PYTHIA: an oracle to guide runtime system decisions

Alexis Colin, François Trahay, Denis Conan

HAL Id: hal-03750441
https://hal.science/hal-03750441v1

Submitted on 12 Aug 2022

PYTHIA: an oracle to guide runtime system decisions

Alexis Colin
Télécom SudParis
Institut Polytechnique de Paris
Palaiseau, France
alexis.colin@telecom-sudparis.eu

Francois Trahay
Télécom SudParis
Institut Polytechnique de Paris
Évry, France
francois.trahay@telecom-sudparis.eu

Denis Conan
Télécom SudParis
Institut Polytechnique de Paris
Évry, France
denis.conan@telecom-sudparis.eu

Abstract—Runtime systems are commonly used by parallel applications in order to efficiently exploit the underlying hardware resources. A runtime system hides the complexity of the management of the hardware and exposes a high-level interface to application developers. To this end, it makes decisions by relying on heuristics that estimate the future behavior of the application. In this paper, we propose PYTHIA, a library that serves as an oracle capable of predicting the future behavior of an application, so that the runtime system can make more informed decisions. PYTHIA builds on the deterministic nature of many HPC applications: by recording an execution trace, PYTHIA captures the application main behavior. The trace can be provided for future executions of the application, and a runtime system can ask for predictions of future program behavior. We evaluate PYTHIA on 13 MPI applications and show that PYTHIA can accurately predict the future of most of these applications, even when varying the problem size. We demonstrate how PYTHIA predictions can guide a runtime system optimization by implementing an adaptive thread parallelism strategy in GNU OpenMP runtime system. The evaluation shows that, thanks to PYTHIA prediction, the adaptive strategy reduces the execution time of an application by up to 38 %.

Index Terms—Runtime system, Performance analysis, Performance prediction.

I. INTRODUCTION

Servers and computers are increasingly complex systems with more and more CPU cores and parallel capacities. As a result, developing a parallel application that fully exploits the hardware resources is tedious. In order to efficiently write portable parallel programs, developers use runtime systems such as communication libraries, task schedulers, or memory management systems. These frameworks hide the complexity of the hardware and provide common programming interfaces to several hardware resources. Moreover, to ensure performance portability, runtime systems make decisions so as to exploit hardware efficiently. These decisions can have a significant impact on program performance and are usually based on heuristics. These heuristics generally use data collected during the current execution of the program to estimate the future behavior of the application. For example, the first-touch memory allocation policy implemented in Linux allocates a memory page on a NUMA node close to the first thread that accesses it. It assumes that this thread will probably use the memory page in the near future, so it favors memory locality.

However, the heuristic may be wrong and the application may behave differently than expected.

Since the behavior of many parallel applications varies little or not at all from run to run, knowledge of the past executions of a program can provide valuable information about its future behavior. This knowledge of the application's behavior could lead to better decisions than those based on current heuristics. However, runtime systems are supposed to remain program agnostic by design. Therefore, they cannot keep track of program behavior and structure by themselves.

In this paper, we propose PYTHIA, an oracle that provides runtime systems with predictions on the future behavior of an application. This oracle can be used by runtime systems to make better decisions than relying on heuristics. When the application first runs, the runtime notifies events that are relevant for deducing the structure of the program. These events are collected and analyzed by PYTHIA, and stored in a trace file at the end of the execution. At the next execution, the oracle reloads the saved data and uses them to provide predictions to the runtime.

The contributions of this paper are as follows:
- We propose a tool able to capture the behavior of an application and to store this structured data in a file;
- We propose an oracle that compares the current execution of a program with a trace file, and that predicts the future behavior of the application.

We evaluate PYTHIA on 13 MPI or MPI+OpenMP applications, and show that capturing the behavior of these applications does not significantly alter their performance. Moreover, the proposed oracle accurately predicts the future behavior of most applications, even when varying the problem size. Finally, we demonstrate that PYTHIA prediction can be used in a runtime system to dynamically adapt the number of OpenMP threads, which can improve the performance of an OpenMP application by up to 38 %.

The paper is organized as follows. We present the PYTHIA library in Section II. In Section III, we evaluate PYTHIA performance, that is the accuracy and the overhead of PYTHIA prediction, and the usability of PYTHIA, that is how PYTHIA can be integrated in a runtime system to improve the overall performance of an application. We discuss related works in Section IV. Finally, we conclude the paper in Section V.

II. The Pythia oracle library

We propose Pythia[1], a library that allows runtime systems to record events during the execution of a program, and that produces predictions during future executions. During the first execution of a program, called the *reference execution* in the sequel, Pythia-record collects events and stores them in a trace file as a data structure that describes the application's behavior. On subsequent executions, Pythia-predict loads the produced data structure and compares the given new sequence of events to the trace. A runtime system can ask Pythia-predict which events will occur in the future and when these events will occur. The runtime system can make a decision based on this prediction instead of relying on a heuristic.

In Section II-A, we describe how Pythia-record collects events and reduces sequences of events into a grammar that describes the program structure. In Section II-B, we present how Pythia-predict compares the current execution of a program with its reference trace. Finally, in Section II-C, we describe how Pythia-predict predicts which events will happen in the future.

A. On-the-fly program structure compression

During the first execution of the program, the runtime notifies Pythia-record of an *event* when the application reaches a key point. This key point can be, for instance, the entry or exit of a particular function (e.g. `MPI_Send`), the start or end of a construct (e.g. a loop, an OpenMP parallel region), or other types of important events (e.g. the submission of a task to be processed). Each event is composed of an integer that identifies the key point and optionally additional informations such as a timestamp, or the destination of an MPI message.

Pythia-record accumulates the events of each thread into a *trace* while detecting redundancies and repetitive sequences in order to extract the program structure. To do this, we adopted an approach inherited from the algorithm Sequitur [1].

Pythia reduces the sequential list of events into a grammar on the fly. The *grammar* consists of a set of *terminal symbols* and a set of *non-terminal symbols*. The terminal symbols represent the events raised by the runtime and the non-terminal symbols represent the recurrent sequences identified in the trace by the reduction algorithm. Each non-terminal symbol is associated with a finite sequence of terminal and non-terminal symbols, which symbol can be replaced with to recover the complete sequence of terminal symbols it represents. Moreover, each symbol usage is associated with a number of successive repetitions. One of the non-terminal symbols is called the root symbol and represents the complete sequence of the trace. Observe that the trace is the only expression that can be constructed with the grammar.

