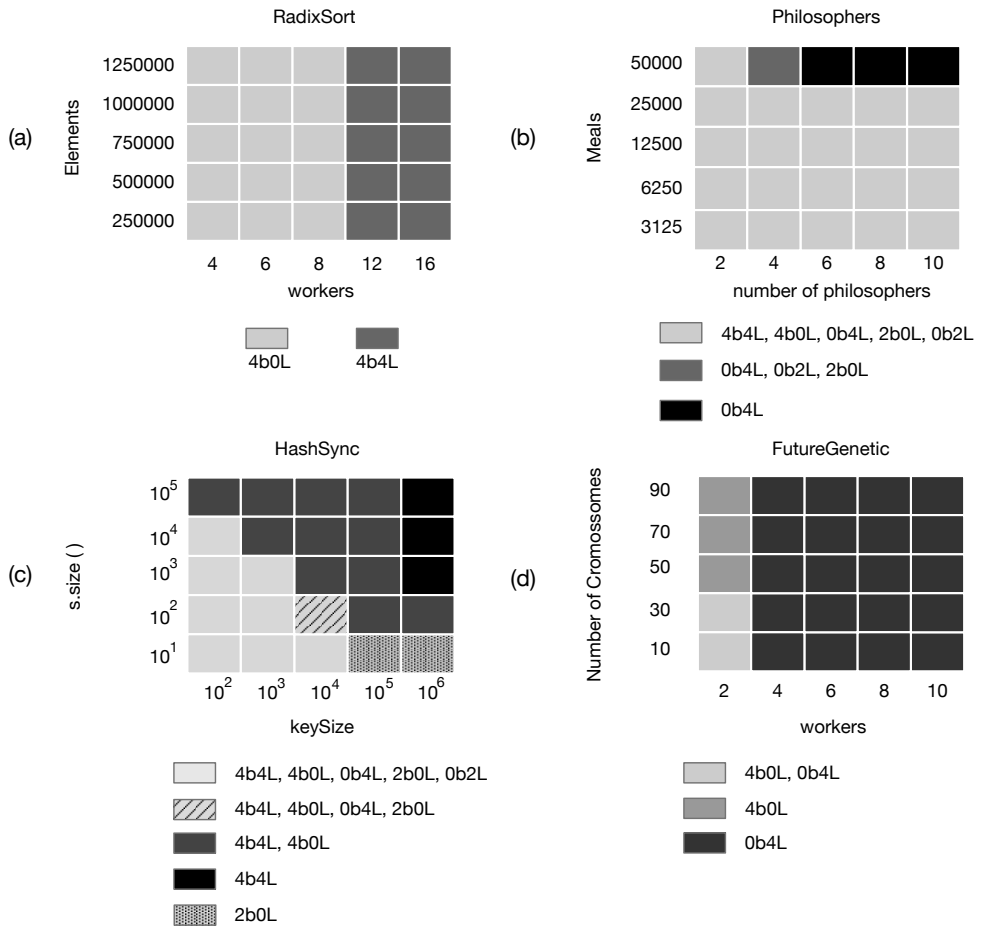


Fig. 21. Best configurations for 4 benchmarks used in our evaluation. The charts exemplify the convex space over benchmarks inputs. HashSync and FutureGenetic receive 3 inputs each, but for this experiment we fixed the number of workers in HashSync to 16 and the number of generations in FutureGenetic to 5000.

```
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. Let configuration h_c be the optimal hardware configuration for procedure `comp_intensive`. The space occupied by h_c, i.e., $[-\infty, 10[\cup], 100, +\infty]$ is non-continuous; hence, concave.

The program discussed in Example 4.1 is unlikely to exist in real-world code. To support this statement, Figure 21 provides a glimpse of the best hardware configurations for different inputs of four benchmarks in our collection. The figure contains four parts. Each part is a table, which associates a pair of inputs with the hardware configurations that yielded the fastest 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 only. However, to avoid having to draw a 3D-figure, we have fixed one input for each benchmark[6]: `number_of_Generations` for FutureGenetic, and `number_of_workers`, for HashSync. The choice of benchmarks is arbitrary. We did not add more benchmarks to this experiment because the generation of the data necessary to build each table demands considerable computational time, e.g.:

- RadixSort: 1 hour and 37 minutes
- Philosophers: 2 hours and 24 minutes
- FutureGenetic: 55 hours and 53 minutes
- HashSync: 58 hours and 12 minutes

Furthermore, to keep evaluation within a reasonable time frame, we have used only the frequency level of 1.8GHz for the big cores. Had we also included the level of 1.6GHz, as in the previous sections, then our total running time would more than double.

