# Detecting Interference Between Applications and Improving the Scheduling Using Malleable Application Proxies

Alberto Cascajo(✉), David E. Singh, and Jesus Carretero

Computer Science and Engineering Department, Universidad Carlos III de Madrid, Madrid, Spain
acascajo@inf.uc3m.es

**Abstract.** LIMITLESS is a lightweight and scalable framework that provides a holistic view of the system employing the combination of both platform and application monitoring. This paper presents a novel feature for improving the scheduling process based on the performance prediction and the detection of interference between real applications. This feature consists of using malleable synthetic benchmark clones (proxies) for the applications executed in the system with two objectives: (1) build large and representative datasets that can be used to train the machine learning algorithms for predicting, and (2) evaluate if two applications can share the same compute node in order to leverage the unused node resources.

Other related works use detailed micro-architecture independent metrics obtained from functional simulators, which are hard to generate in many new applications. The results are proxies that preserve many of the original features of the applications (control flow, memory access pattern, etc.), and their code needs obfuscation to make impossible the use of reverse engineering. LIMITLESS generates application proxies based on generic-purpose performance information collected from monitoring. It means that other methods may obtain more accurate execution behaviours. However, LIMITLESS' proxies generate similar performance without extracting data from the binaries, without the necessity of managing code or data from the applications, and they can be shared securely because they have not been generated using any piece of the original code.

LIMITLESS leverages the generated proxies to execute them offline. Each execution increases the datasets of the machine learning algorithms to improve the application scheduling. Besides, the executions between proxies are combined to detect performance degradation (interference) without the necessity of waiting for the execution of the real applications, which depends on the users. In this work, we evaluate the proposed proxy generation approach on a set of benchmarks and applications. We compare the performance obtained during the execution of the proxies and

This work has been partially funded by the European High-Performance Computing Joint Undertaking (JU) under the ADMIRE project (grant agreement No 956748) and the Spanish Ministry of Science and innovation Project DECIDE (Ref. PID2019-107858GB-I00.).

© Springer Nature Switzerland AG 2022
H. Anzt et al. (Eds.): ISC High Performance 2022 Workshops, LNCS 13387, pp. 129–146, 2022.
https://doi.org/10.1007/978-3-031-23220-6_9

the applications to show their similarity. Finally, we include an evaluation of the interference detection using this approach. As far as we know, this is the first work that uses malleable proxies.

**Keywords:** Malleable proxy · Malleable synthetic benchmarks · Performance cloning · Interference detection · Application scheduling

## 1 Introduction

One of the key challenges in large-scale clusters is to determine as accurately as possible the status of the system. In this work, we combine system and application monitoring in order to provide, not only a more accurate cluster monitoring but also a scheme that permits to model the application behaviour. The goal is to generate proxies that can be used as benchmarks and to use those proxies to generate more information to improve the application scheduling in two ways: by predicting the performance of the applications and by detecting interference between applications. Initially, we depend on the user and the executions he wants to run. However, LIMITLESS can perform different actions without waiting for the original executions due to the proxies.

The use of benchmarks is one of the keys for assessing computer systems performance. Researchers and engineers need to quantify the performance of their applications by running them many times and in different architectures. Some uses of those benchmarks are to compare the design alternatives during development, test computer systems for guiding development, or enable a fair evaluation of the performance in different architectures. For example, SPEC, CPU2006, ImplanBench, PARSEC, etc., are benchmarks that provide suites for evaluating the performance of general-purpose processors. These standard benchmarks are generally generated based on open-source programs. Their main limitation is that they are not representative of real-life applications, and usually, they are very different from the applications of interest to the developers and researchers. The alternative consists of using real-life applications, but the code are typically proprietary. The industry could benefit from the researchers because they can improve their applications: the computer systems could be designed to provide a good performance of these applications, or by applying new optimizations. And the researchers could benefit from the industry by using their real applications to find better design solutions or studying new research lines based on the results.

This paper presents a new alternative for proxies creation based on the generic performance information obtained by the LIMITLESS system monitor. The monitor collects the performance metrics during the execution of an application in a compute node and stores them in a database. Then, the analytic component processes those metrics to generate a malleable proxy (using FlexMPI [10]) that reproduces the same performance metrics and can be reconfigured in run time.

The proposed proxy generation features three key properties: (1) no information of the proprietary application is revealed, (2) the performance metrics obtained by executing the proxy are similar to the original application so that the proxy can serve as a benchmark to evaluate possible performance behaviours

in whatever architecture, and (3) related to the last point, an intelligent scheduler could combine executions of proxies, and applications and proxies, to check if there is interference between them, allowing the system to share nodes between non-conflicting applications. Note that the malleability in already implemented in the proxies due to their integration with FlexMPI.