By convention, we note terminal symbols in lower case and non-terminal symbols in upper case. The root of the grammar is named R. The operator $\mapsto$ associates a non-terminal symbol
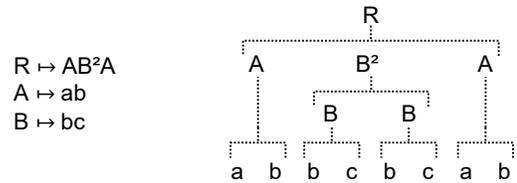
---

[1]Available as open-source at https://github.com/libPythia/pythia

Fig. 1: Unfolding of a grammar representing the trace "$abbcbcab$".

(a) Example of source code	(b) Resulting grammar

```
int i;
for (i = 0; i < 100; ++i) {
    if (i % 2 == 0)
        raise_event_a();
    else
        raise_event_b();
}
```
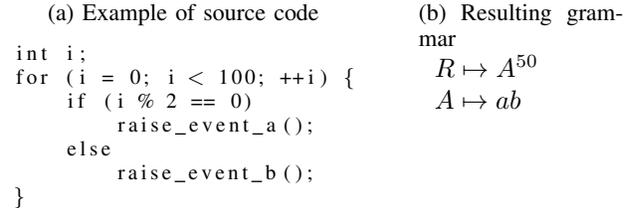
$$R \mapsto A^{50}$$
$$A \mapsto ab$$

Fig. 2: Source code example and the corresponding grammar.

with the sequence of symbols it represents. The number of consecutive repetitions is noted with an exponent and is ignored if it is equal to 1. For example, $A \mapsto ab^2c$ means that the non-terminal symbol A represents the sequence "$abbc$".

The reduction of a sequence of events into a grammar defines the structure of the program, while allowing the sequence to be unfolded by recursively replacing all non-terminals with their corresponding sequences.

For example, Figure 1 illustrates how a grammar can be the reduction of a trace and how the grammar can be unfolded to retrieve the trace. In practice, the data structure of the grammar (which is drawn on the left in the figure), is stored at the end of the first execution but not the corresponding trace (which is drawn on the right in the figure). During subsequent executions, it is the grammar that is loaded in memory and used (without the trace being reconstructed in memory).

Please note that the trace does not represent the exact structure of the source code of the application, but the structure of the execution. For example, Figure 2 shows how a loop containing a simple condition is represented by the grammar. The code in Figure 2a consists in a loop of a hundred iterations. Even iterations raise event a and odd iterations raise event b, but the grammar in Figure 2b represents a loop of fifty repetitions of the sequence "$A \mapsto ab$".

During the execution of the program, Pythia-record updates the grammar according to the events sent by the runtime system and makes sure that at any time the grammar representing the program trace respects three rules:

- All non-terminal symbols are used at least twice in the grammar, which means that each non-terminal symbol represents a sequence that repeats in the trace.
- All couple of symbols appear at most once side by side in the grammar, i.e. all the sequences that repeat in the trace are represented by a non-terminal symbol in the grammar.

(a) Initial 1
$$R \mapsto ...Bb^5 \leftarrow c$$
$$A \mapsto b^3c^2$$
$$B \mapsto b^2A$$

(b) Step 1.1
$$R \mapsto ...Bb^2 \leftarrow C$$
$$A \mapsto Cc$$
$$B \mapsto b^2A$$
$$C \mapsto b^3c$$

(c) Step 1.2
$$R \mapsto ...Bb^2C$$
$$A \mapsto Cc$$
$$B \mapsto b^2A$$
$$C \mapsto b^3c$$

(d) Initial 2
$$R \mapsto ...Bb^2C \leftarrow c$$
$$A \mapsto Cc$$
$$B \mapsto b^2A$$
$$C \mapsto b^3c$$

(e) Step 2.1
$$R \mapsto ...Bb^2 \leftarrow A$$
$$A \mapsto Cc$$
$$B \mapsto b^2A$$
$$C \mapsto b^3c$$

(f) Step 2.2
$$R \mapsto ...Bb^2 \leftarrow A$$
$$A \mapsto b^3c^2$$
$$B \mapsto b^2A$$

(g) Step 2.3
$$R \mapsto ...B \leftarrow B$$
$$A \mapsto b^3c^2$$
$$B \mapsto b^2A$$

(h) Step 2.4
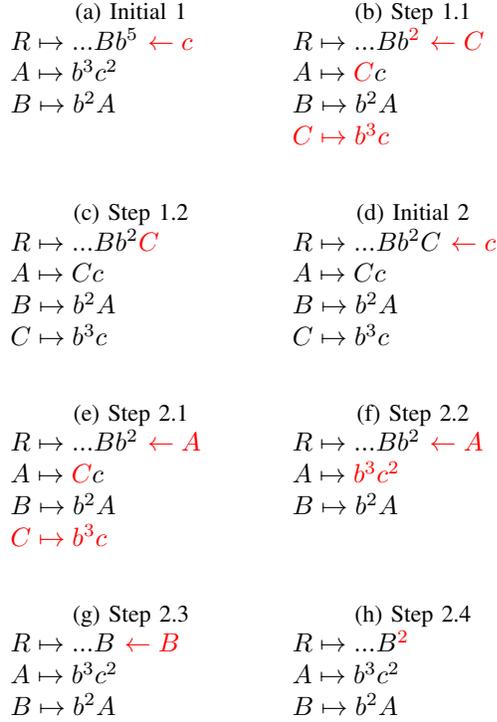$$R \mapsto ...B^2$$
$$A \mapsto b^3c^2$$
$$B \mapsto b^2A$$

Fig. 3: PYTHIA-RECORD adding symbols in the grammar.

- No symbol appears twice side by side in the grammar. All repetitions in the form of "$a^n a^m$" are merged into "a^{n+m}".

When a new event is submitted by the runtime, PYTHIA-RECORD adds the corresponding terminal symbol to the root sequence associated as follow:

- If the last symbol of the sequence is the symbol to be added, PYTHIA-RECORD increments its number of consecutive repetitions;
- Otherwise, if the couple formed by the last symbol of the sequence plus the symbol to be added do not appear anywhere else in the grammar, PYTHIA-RECORD adds the new symbol at the end of the sequence;
- Otherwise, PYTHIA-RECORD either creates a new non-terminal symbol representing the couple or, if possible, reuses an existing one, and then replaces the existing occurrence of the couple by the non-terminal. The last symbol of the root sequence is deleted and the new non-terminal symbol is added recursively to its end.

Figure 3 shows a complete example of the execution of the algorithm by which PYTHIA-RECORD adds the terminal symbol c to an existing grammar twice in a row. As depicted in Figure 3a, the first step consists in adding c at the end of the sequence associated with R and ending with "b^5". Because the sequence "bc" already exists in the grammar as "b^3c", a new non-terminal C is created to represent this sequence and "b^3" is removed from the end of R in Figure 3b. Next, PYTHIA-RECORD adds the symbol C to R. The sequence "bC" is not present in the grammar, so PYTHIA-RECORD appends C to R

in Figure 3c.