Experimental Setup: The four tables in Figure 21 show convex spaces: any sequence of rows or columns traverses a continuous region. Example 4.2 illustrates what we mean by a continuous region.

Example 4.2. Consider HashSync in Figure 21(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.

However, to arrive at this result, we had to account for small variations in runtime. To generate the data in every table seen in Figure 21, we considered 5×5 combinations of inputs, and the five 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. Thus, to deal with variance, we had to resort to more sophisticated statistical tools, as we explain in the rest of this section.

Dealing with variance: Had we simply picked for every input pair the configuration with the best average runtime, then variations would lead to almost random results for small inputs. To avoid this problem, we consider not the best, but the set of best hardware configurations per input. For each input, we fitted our linear regression model (via *Python's statsmodels* module (Seabold and Perktold 2010)) using the ordinary least squares method to estimate the model parameters. Then, we analyze the differences among group means with standard analysis of variance (ANOVA) (Fisher 1918). ANOVA generalizes 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 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 existing pair of configurations. As a result, the significance level had to be 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

[6]Notice that varying all the three inputs would also increase substantially the time to run this experiment. We speculate that this time would jump from 58 hours up to 12 days for HashSync only.

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. Otherwise, the two groups are considered identical and become part of the same cluster of configurations. The runtime of a cluster of configurations is the average of all the samples in that cluster.

5 RELATED WORK

This paper uses a type of machine learning technique –multivariate linear regression– to solve an instance of program scheduling in heterogeneous architectures. Machine learning and scheduling in heterogeneous systems have played an important role in compiler design in recent years. For an overview of the impact of machine learning onto compiler construction, we recommend surveys from Wang and O'Boyle and Ashouri et al. The rest of this section focuses on scheduling.

5.1 A General Overview on Program Scheduling in Heterogeneous Systems

The general problem of scheduling computations in heterogeneous architectures has attracted much attention in recent years, as Mittal and Vetter have thoroughly discussed. 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 questions:

- **Architecture**: do they apply to Single or Multi-ISA systems?
- **Source**: is the program's code modified?
- **Input**: is the approach input-aware?
- **Auto**: is user intervention required to choose a 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. These 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 scheduling decisions are made before the program runs, no dynamic checks are necessary to schedule computations. However, these techniques 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). Examples of runtime information include 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 does not spend much time weighing choices. Thus, the scheduler might take suboptimal decisions due to its inability to solve hard combinatorial problems.

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
(Nishtala et al. 2017)	O	Single	No	No	Yes	Yes	Yes
(Petrucci et al. 2015)	O	Single	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).

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* have used a compiler to instrument a program with guards that determine, based on input sizes, where each loop should run. Cong and Yuan, 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.* 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 in 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 Single-ISA systems, as Mittal 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 (Cai et al. 2016; Garcia-Garcia et al. 2018; Mittal 2016; Park et al. 2018; Van Craeynest et al. 2012b). This section describes works that, like Jinn-C, are adaptive, and have been specifically designed for big.LITTLE architectures. Table 4 categorizes these techniques along the following lines:

- **Granularity**: what is the data used for training? Most of the techniques use the system's workload –available through performance counters. Choamp relies on features mined from the target's program code. We use the program's inputs to perform predictions.
- **Training**: when does learning occur? Off-line systems calibrate the prediction model before the target program runs; on-line systems do it while the program executes.

	Approach	*Granularity*	*Training*	*Data*	*Target*	*Level*
Octopus-Man (Petrucci et al. 2015)	runtime	on-line	self	server	OS
Sparta (Donyanavard et al. 2016)	runtime	off-line	μ-bench	client	OS
DyPO (Gupta et al. 2017)	runtime	off-line	μ-bench	client	OS
Tzilis et al. (2019)	runtime	off-line	μ-bench	client	OS
Hispter (Nishtala et al. 2017)	runtime	off/on-line	μ-bench+self	server	OS
Choamp (Sreelatha et al. 2018)	syntax	off-line	μ-bench	client	Comp.
SIAM (Krishna and Nasre 2018)	syntax+data	off-line	self	client	Comp.
Jinn-C	data	off-line	self	client	Comp.

Table 4. Different solutions to Isha published in recent years.

- **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, ours included, present solutions for embedded devices and smartphones. Octopus-Man and Hipster were designed for data-centers.
- **Level**: as seen in Table 3. The different adaptive techniques that we list in Table 4 either run on the operating system (OS), or are implemented in the compiler.