The main contributions of this work are:

- A proxy generation feature to provide synthetic malleable micro-benchmarks based on the collected performance behaviour.
- An improvement in the application scheduling employing machine learning algorithms trained with proxies executions.
- A methodology to improve the application scheduling through the malleable proxies, combining them with the real applications to identify interference. In this context, malleability means that the system can use a single proxy for evaluating different configurations (number of processes) at run-time.

The structure of the paper is as follows: Sect. 2 describes the architecture organization; Sect. 3 describes LIMITLESS's features for providing proxy generation, Machine Learning training algorithms, and the studies of the interference between applications; Sect. 4 provides a practical evaluation of the performance metrics obtained from the proxy executions, the accuracy of the prediction algorithms, and the results of the studies related to the interference between applications; Sect. 5 shows relevant works related to our proposal. Finally, Sect. 6 summarizes the main conclusions and future work.

## 2 Monitor Architecture

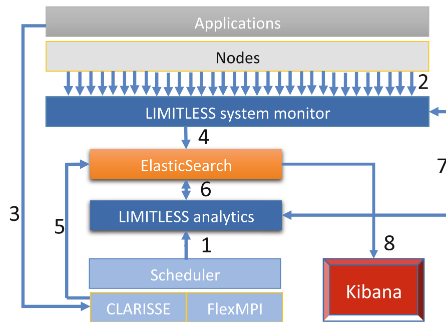



**Fig. 1.** General overview of the system architecture and interrelation with other components.

LIMITLESS is a light-weight scalable monitor that operates on each compute node and provides information about available system resources and the performance of the applications that are being executed. Figure 1 shows a general

overview of the LIMITLESS architecture. It is integrated with other components like the application scheduler, FlexMPI and CLARISSE runtimes to extend its capabilities, for example, including features for application-level monitoring. LIMITLESS includes four main modules: a *System monitor* that collects the performance metrics from the cluster, an *ElasticSearch* database [6] that provides persistent storage, Kibana, a *GUI* for displaying the cluster information in a user-friendly format, and an *Analytic* component that analyses and models the executing applications, and generates proxies.

LIMITLESS Analytics (LAN) is the component that deals with the storage, visualization, communication with the scheduler and is responsible of the application performance prediction. It stores and manages the application models, generates the predictors, trains and executes the machine learning algorithms, and it generates the malleable proxies.

In order to explain the system dataflow, the arrows in Fig. 1 include numbers. When one application is executed, the scheduler notifies LIMITLESS Analytics (arrow 1) about the application characteristics (which is used to identify and classify the application). After that, when the applications are executed two different metrics are collected simultaneously: at node level to the monitor (arrow 2) and application level to FlexMPI and CLARISSE (arrow 3). Then, both metrics are processed by the respective runtimes and are written into Elastic search (arrows 4 and 5). Then, the LIMITLESS analytics creates an application model using the information stored in ElasticSearch (arrow 6). Once the application model is generated, the analytics also creates the proxy associated with the application, which can be used to generate more performance metrics to increase the size of the dataset. Then, the predictors are refined using this offline information. And finally, the prediction model (arrow 7) is sent to LIMITLESS to predict the performance of the applications on each node. During all these processes, Kibana may be used to visualize (arrow 8) the cluster status.

### 2.1 System Monitor

The LIMITLESS monitor is designed to provide performance information of the nodes and applications in large scale systems. LIMITLESS allows to change the the monitoring period (also called *sample interval*) online, having one different for each node, and without the necessity of restarting the system or the monitor. The monitoring interval can be set in a range of time from hours to seconds and also sub-second. Moreover, the overhead in the compute nodes is low ($<1\%$ in CPU consumption and a memory footprint of 3890 KB in resident), which means that the monitoring does not interfere with the applications.