When the runtime system submits another c event, PYTHIA-RECORD appends c to the root of the grammar as depicted in Figure 3d. Because the "Cc" sequence is already present in the grammar and is already represented by A, the C sequence is removed from the end of R as displayed in Figure 3e. Also, in Figure 3f, because the C symbol is only used once in the grammar, it is removed from the grammar. Next, PYTHIA-RECORD tries to add A to the end of R. Because sequence "bA" is already present as "b^2A" and is represented by B, "b^2" is removed from the end of R in Figure 3g. Finally, in Figure 3h, B is appended to R so that its consecutive repeat number is incremented.

Along with the grammar reduction, PYTHIA-RECORD optionally records the timestamp of each event occurrence sequentially for later computations. At the end of the execution of the application, PYTHIA-RECORD saves the generated grammar in a file for the following executions of the same application.

B. Predicting the future behavior of the application

Once the grammar has been generated to model the structure of a program, the grammar can be used for subsequent executions. The grammar is loaded at startup, and when the runtime system submits events to the oracle library, they are searched in the grammar. Thereafter, the runtime system requests predictions of events that will occur in the near future.

1) Following the progress of the application: To explain our approach, we proceed by example. Let's start with the grammar of the reference execution that is depicted in Figure 1. The application is executed again and let us start the execution of PYTHIA-PREDICT at random, for example from the raising of an event corresponding to the terminal symbol b. We do not start the execution of PYTHIA-PREDICT at the beginning of the execution to show that we do not need this strong assumption; it is this tolerance that allows us to tolerate unforeseen events (with respect to the grammar of the reference execution). Let us start with a receiving b. PYTHIA-PREDICT tries to find the parts of the reference execution corresponding to the new sequence of events that begins with b. The reference execution modeled in the grammar contains four occurrences of terminal symbol b. Thereafter, the runtime submits event c. Of the four occurrences of symbol b, only two of them are followed by an occurrence of symbol c, so that PYTHIA-PREDICT can identify that the application is currently executing one of the occurrences of sequence B. If the runtime submits the next b event, PYTHIA-PREDICT then assumes that the application is starting a new occurrence of the B symbol sequence. Again, we don't need to know if sequence A preceded sequence B. The "Bb" sequence in this example is what we call in the following the progress sequence.

When the runtime system submits events, PYTHIA-PREDICT tracks the progress of the application by locating the submitted events in the grammar. Each occurrence of an event can be denoted by the *progress sequence*, which is the path from the terminal towards the root of the grammar. For example, as
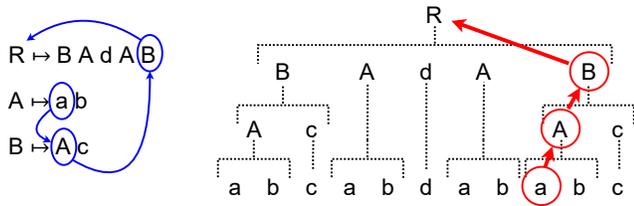
Fig. 4: Progress sequence (in blue) representing the fourth occurrence of a in the grammar representing sequence "$abcabdababc$", with the corresponding path (in red) if the grammar were unfolded.

(a) Initial   (b) Step 1   (c) Step 2

(d) Step 3

Fig. 5: PYTHIA-PREDICT updates the progress sequence.

Fig. 6: Progress sequence representing all the occurrences of terminal b following an occurrence of terminal a and followed by an occurence of terminal c in the grammar modeling the trace "$abcabdababc$".

displayed in Figure 4, in the grammar modeling the sequence "$abcabdababc$", the fourth occurrence of a is the first symbol of the sequence of A (which corresponds to the sequence "ab"), which is itself the first symbol of the sequence of B (which corresponds to the sequence "Ac") that is the final symbol of the root symbol. As a result, if the application behaves exactly as in its previous execution, each event received by PYTHIA-PREDICT is associated with the same event in the sequence reduced in the grammar by PYTHIA-RECORD. In this case, the maintenance of the correspondence between the reference trace (corresponding to the grammar) and the trace of the current execution consists in updating a unique progress sequence with the following depth first traversal algorithm.

Figure 5 illustrates how PYTHIA-PREDICT updates a progress sequence. Initially, in Figure 5a, the progress sequence is "bA", i.e. it points to the third occurrence of the terminal b in the trace "$abcabdababc$". In Figure 5b, it removes the terminal b from the progress sequence because b has no successor in the sequence associated with A. In Figure 5c, it replaces the first element of the progress sequence with its successor, in this case the last B of R. Finally, in Figure 5d, PYTHIA-PREDICT completes the progress sequence by successively adding the non-terminals until it reaches the first terminal symbol, here by adding a. The progress sequence is updated to "aAB", i.e. it points to the fourth occurrence of the terminal a in the trace "$abcabdababc$".

2) Tolerance to unexpected events: During the execution, the program behavior may differ from the reference execution. When this happens, the runtime submits an event that does not appear in the next possible progress sequences. If the event raised never occurred in the first execution, the oracle has no information about the possible behavior of the program and the runtime system must again temporary rely on heuristics. However, if the event was present in the reference execution, the program probably triggered a different code path, or skipped some function calls, or even made them in a different order. In this situation, the oracle can build partial knowledge of the execution state by listing all the possibles sequences starting with this event. To do this, PYTHIA-PREDICT stores the progress sequences containing only the terminal corresponding to the last event. From then on, at each new event, PYTHIA-PREDICT tries to extend the progress sequence by
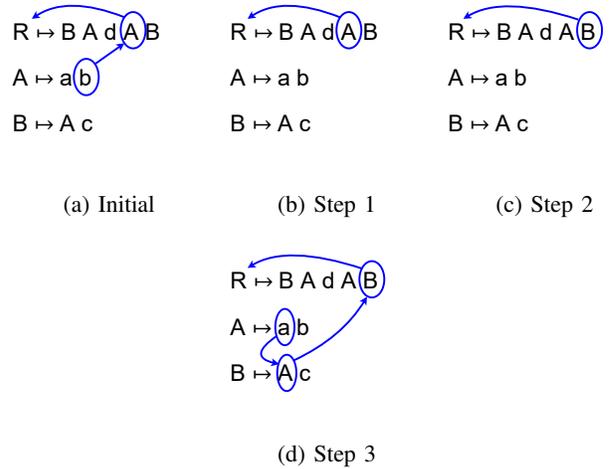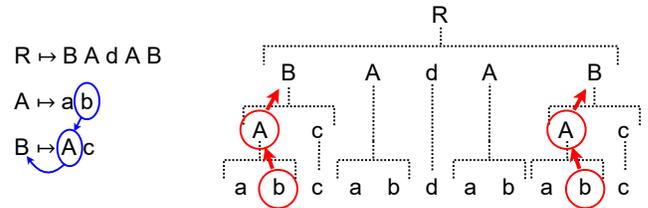
adding a non-terminal whenever it recognizes the associated sequence in the runtime event sequence.