The two related works that implement scheduling of computations in big.LITTLE architectures at the compiler level are Sreelatha et al.'s CHOAMP, and Krishna and Nasre'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 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 SpaningForest 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.

6 CONCLUSION

This paper has presented a code generation technique that adapts programs to good hardware configurations, in the context of a single-ISA heterogeneous system. 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. To capitalize onto 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 speed, the default Linux scheduler (the Global Task Scheduler), and CHOAMP, a state-of-the-art tool that predicts the best hardware configuration to a program based on its syntax (and implied semantics). The intuition nurtured during the craft of our tool, Jinn-C, lets us believe that our technique –linear regression on function inputs– can be applied onto different programming languages and runtime environments. The realization of such intuition on concrete technologies is an interesting research direction that we still would like to explore in the future.

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.

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.

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.

Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 52:1–52:27. DOI: http://dx.doi.org/10.1145/3133876

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.

Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. 2012. The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software. In *ISCA*. IEEE, Washington, DC, USA, 225–236.

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.

A. Cohen, F. Finkelstein, A. Mendelson, R. Ronen, and D. Rudoy. 2003. On Estimating Optimal Performance of CPU Dynamic Thermal Management. *IEEE Computer Architecture Letters* 2, 1 (Jan 2003), 6–6. DOI: http://dx.doi.org/10.1109/L-CA.2003.5

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.

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.

Francisco De Leon and Adam Semlyen. 1995. A simple representation of dynamic hysteresis losses in power transformers. *IEEE Transactions on Power Delivery* 10, 1 (1995), 315–321.

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

Michael Ditty, Tegra Architecture, John Montrym, and Craig M. Wittenbrink. 2014. NVIDIA's Tegra K1 system-on-chip. In *HCS*. IEEE, Los Alamitos, CA, USA, 1–26.

Alastair F. Donaldson, Paul Keir, and Anton Lokhmotov. 2008. Compile-Time and Run-Time Issues in an Auto-Parallelisation System for the Cell BE Processor. In *Euro-Par Workshops*. Springer, Berlin, Germany, 163–173.

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.

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

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

Michael Garland and David B. Kirk. 2010. Understanding throughput-oriented architectures. *Commun. ACM* 53 (2010), 58–66. Issue 11.

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

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

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.

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.

Adam Jundt, Allyson Cauble-Chantrenne, Ananta Tiwari, Joshua Peraza, Michael A. Laurenzano, and Laura Carrington. 2015. Compute Bottlenecks on the New 64-bit ARM. In *E2SC*. ACM, New York, NY, USA, 6:1–6:7.

Melanie Kambadur and Martha A. Kim. 2014. An experimental survey of energy management across the stack. In *OOPSLA*. ACM, New York, NY, USA, 329–344.

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, Norman P. Jouppi, and Parthasarathy Ranganathan. 2005. Heterogeneous Chip Multiprocessors. *Computer* 38, 11 (2005), 32–38.

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.

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.

Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, and others. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning Control for Predictable Latency and Low Energy. In *ASPLOS*. ACM, New York, NY, USA, 184–198.

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.

John Nickolls and William J. Dally. 2010. The GPU Computing Era. *IEEE Micro* 30 (2010), 56–69.

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.

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.

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.

Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Understanding Energy Behaviors of Thread Management Constructs. In *OOPSLA*. ACM, New York, NY, USA, 345–360.

Jean C Piquette, Elizabeth A McLaughlin, Wei Ren, and Binu K Mukherjee. 2002. Generalization of a model of hysteresis for dynamical systems. *The Journal of the Acoustical Society of America* 111, 6 (2002), 2671–2674.

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.

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.

Skipper Seabold and Josef Perktold. 2010. Statsmodels: Econometric and statistical modeling with python. In *SciPy*, Vol. 57. SciPy.org, Austin, Texas, USA, 61.

Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, and Michael L. Scott. 2002. Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *HPCA*. IEEE, Washington, DC, USA, 29–.

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.

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.

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.

Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson. 2018. GPU Schedulers: How Fair Is Fair Enough?. In *CONCUR*. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, 23:1–23:17.

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.

Twitter. 2019. Open-Source Twitter Finagle Repository at GitHub. https://github.com/twitter/finagle. (2019).

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

Franz Wilhelmsttter. 2019. Open-Source Java Jenetics Repository at GitHub. https://github.com/jenetics/jenetics. (2019).

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.