The system monitor consists of one *LIMITLESS Daemon Monitor* (LDM) per node, which periodically collects the performance metrics; a set of *LIMITLESS DaeMon Aggregators* (LDAs), that forwards the information from the LDMs to other aggregators or servers; and the *LIMITLESS DaeMon Server* (LDS) that gathers and stores the monitoring information in ElasticSearch. The deployment over the architecture is done hierarchically, generating a data flow from the nodes (LDMs) to the main server (LDS) and the database. The user

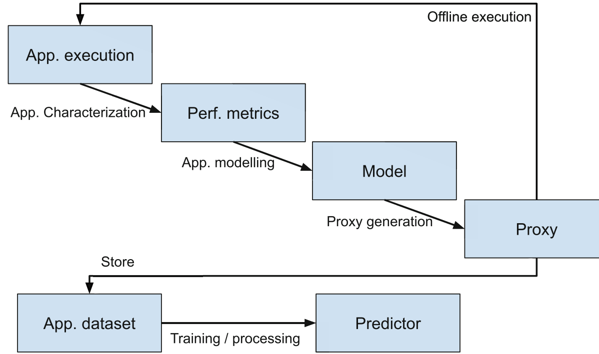



**Fig. 2.** Designed methodology to create the proxy benchmarks based on the monitoring data, and how that new data is used to produce more accurate predictors.

defines the hierarchy, but the optimal design consists of mapping the hierarchy with the network topology.

## 3 Building Synthetic Micro-benchmarks

The proposed proxy generation process can be seen in Fig. 2. The first step is the application characterization. This process consists of collecting the performance metrics associated with the running applications. The LIMITLESS Monitor is in charge of providing this collection of metrics. The execution time is also obtained from the scheduler. The second step consists of storing the performance metrics associated with each application in the database. Then, the analytic component generates a model of each application based on the collected performance metrics. Finally, the last step is to generate the proxy based on that model, resulting in an executable that tries to reproduce the same performance metrics as the original application.

The LIMITLESS Analytic component uses the performance models to generate the proxy benchmark. We extend the previous works by a new lightweight proxy generation that do not contain proprietary information, can be shared without any issue, do not need input data, and it is malleable. However, there are also some weaknesses. Note that the resulting proxies are not as accurate as other proposals due to the general-purpose source data.

The LIMITLESS' scheduling policies have been designed for clusters that use shared nodes. They can be applied to clusters with exclusive resource allocations, but they have no potential for improvement. There are three strategies to schedule the applications: the first one is based on monitoring information, the second one is based on prediction, and the last one is based on proxies utilization. The first alternative was implemented in [2] and uses the monitoring information to make decisions about the application schedule depending on the available resources, the performance of the running applications and other user-defined metrics. The second alternative uses the generated application models to

predict the future performance of the applications to make decisions about application scheduling in advance, which is a process that does not consume resources or CPU. However, this alternative depends on the accuracy of the predictors. The third strategy consists of using the proxies to combine their executions with other applications to identify pairs of applications that can run concurrently in the same compute node (to leverage the unused resources). The concepts of the second and third strategies are explained below.

In order to have a large dataset for the training and test phases, the framework executes the proxies multiple times until the accuracy of the prediction algorithms enhanced until 85%. Typically, during our tests, this value is achieved when the applications have been executed three times. However, the proxies are not as much accurate as the original application, which means that the training with proxies needs more executions. During our tests, we achieved that accuracy with 10 executions. Instead, LIMITLESS uses the compute-nodes to execute the proxies when there are free computational resources, and the scheduler does not have tasks ready to be run.

### 3.1 Application for Improving Machine Learning Algorithms

Deep-learning networks perform automatic feature extraction from the datasets independently. Most traditional machine-learning algorithms need to analyze large amounts of data in order to provide accurate predictions, and those datasets has to be large and representative enough of the features that the users want to extract.

The feature extraction process can take a long time to accomplish using statistical analysis by hand. Besides, there is no applications for generating well datasets for training, validating and processing. However, the more data a net can train on, the more accurate it is likely to be. So, the fact of having large datasets with representative data for each feature is directly related to the accuracy improvement.

Following this idea, LIMITLESS uses the proxy generator to produce new synthetic micro-benchmarks to execute at non critical hours and generate more data. Each execution of an application proxy is stored as a model of the original application, increasing the dataset for that application. Then, this dataset is used by the prediction algorithms to predict the performance of the running applications. Predicting the performance behaviour permits the scheduler to improve its policies, making decisions based on possible future scenarios. This proposal is the continuation of a previous work [2] and [3], in which LIMITLESS uses monitoring information to schedule the jobs dynamically. The proxies allow the scheduler to predict possible future states of the nodes and applications. Currently, this information is used to perform dynamic application scheduling based on predicting the future states of the cluster.

Note that this scheduling strategy can be used when an application has run one time because LIMITLESS generates the proxy. Then, the proxy model is used to train and predict. This process substitutes the human action of executing the

applications by hand, saving time until the users re-run their applications, and without consuming resources.