C. Predicting the following events

By applying the algorithm seen previously, predicting future events boils down to simulating the future execution from a copy of the current progress sequences. Since PYTHIA-PREDICT is capable of tracking multiple progress sequences at the same time, it must estimate the probability of each. Given a set of possible progress sequences, estimating the probability that each one occurring relative to the others is equivalent to counting the number of uses of them in the reference execution event sequence. In grammatical form, the probability is calculated using a recursive algorithm that counts the number of occurrences of the symbol in the grammar.

Optionnaly, PYTHIA-PREDICT can predict the duration before future events are raised. This is done by recording the timestamps of all events raised by the runtime during the reference execution. Practically, at the end of the first run, PYTHIA-RECORD replays the sequence of events in the grammar using the same prediction algorithm: At each step of the "replay", list all the possible associated progress sequences, and for each of them, calculate the average elapsed time from the previous event to the one designated by the progress sequence.

These average elapsed time are saved with the grammar so that PYTHIA-PREDICT is able to recover an estimate of the duration between the last event raised by the runtime and future events it predicts.

Thanks to the progress sequence, the time prediction takes into account the context. For example, PYTHIA-RECORD associates to the "BAb" progress sequence represented in Figure 6 the average duration between an a event and a b event in the specific case where a c event is expected next. As previously seen, this progress sequence represents two occurrences of event b in the reference execution. The duration between an a event and a b event when the next event is not known is associated to the "Ab" progress sequence and corresponds to the average time between the four possible occurrences of a b event following an a event.

III. EVALUATION

We evaluate PYTHIA and its usefulness as an oracle for runtime systems[2]. In Section III-A, we describe the evaluation environment and the evaluated applications. In Section III-B, we describe two runtime systems that rely on PYTHIA for recording events and then for predicting the future behavior of an application. In Section III-C, we evaluate the overhead of PYTHIA-RECORD, we assess the generated grammars, and we measure the accuracy and the cost of PYTHIA-PREDICT predictions. In Section III-D, we design and evaluate a runtime system that uses PYTHIA-PREDICT prediction to improve the performance of an application. Finally, in Section III-E, we assess PYTHIA resilience to unexpected events.

A. Experimental environment

1) Experimental platform: We run experiments on three different machines:

1) **Pudding** is equipped with two Intel Xeon Silver 4116 CPUs with 12 cores / 24 threads each (total: 24 cores) running at 2.1 GHz, and 64 GiB of DRAM. The machine runs Linux kernel version 5.4.0;
2) **Pixel** is equipped with two Intel Xeon E5-2630 v3 with 8 cores / 16 threads each (total: 16 cores) running at 2.4 GHz, and 32 GiB of DRAM. The machine runs Linux kernel version 5.4.0;
3) **Paravance** is a cluster of 72 machines, each equipped with 2 Intel Xeon E5-2630 v3 CPUs with 8 cores each (total: 16 cores per machine) running at 2.4 GHz, and 128 GiB or RAM; the machines are interconnected with a 10Gbps Ethernet network. They run Linux kernel version 5.10.0;

2) Evaluated applications: We evaluate PYTHIA on several applications. For each application, we define three types of working set: **small**, **medium**, and **large**.

- **NAS Parallel Benchmarks** (version 3.3.1) is a set of programs designed to evaluate supercomputers. We use the MPI implementation of the benchmarks. We evaluate

[2]Scripts for reproducing our experiments are available at https://github.com/libPythia/Pythia_cluster22_reproductibility

the following kernels: BT, CG, EP, FT, IS, LU, MG, SP. The small, medium, and large working sets use the NAS Parallel Bencharks problem sizes A, B, and C;
- **AMG** is a parallel algebraic multigrid solver that mixes MPI and OpenMP. The working sets use the following parameters: `-n 100 100 100` (small), `-n 150 150 150` (medium), and `-n 200 200 200` (large);
- **Lulesh** is an MPI+OpenMP application that solves a Sedov blast problem [2]. The working sets use the following parameters: `-s 10` (small), `-s 30` (medium), and `-s 50` (large);
- **Kripke** is a deterministic particle transport application that uses MPI and OpenMP. The working sets use the following parameters: `--procs 2,2,2 --groups 128` (small), `--procs 2,2,2 --groups 512` (medium), and `--procs 2,2,2 --groups 1024` (large);
- **miniFE** is a proxy application for unstructured implicit finite element codes. This application mixes MPI and OpenMP. The working sets use the following parameters: `-nx 100 -ny 100 -nz 100` (small), `-nx 200 -ny 200 -nz 200` (medium), and `-nx 300 -ny 300 -nz 300` (large);
- **Quicksilver** is a proxy application that solves a dynamic monte carlo particle transport problem. This application mixes MPI and OpenMP. The working sets use the following parameters: `--lx 500 --ly 500 --lz 500 -n 1000000` (small), `--lx 500 --ly 500 --lz 500 -n 10000000` (medium), and `--lx 500 --ly 500 --lz 500 -n 20000000` (large).

B. Runtime systems

To assess PYTHIA usability and performance, we develop two runtime systems.

The **MPI** runtime system mimics the behavior of an MPI implementation that uses PYTHIA to optimize the communication patterns of an application. Implementing an actual communication optimization is beyond the scope of this paper, but the optimization could consist in aggregating multiple successive MPI send messages [3], or setting up persistent communication if a communication pattern repeats. To implement the MPI runtime system, we intercept the application calls to MPI primitives using `LD_PRELOAD`. For each MPI function call, we record an event with PYTHIA-RECORD. Each event consists of the MPI function being called, as well as an additional information for some functions: the source or destination rank for point-to-point communication primitives (such as `MPI_Send`, or `MPI_Recv`), the reduction operation for reduction primitives (such as `MPI_Reduce`), and the root rank of collective communication primitives (such as `MPI_Bcast`). In addition to submitting events to PYTHIA, the MPI runtime system asks for predictions when entering an `MPI_Wait` function (including `MPI_Waitall`, etc.), or when entering a collective communication primitive (such as `MPI_Barrier`). This mimics the behavior of an MPI runtime system that would use the synchronization time to perform an optimization.

The **OpenMP** runtime system mimics the behavior of an OpenMP implementation that uses PYTHIA to optimize the handling of OpenMP threads. We describe and evaluate an example of such optimization in Section III-D1. Our OpenMP runtime system intercepts the application calls to GNU OpenMP runtime system functions (such as `GOMP_parallel`, or `GOMP_critical_start`). In addition to submitting events to PYTHIA, the OpenMP runtime system asks for predictions when starting an OpenMP parallel region. In Section III-D1, we further modify this runtime system to implement an optimization that relies on PYTHIA-PREDICT predictions.