### 3.2 Application for Application Interference Analysis

One of the main objectives to generate these proxies based on application monitoring is the interference evaluation when two applications are running in the same node (note that one application could use more than one node with a different number of processes). This situation can occur when the scheduler allocates the jobs in non-exclusive nodes in order to perform a better utilization of the resources. With this configuration, depending on the available resources of a certain compute node, another application can share the unused ones. However, there is a potential risk of performance degradation (interference) between them. To improve the scheduling task, we propose the use of our malleable proxies to generate a *profiling study* under different workloads while a real application is running in the system.

To know if there is interference between two jobs (two applications, an application and a proxy, or two proxies), the system collects the performance metrics of the applications at the beginning, during a short period of time when the application is running exclusively in the node. Then, when another application is allocated in the same compute node the same performance counters are collected. By comparing the *exclusively-collected* and the *shared-collected* metrics, the system can identify if there is performance degradation. Using this information, the scheduler can make decisions about the scheduling, for instance migrating one of them to avoid the interference, or evaluating if that interference is mitigated when the number of processes is increased or decreased.

The collected information results in three performance counters: `RTIME` indicates the CPU time per group of iterations, `CTIME` is the communication time per group of iterations, and finally the execution time `TIME`. The execution time indicates if there is generic interference between two applications: if the execution time of an application is lower than the obtained when another application is running in the same node, it means that the second application is interfering with the first one. However, using the other counters, the system can identify more details about the reasons behind the performance degradation. This information contributes to making decisions to avoid it. For instance, if the interference is produced at CPU-level, the second application could reduce its processes to mitigate it, or the scheduler could move the second application to another compute node. FlexMPI performs these operations of expanding, reducing and migrating. It allows the application to increase or decrease its processes and redistribute the data every reconfiguration. Note that there is no necessity to kill the job and restart it with the new configuration. In the case of communication-based interference, the solution could be the reduction of the number of processes to reduce the communication between them.

Currently, the malleable proxies are relevant because, during the execution of the real application, the proxies can be run with different configurations: the scheduler, employing FlexMPI, can reconfigure a proxy from 2 processes to $n$ to

collect information about the performance and the interference. The objective is to evaluate different configurations to generate a scalability model that could support the scheduler with the scheduling making decisions. Note that, different from the last strategy, this one consumes computational resources due to the concurrent execution of the proxy and the applications. However, the interference evaluation is done with only one execution of the original application and one execution with the proxy to evaluate the different scenarios, which is faster than executing the original application and every configuration of the proxy.

## 4 Evaluation

We have implemented a proof-of-concept of application proxy generation based on CPU, memory and communication usage. We do not use profilers or perform reverse engineering like other related works.

The evaluation has been divided into three sections. The first one shows a comparison between the original benchmark and the proxy based on it. The original benchmarks used come from the Princeton Application Repository for Shared-Memory Computers (PARSEC) [14], which is a benchmark suite composed of multi-threaded programs that are focused on emerging workloads, and NASA Advance Supercomputing (NAS) Parallel Benchmarks (NPB) [11], which consists of a small set of applications designed to evaluate the performance of parallel supercomputers. The second section consists of a brief evaluation of the predictors when LIMITLESS uses the proxies to train the algorithms instead of the original applications. Finally, the third section corresponds to the evaluation of the interference produced between applications and proxies.

The evaluation has been done in a physical platform that consists of eight compute nodes. One partition of the cluster contains six nodes with Intel(R) Xeon(R) E7 with 12 cores and 128 GB of RAM in the other. The second partition contains two nodes with Intel(R) Xeon(R) Gold 6212U CPU @ 2.40 GHz with 24 cores and 315 GB of RAM. The connection between nodes is a 10 Gbps Ethernet. The I/O is based on Gluster parallel file system.

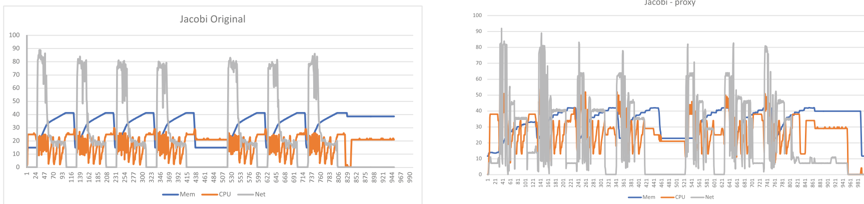
### 4.1 Proxy Accuracy

The different benchmarks used for this evaluation includes, as we have indicated before, a set of applications from PARSEC, NPB and the Jacobi method. The used benchmarks are:

- Jacobi: This is an algorithm for determining the solutions of a diagonally dominant system of linear equations. Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges.
- Integer-Sort: This is a kernel that performs random memory access. It belongs to the class of bucket sort algorithms which perform an all-to-all communication pattern (through OpenMP in this case).