C. Performance evaluation of PYTHIA

1) Performance evaluation of PYTHIA-RECORD*:* To evaluate the cost of recording events, we run the applications presented in Section III-A2 with and without PYTHIA-RECORD. The applications run on 4 nodes of the **Paravance** cluster, and we use the **large** working sets. We run the NAS Parallel Benchmarks applications with 64 MPI ranks (16 ranks per machine), and the other applications (AMG, Lulesh, Kripke, miniFE, and Quicksilver) run with 8 MPI ranks (2 ranks per machine), each running 8 OpenMP threads. When running the applications with PYTHIA-RECORD, we use our MPI runtime system for NAS Parallel Benchmarks applications, and our MPI and OpenMP runtime systems for AMG, Lulesh, Kripke, miniFE, and Quicksilver.

Table I reports the measured execution time when relying on current heuristics of the operating system (Vanilla), and when running the applications with PYTHIA-RECORD. The table also reports the total number of events recorded, and the average number of rules in the generated grammar. The reported performance is the average measured execution time over 10 runs. The results show that recording events with PYTHIA-RECORD does not significantly impact the performance of most applications. The measured overhead of PYTHIA-RECORD ranges from -5.8 % for miniFE, to +4.9 % for Quicksilver, and most applications show an overhead that ranges between -1.1 % and +1.4 %. The unexpected performance impovement we observe may be due to PYTHIA-RECORD memory allocations disrupting the programs behavior.

The number of collected events varies from a few hundred events for applications that only perform a few MPI collective communications (such as EP or FT), to several millions events for applications that extensively use MPI (such as LU), or MPI and OpenMP (such as Lulesh or Quicksilver).

As the runtime systems submit events to PYTHIA-RECORD, a grammar that represents the program execution is maintained for each thread. The size of this grammar can be measured as the average number of rules of the grammar. The results show that for most applications, PYTHIA-RECORD builds a grammar that consists of less than 15 rules. A manual analysis of the generated grammars shows that their structure is similar to the application structure. For example, Figure 7 shows the grammar generated by one of the MPI ranks when running BT. This grammar consists of a loop that contains a few calls

$R \mapsto \text{Bcast}^6 B \text{ Barrier } A^{200} \text{ Allreduce Allreduce } B$
$\qquad \text{Reduce Barrier}$
$A \mapsto B \text{ Isend Irecv } [...] \text{ Wait}^2$
$B \mapsto \text{Irecv Irecv } [...] \text{ WaitAll}$

Fig. 7: Overview of the grammar extracted from BT.large. (The `MPI_` prefix is removed to simplify the presentation.)

to MPI functions. This structure is similar to the behavior of the application that iterates 200 times.

A few applications have irregular execution patterns that generate a large number of rules. For example, Quicksilver sends a particle when it exits the domain of an MPI worker. As a result, its MPI communication pattern depends on the particles' position, and the grammar generated by PYTHIA-RECORD becomes complex.

This experiment shows that most of the evaluated applications have repetitive patterns and can be summarized with simple grammar rules. It also shows that the collection of events and the generation of a grammar at runtime does not significantly impact the performance of applications.

2) Accuracy of PYTHIA-PREDICT *predictions:* To evaluate the accuracy of PYTHIA-PREDICT predictions, we measure the success rate of predictions. We generate traces of the evaluated applications with PYTHIA-RECORD using the **small** working sets, and we provide these traces to PYTHIA-PREDICT when running the applications with the different working sets.

The event prediction is implemented as follows. When entering an MPI blocking function such as `MPI_Wait`, or any MPI blocking collective primitive (e.g. `MPI_Allreduce`), PYTHIA-PREDICT predicts the event that will happen in x events, and we vary the value of x. For example, if $x = 1$, we predict the next event. We then count how many predictions are correct, or incorrect (which means that PYTHIA-PREDICT predicted event e_i, but event e_j occurred).

Figure 8 reports the results we measured. The results show that the accuracy of short-term prediction is high for all the applications. As the *distance* of the prediction increases, the accuracy of PYTHIA-PREDICT slowly decreases while remaining high for most applications. For 8 out of the 13 tested applications, PYTHIA-PREDICT achieves a prediction accuracy higher than 90 % at a distance of 128. Despite the irregular behavior of Quicksilver and AMG (as described in Section III-C1), PYTHIA-PREDICT accuracy remains above 70 % for those applications for short-distance predictions. As the prediction distance increases, the predictions become inaccurate. Some applications such as LU or MG perform the same algorithm for any working set, but the number of iterations of the algorithm depends on the size of the data set. This causes PYTHIA-PREDICT to mispredict events when reaching the loops boundaries.

This experiment shows that most of the tested applications have the same behavior even when the working set changes. PYTHIA-PREDICT is able to accurately predict the future behavior of most applications, even when the distance of

Application	Vanilla (s)	PYTHIA-RECORD (s)	overhead(%)	# events	# rules
BT.Large	24.2	24.2	0.7	2,329,920	3
CG.Large	9.9	9.9	-0.3	3,837,890	15
EP.Large	4.2	4.1	-3.8	384	1
FT.Large	17.4	17.4	0.2	3,072	2
IS.Large	3.2	3.2	0.1	2,493	2
LU.Large	23.0	23.3	1.4	18,164,200	11
MG.Large	4.2	4.1	-0.5	609,888	14
SP.Large	24.3	24.4	0.2	356,870	9
AMG.Large	38.7	38.4	-0.9	118,438	150
Lulesh.Large	125.6	124.2	-1.1	28,150,300	12
Kripke.Large	59.8	61.0	2.0	9,881	46
miniFE.Large	25.8	24.3	-5.8	39,272	8
Quicksilver.Large	35.9	37.6	4.9	26,786,800	409