- Multi-grid: This benchmark solves a 3D Poisson equation using a V-cycle multigrid method. It exhibits structured, long range communications.
- Bodytrack: This application tracks a human body with multiple cameras through an image sequence.
- Blackscholes: This application calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE). There is no closed-form expression for the Black-Scholes equation and as such it must be computed numerically [13].

Figure 3a shows the performance behavior of the Jacobi method. It exhibits characteristic CPU, memory and communication patterns. The CPU phases are correlated to the memory and the communication phases. Once LIMITLESS has modelled the application, the LAN component generates its proxy, which produces the performance behavior that can be seen in Fig. 3b.




(a) Original 1. Jacobi I/O (JIO) executing with six parallel threads.

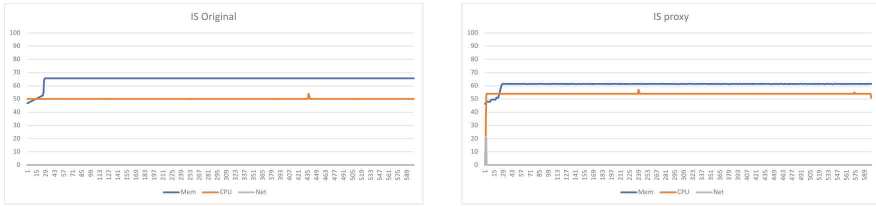
(b) Proxy 1. Jacobi I/O (JIO) proxy executing with six parallel threads.

**Fig. 3.** Jacobi I/O model. The X-axis represents the time in seconds while the Y-axis represents the usage percentage.

The next two figures (Figs. 4a and 4b show the performance behavior of Integer-sort (IS) and Multi-grid (MG) benchmarks from the NAS Parallel Benchmarks. Figure 4a shows the performance behavior of the Integer-sort benchmark. In this case, the execution performs a series of computations, including a gradual increase in memory usage until the data load is complete (the first 30 s of the execution). Figure 4b corresponds to the performance behavior obtained from the proxy execution. In this case, the CPU usage is a bit higher as the original due to the overhead of the memory replication.

Figure 5a shows the performance behavior of the Multi-Grid benchmark. This use case is similar to the last one (and similar to the rest of the benchmarks of the NPB 1). MG also performs a series of computations keeping the CPU and the memory barely constant along the execution time. Figure 5b corresponds to the performance behaviour obtained from the proxy execution, which is reproduced with high fidelity.

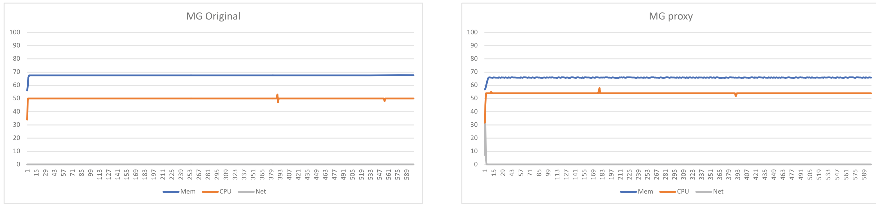
Figures 6a and 6b show the performance behavior of Bodytrack and Blackscholes benchmarks from the PARSEC Benchmarks. The first one corresponds




(a) Original 2. Integer-Sort (IS) executing with 12 parallel threads.

(b) Proxy 2. Integer-Sort (IS) proxy executing with 12 parallel threads.

**Fig. 4.** Integer-Sort model. The X-axis represents the time in seconds while the Y-axis represents the usage percentage.




(a) Original 3. Multi-Grid (MG) executing with 12 parallel threads.

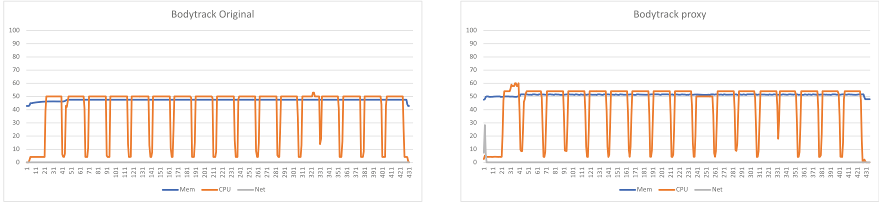
(b) Proxy 3. Multi-Grid (MG) proxy executing with 12 parallel threads.