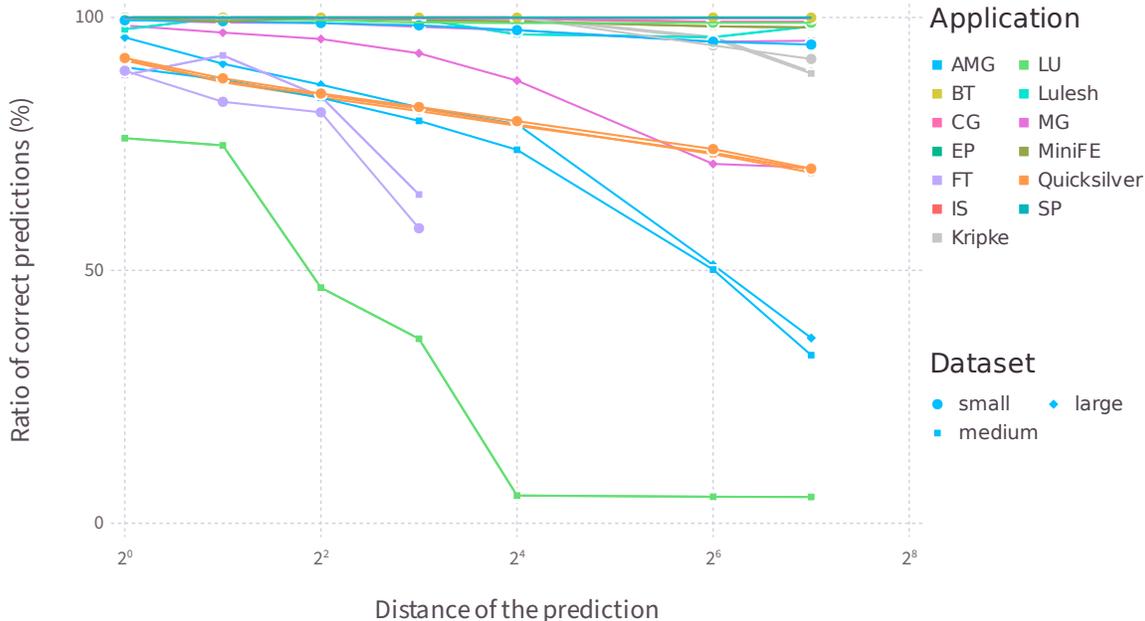TABLE I: Performance evaluation of PYTHIA-RECORD.

Fig. 8: Accuracy of PYTHIA-PREDICT predictions on Paravance.

prediction is high.

3) Overhead of PYTHIA-PREDICT *predictions:* When a runtime system requests a prediction from PYTHIA-PREDICT, the response time of the oracle is critical because runtime system uses the prediction to perform an optimization. The response time thus has to be lower than the expected gain of the optimization. We now evaluate the cost of a PYTHIA-PREDICT prediction while varying the prediction distance.

Figure 9 reports the average duration of a prediction for several applications running the **large** working set. The cost of a prediction grows linearly when the distance of the prediction increases. We observe that the cost of a prediction varies from one application to another, and that irregular applications with complex grammar rules induce higher costs of prediction. This is because PYTHIA-PREDICT browses through the grammar graph to predict future events. Browsing a graph that is composed of many nodes can become costly for applications

such as Quicksilver.

For most applications, the cost of short distance predictions is between a few hundreds nanoseconds to less than 2 μs, which would allow a runtime system to perform a fine-grain optimization. Predicting an event that happens in a far distance is costlier because it requires to browse the grammar graph extensively. The cost of prediction for a distance of 64 is less then $20\mu s$ for most applications, which would allow a runtime system to conduct coarse-grain optimization such as prefetching data.

D. Evaluation of the usability of PYTHIA

We now evaluate the usability of PYTHIA to perform an optimization in a runtime system.

1) Implementing an optimization based on PYTHIA-PREDICT *predictions:* GNU OpenMP is the runtime system used by the GCC implementation of OpenMP. It manages
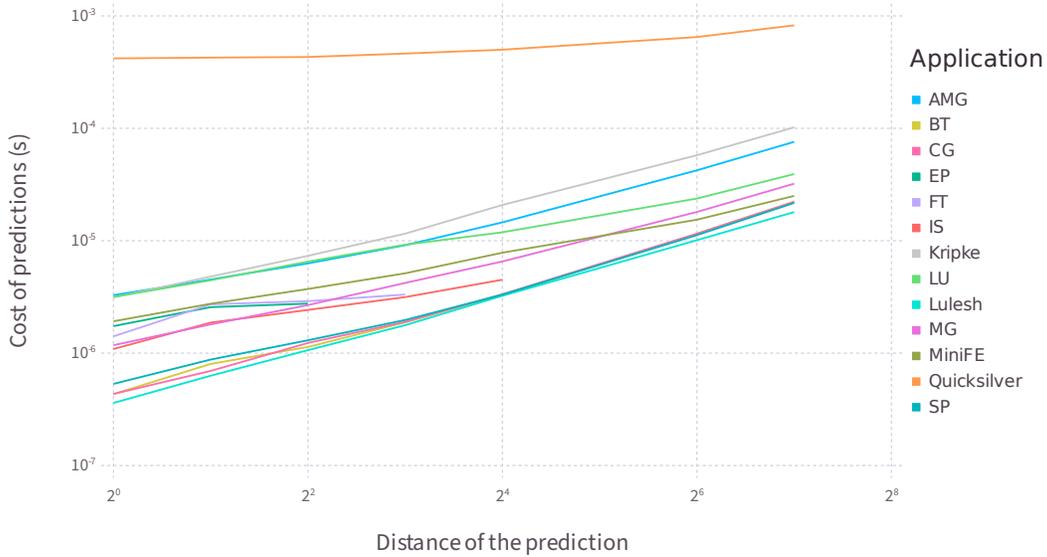
Fig. 9: Cost of PYTHIA-PREDICT predictions on Paravance.

threads and distributes the OpenMP workload among the threads. In this section, we show how PYTHIA can be used in GNU OpenMP to predict the future behavior of an OpenMP application to improve the overall performance of the program.

To parallelize a piece of code, the programmer can annotate it with OpenMP directives and the OpenMP runtime automatically spawn threads (or reuse existing ones) to process that parallel region. When deciding how many threads to use for a parallel region, the OpenMP runtime trades off the speedup due to many threads processing a workload in parallel against the cost of synchronizing the threads. The GNU OpenMP runtime usually chooses the maximum number of threads to process a parallel region, which can be expensive if the application consists of many small parallel regions.

We modified the GNU OpenMP runtime so that it uses PYTHIA in order to decide how many threads should be used for each OpenMP parallel region. These modifications consist in:

- Submitting events to PYTHIA-RECORD at the beginning and end of each parallel region. We use the function pointer that contains the code of the parallel region as an event identifier to distinguish the different regions of the application;
- Requesting a prediction to PYTHIA-PREDICT at the beginning of a parallel region. Thanks to this prediction, the runtime system knows the probable duration of the parallel region. Based of the estimated duration D_{est}, GNU OpenMP decides how many threads should be used, e.g. 1 thread if $D_{est} < t_1$, 4 threads if $D_{est} < t_4$, 8 threads if $D_{est} < t_8$, and so on.

Overall, exploiting PYTHIA predictions in GNU OpenMP required less than 100 lines of code.