**Fig. 5.** Multi-Grid model. The X-axis represents the time in seconds while the Y-axis represents the usage percentage.

to a computer vision workload, which performs medium working sets of computation. The second one is the simplest of all PARSEC workloads because it performs small working sets with no communication until execution end. The first one shows the performance behavior of Bodytrack benchmark. It consists of seventeen computation phases that are well replicated by the proxy in the second figure. There are no significant changes in the memory consumption, and it keep constant along the time.

The performance of the last original use case can be seen in Fig. 7a, which corresponds to Blackscholes benchmark. It performs an unique computation (but longer) phase at the middle of the executions. At the end of the execution there is a peak of communication. The performance metrics obtained from the proxy execution can be seen in Fig. 7b.

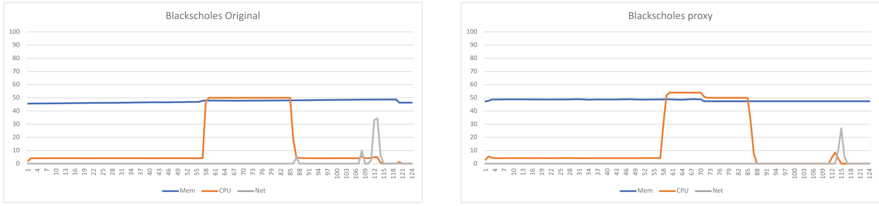
As it can be seen, in all the cases the proxy program is able to reproduce the original workload, despite the fact of existing small differences between the original program and the proxy. It is due to LIMITLESS does not perform deep profiling of the applications to produce 100% accurate proxies. Instead, LIMITLESS tries to build generic proxies that reproduce, with certain accuracy, the performance behaviour, the computation phases, the memory consumption and the network traffic. Taking all into account, LIMITLESS uses these proxies offline to generate new data to refine faster the performance predictors. It is important




(a) Original 4. Bodytrack (BT) executing with 12 parallel threads.

(b) Proxy 4. Bodytrack (BT) proxy executing with 12 parallel threads.

**Fig. 6.** Bodytrack model. The X-axis represents the time in seconds while the Y-axis represents the usage percentage.




(a) Original 5. Blackscholes (BS) executing with 12 parallel threads.

(b) Proxy 5. Blackscholes (BS) proxy executing with 12 parallel threads.

**Fig. 7.** Blackscholes model. The X-axis represents the time in seconds while the Y-axis represents the usage percentage.

to highlight that the predictors are re-built every time new model is stored in the LIMITLESS database, so the proxy programs improve their accuracy over the time.

## 4.2 Prediction Algorithms Improvement

LIMITLESS includes one analysis method to predict the performance of the applications. It is based on multi-variable analysis, and uses a federation of machine learning algorithms: Nearest Neighbour (NN), AdaBoost and Support Vector Machines (SVM). The purpose of having this prediction feature is to improve the application scheduling by means of evaluating possible future states of the system. If the scheduler needs to schedule an application *App*, it can predict the complete behaviour of *App*, select better nodes to run *App*, or decide if any of the current running applications could share a node with *App*.

Regarding the accuracy by using proxies, Table 1 shows the accuracy of the predictors using the five use cases previously described. Only the first execution stores real data in the dataset. Hence, another 20 execution patterns are stored using proxies. At first, it is important to know that these machine learning algorithms showed an average accuracy of 97% by using real executions and patterns. As it can be seen in Table 1, the average accuracy for all these use

cases is 87.5% (77.6% for memory patterns and 97.4% for CPU patterns). Note that these values include a tolerance of 3% (if both, the original and predicted values, have a difference within this range, we consider the prediction a hit). CPU is better predicted because generating CPU loads is easier than other factors. However, memory, I/O, and communications are harder to replicate without using the same code structure and operations, as [8] and [17] suggest. In our case, we do not try to consider the memory pattern, the execution flow, the system calls used, etc., which should improve the accuracy. Instead, we try to generate generic algorithms to provide similar proxies to replicate the performance of the original applications.

**Table 1.** Accuracy of the machine learning algorithms using datasets without real application executions, taking into account a tolerance of +/–3%. Note that the first execution is provided by the real application (first execution in the system), and then it is used as a model for generating the proxies.

| Application | Memory | CPU |
|---|---|---|
| BS | 55% | 97% |
| BT | 92% | 98% |
| IS | 50% | 99% |
| MG | 99% | 99% |
| JIO | 92% | 94% |

### 4.3 Interference Detection Using Malleable Proxies

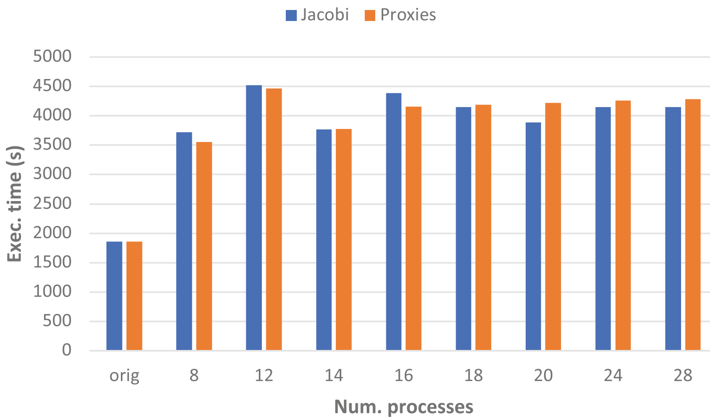



**Fig. 8.** Use case Jacobi - comparison between the execution time of Jacobi instances running concurrently in the same compute node and the same executions using proxies. The interference is directly related to the execution time.

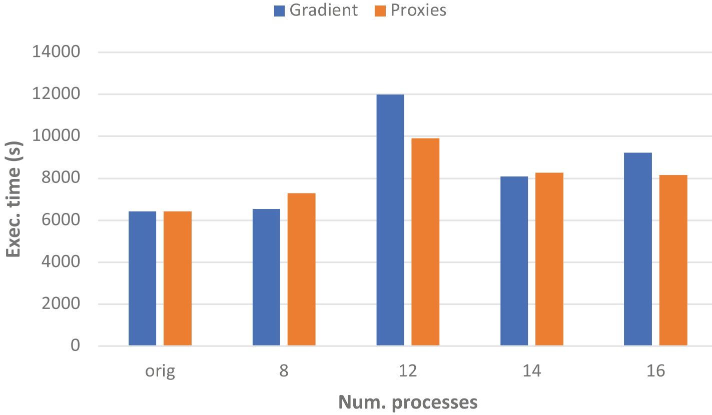



**Fig. 9.** Use case Gradient - comparison between the execution time of Conjugate Gradient instances running concurrently in the same compute node and the same executions using proxies. The interference is directly related to the execution time.

In this section we show two examples of how the system studies the interference between two applications using the malleable proxies. With the information related to the interference analysis, the scheduler is able to make more precise decisions for future executions, knowing which applications are compatible (i.e. both can share a node without performance degradation). The following tests have been focused on the second partition of the cluster. Note that the applications can be run in more compute nodes, however only the last allocated node could have available resources to share with other applications. For example, *App1* requires 28 processes and *App2* 20 in a cluster with two nodes with 24 cores each. The scheduler will allocate 24 processes in the first node and the other 4 processes in the second node for *App1*. As there are enough free resources in the second compute node, the scheduler can allocate *App2* on them.

Following this idea, the first use case corresponds to the Jacobi I/O application. The second use case corresponds to the Conjugate Gradient algorithm. Figure 8 shows the execution time of the Jacobi use case under different conditions. The experiments start with the execution time of the Jacobi application running in an exclusive node. Then, each experiment corresponds to a Jacobi instance with 12 processes (in blue) combined with another instance with processes from 8 to 28 (in orange). The objective is to quantify the interference under different scenarios. As it can be seen, the interference reaches the maximum value with 12 processes per application because all the cores are in use, and both instances are performing the same operations.

The same scenario is proposed in Fig. 9 with the Conjugate Gradient use case. In this scenario, the evaluation is done between 8 and 16 processes due to the size of the problem. There is no possibility to increase the processes because of the data redistribution. The same behaviour can be observed in the 12-processes experiment. Before and after this, the interference is lower because the interfere

operations overlap in a lower percentage in the time because the load changes on each experiment (load distribution between processes).

The last scenarios have been done statically, with the proxies previously configured with the concrete number of processes. However, due to the malleability, the same experiments can be done in a row, executing one instance of Jacobi or Gradient, and a malleable proxy that increases its number of processes periodically. The results of these experiments can be seen in Tables 2 and 3. They show the overhead of performing the interference study using malleability. Note that each experiment takes the time per iteration instead of the execution time (which multiplied by the number of iterations results in the estimated execution time). Taking into account the overheads, the difference between the static and the malleable evaluation is the time needed to get the results: *32,734* s for the first use case and *44,868* s for the second one with the static model. In the case of the malleable mode, the time needed is *3850* s for the first one, and *6003* s for the second one. Note that the executions start with 8 processes. Malleability generates overheads for process creation/destruction, but it is compensated by the time saved when the application is running with more processes.