In addition to these changes, we have also changed the way GNU OpenMP manages its threads pool. By default, GNU OpenMP destroys spurious threads when the number of OpenMP threads (specified by `omp_set_num_threads`) decreases. In order to reduce the overhead of creating and destroying threads when the number of OpenMP threads varies, we have made the spurious threads wait until they are needed again.

2) Application: Our illustrative use case is the OpenMP version of Lulesh that contains 30 parallel regions of different sizes. For our experiments, we modify Lulesh slightly in two ways. First, some parallel regions of Lulesh were written as if the number of OpenMP threads could not change during program execution. The fix consists of a few calls to `omp_get_num_threads` instead of keeping the maximum number of threads in a variable. Second, the Lulesh memory allocation model generates many page faults. Surprisingly, when recording events with PYTHIA-RECORD, the allocation model is disrupted and the performance of Lulesh improves by 15 %. To remove this bias in our measurements, we modify Lulesh to reuse memory from one computation step to the next to reduce the number of page faults and thus mitigate the variation in performance.

All the measurements are obtained over ten runs and we report the minimum, maximum, and average execution times. Unless otherwise specified, we run 1 thread per core, *i.e* we do not use hyperthreading, and threads are bound in a round-robin fashion.

3) Performance evaluation of the proposed optimization: Figures 10 and 11 shows how the execution time of Lulesh varies with the problem size when running on machines Pudding and Pixel. Note that the Y axis is in logarithmic scale. The measurements show that event recording with PYTHIA-RECORD does not significantly affect performance
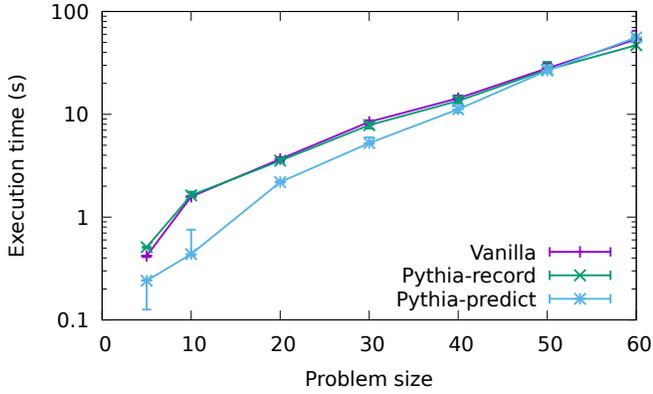
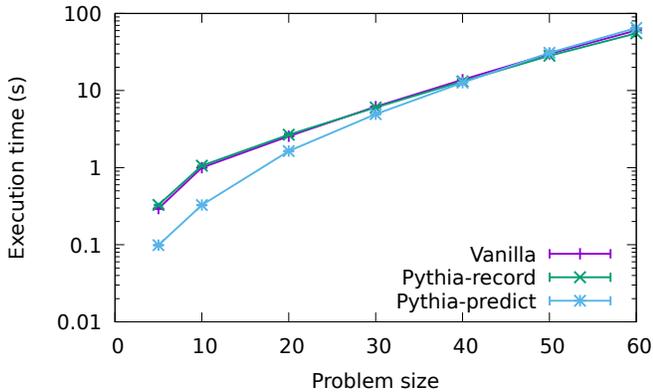Fig. 10: Execution time of Lulesh as a function of the problem size (on Pudding with 24 threads).

Fig. 11: Execution time of Lulesh as a function of the problem size (on Pixel with 16 threads).
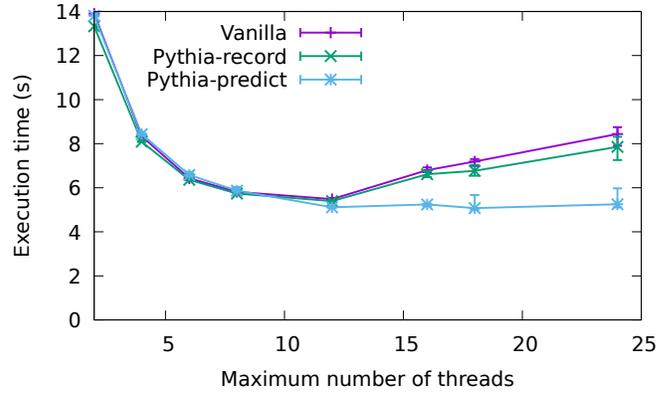

Fig. 12: Execution time of Lulesh as a function of the maximum number of threads (on Pudding for a problem size of 30).

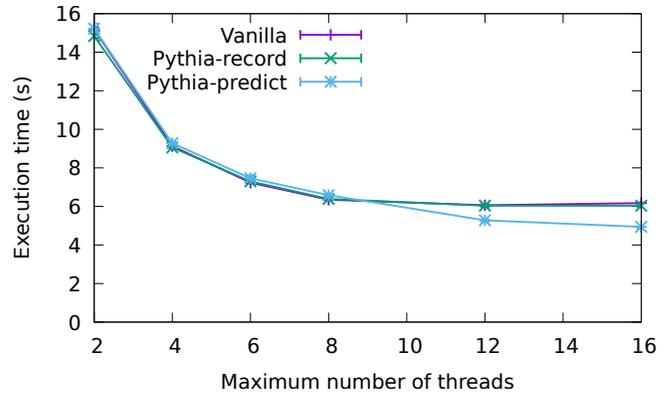

Fig. 13: Execution time of Lulesh as a function of the maximum number of threads (on Pixel for a problem size of 30).

compared to VANILLA. On the other hand, PYTHIA-PREDICT significantly reduces the execution time, especially for small problems. For example, for a problem size 30 on Pudding, Vanilla executes in 8.44s, while PYTHIA-PREDICT executes in 5.25s, improving performance by 38 %. As the problem size increases, the performance improvement of PYTHIA-PREDICT slowly decreases. This is due to the small parallel regions that account for a significant portion of the run time for small problems, but become negligible for larger problems. Therefore, running the maximum number of threads for large problems is the optimal solution, and the prediction of PYTHIA-PREDICT does not help the runtime system.

To further analyze the performance implications of PYTHIA, we now run Lulesh with a problem size of 30, and vary the maximum number of threads. In this experiment, VANILLA and PYTHIA-RECORD use the maximum number of threads, and PYTHIA-PREDICT dynamically adapts the number of threads while respecting the specified maximum number. Figures 12 and 13 report the performance measured on Pudding and Pixel.

When the maximum number of threads is low (typically up to 8 threads), all three OpenMP implementations have

similar performance. For larger numbers of threads, PYTHIA-PREDICT improves performance by up to 38.8 % (on Pudding) and 20.0 % (on Pixel). This is due to VANILLA and PYTHIA-RECORD suffering from the synchronization overhead during the many small parallel regions, while PYTHIA-PREDICT dynamically lowers the number of threads for these regions.

E. Resilience to unexpected events

In order to evaluate the performance of PYTHIA-PREDICT when the runtime system generates unexpected events, we modify GNU OpenMP to randomly submit unexpected events with a given *error rate*, and measure the performance on Lulesh. Figure 14 shows how Lulesh (with a problem size of 30 on Pudding) performs with PYTHIA as the error rate increases. The results show that for low error rates, PYTHIA-PREDICT performs significantly better than VANILLA and PYTHIA-RECORD. When the error rate increases, the performance improvement obtained with PYTHIA-PREDICT decreases. This is due to the predictions becoming less and less reliable, which causes the OpenMP runtime to make bad
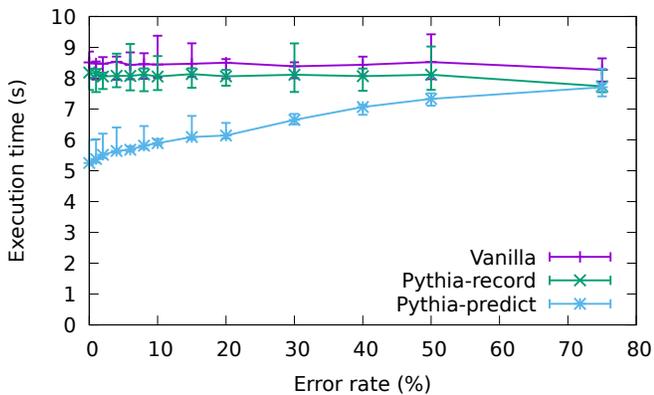
Fig. 14: Execution time of Lulesh as a function of the error rate (on Pudding for a problem size of 30).

decisions such as using the maximum number of threads for a small parallel region.

IV. RELATED WORKS

Over the past few decades, the detection of the structure of execution traces has been studied for multiple purposes. Several works focus on generic data compression [4], [5], or on execution traces compression [6]. ScalaTrace detects the MPI communication patterns of parallel programs in order to compress execution traces [7]. Some works extract the structure of an execution trace by finding different types of redundancies in them [8], [9], while another work simplifies repetitions in execution traces with tolerance to small variations [10]. DiffTrace [11] reduces traces and compares them for debugging purpose. Combining the source code with compiler debugging information can also reduce execution traces [12]. These works are used for reducing the size of trace files, or for helping developers debug or understand the structure of an application.

The idea of representing a trace as a grammar has been studied in several works. Sequitur [1] is an incremental algorithm used for compressing various types of data under the form of a grammar. It has a linear time and space complexity, but suffers from drawbacks for detecting some control flow from execution traces. Sequitur(1) is an improvement of Sequitur that is used to dynamically represent the flow of control of programs [13]. Cyclitur [14] has extended Sequitur to add the notion of consecutive repetitions, and is applied to compression and anomaly detection in execution traces of embedded programs.

Other works are more related to PYTHIA as they analyze traces in order to predict the future behavior of an application. NLR [15] reduces sequences by inferring nested loop generators from their variation patterns and uses it for memory access prediction. Omnisc'IO [16] uses a grammar algorithm named StarSequitur and applies to I/O access prediction. While these work predict the future behavior of an application, they are specific to one type of resource usage prediction, whereas

PYTHIA provides a generic prediction interface that can be used for any kind of runtime system optimization.

V. CONCLUSION

We have proposed an oracle library that provides runtime systems with predictions of the future behavior of an application. This oracle relies on the deterministic nature of many parallel applications and compares the current execution with a previous execution in order to predict the program future behavior in the current execution. The evaluation shows that PYTHIA captures the behavior of an application as a grammar without altering the program performance, that the collected data can be used to guide a runtime system during future executions of the same application. To demonstrate the use of PYTHIA in a runtime system, we integrate it into the GNU OpenMP runtime system in order to predict the duration of the application's OpenMP parallel regions. Using the prediction of PYTHIA, GNU OpenMP dynamically selects the number of threads for each parallel region. Experiments show that PYTHIA correctly predicts the future behavior of the program, and that this information can significantly improve the performance of the application. Further investigations are needed to make Pythia able to predict accurately when the application runs with different configuration (number of threads, number of processes,...)

This paves the ways for other types of runtime optimizations. Instead of relying on heuristics, a runtime system can base its decision on the expected future of the application.

REFERENCES

[1] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *Journal of Artificial Intelligence Research*, vol. 7, pp. 67–82, 1997.

[2] I. Karlin, J. Keasler, and J. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[3] O. Aumage, E. Brunet, N. Furmento, and R. Namyst, "New madeleine: A fast communication scheduling engine for high performance networks," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.

[4] E.-H. Yang and J. C. Kieffer, "Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. i. without context models," *IEEE Transactions on Information Theory*, vol. 46, no. 3, pp. 755–777, 2000.

[5] J. Kieffer and E.-h. Yang, "Lossless data compression algorithms based on substitution tables," in *Conference Proceedings. IEEE Canadian Conference on Electrical and Computer Engineering (Cat. No. 98TH8341)*, vol. 2. IEEE, 1998, pp. 629–632.

[6] A. Milenkovic and M. Milenkovic, "Stream-based trace compression," *IEEE Computer Architecture Letters*, vol. 2, no. 1, pp. 4–4, 2003.

[7] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. De Supinski, "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 696–710, 2009.

[8] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting sequence diagram from execution trace of java program," in *International Workshop on Principles of Software Evolution (IWPSE)*, 2005, pp. 148–151.

[9] F. Trahay, E. Brunet, M. M. Bouksiaa, and J. Liao, "Selecting points of interest in traces using patterns of events," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, pp. 70–77.

[10] A. Knupfer and W. E. Nagel, "Construction and compression of complete call graphs for post-mortem program trace analysis," in *International Conference on Parallel Processing (ICPP)*, 2005, pp. 165–172.

[11] S. Taheri, I. Briggs, M. Burtscher, and G. Gopalakrishnan, "Difftrace: Efficient whole-program trace analysis and diffing for debugging," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–12.

[12] D. Myers, M.-A. Storey, and M. Salois, "Utilizing debug information to compact loops in large program traces," in *European Conference on Software Maintenance and Reengineering*, 2010, pp. 41–50.

[13] J. R. Larus, "Whole program paths," *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 259–269, 1999.

[14] A. Amiar, M. Delahaye, Y. Falcone, and L. Du Bousquet, "Compressing microcontroller execution traces to assist system analysis," in *International Embedded Systems Symposium*, 2013, pp. 139–150.

[15] A. Ketterlin and P. Clauss, "Prediction and trace compression of data access addresses through nested loop recognition," in *IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 94–103.

[16] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 623–634.