**Table 2.** Jacobi use case - Interference evaluation using malleability with one proxy, from 8 processes to 28.

| N. procs | Expand/shrink time (s) | Data redistribution |
|---|---|---|
| 8 to 12 | 0.891120 | 0.121103 |
| 12 to 14 | 0.949042 | 0.116333 |
| 14 to 16 | 0.898925 | 0.114143 |
| 16 to 18 | 0.901130 | 0.120974 |
| 18 to 20 | 0.908145 | 0.114886 |
| 20 to 24 | 0.909972 | 0.104716 |
| 24 to 28 | 0.907456 | 0.104094 |
| 28 to 8 | 0.017645 | 0.131265 |
| **Total overhead** | **6.383435** | **0.927514** |

**Table 3.** Gradient use case - Overhead using malleability with one proxy, from 8 processes to 16.

| N. procs | Expand/shrink time | Data redistribution time |
|---|---|---|
| 8 to 12 | 0.087823 | 0.125488 |
| 12 to 14 | 0.088597 | 0.105049 |
| 14 to 16 | 0.935568 | 0.115070 |
| 16 to 8 | 0.002273 | 0.158980 |
| Total overhead | **1.111978 s.** | **0.5048587 s.** |

## 5 Related Work

In this section we introduce some related works that are relevant in fields of monitoring, application proxy generation and scheduling. Unfortunately, it does not contains any related work about malleable proxies, because, as far as we know, our proposal is a novelty.

The main goal of the authors in [9] is to provide easy-to-use, portable, transparent, and efficient instrumentation tools (called Pintools) that are written in C/C++ using Pin's rich API. They provide different instrumentation than other similar tools, for example, Valgrind and dynamoRIO. The instrumentation does not interfere with the loads/stores in the registers. The authors provide a comparison between PIN, Valgrind and dinamoRIO, where we can observe that PIN and dinamoRIO outperform Valgrind without instrumentation, and PIN outperforms both Valgrind and dinamoRIO when we consider performance with instrumentation.

In [4] the authors proposed a synthetic proxy generation to (1) reduce the simulation time employing these proxies generated instead of the original applications, and (2) share these proxies for computer architects, as some of the specific target applications are proprietary, and vendors hesitate to share them. They provided the synthetic clones for CPU2006 and ImplanBench workloads. The metrics used include the Memory Level Parallelism (MLP) of those workloads to estimate the burstiness of accesses to the main memory, and the features needed to characterize a benchmark are: a Statistical Flow Graph (SFG) that is used to capture the control flow behaviour; a branch prediction algorithm based on the branch transition rate; the Instruction Level Parallelism (ILP) in the workload; and the memory access pattern. Instead of capturing data to get the memory access pattern, the authors used a stride base memory access (because Joshi et al. concluded previously in [7] that most of the load and store instructions in CPU200 workloads have that pattern).

Later, in [5], the authors proposed a framework that can generate proxies for real-world multi-threaded applications based on: shared caches, coherence logic, out-of-order cores, interconnection network and DRAM. This framework is evaluated by generating proxies from the PARSEC benchmark suite and comparing their results in terms of performance. Their solution consists of extracting performance information from the applications and then generating the code for the proxies based on a C template with some options. The main benefit of creating and using these proxies is that they have used a simulator to calculate the energy consumption of different workloads and different parameters, and the simulations are four magnitude orders faster using the proxies due to the number of instructions executed (millions versus thousands of instructions).

In [16] the authors proposed code mutation (to generate application proxies), a technique that mutates the original code of a real application to make harder/impossible reverse engineering. Their objective consists of allowing the distribution of those proxies that have the same behaviour (in terms of performance) as the original. To test the results, they use the SPEC CPU2000 and MiBench benchmarks. They also provide a comparison between the different

related works for code mutation, because the approaches differ in the way to preserve the proprietary application's memory accesses and control flow behaviour. The code mutation uses the same code structure by maintaining the execution flow but using different instructions, operations and registers. In these cases, the mutant code has the same execution time as the original application, which is one of the main objectives.

In [15] the same authors as the last related work propose a framework to generate synthetic benchmarks based on real applications. To do that, the framework performs different profiling analyses, similar to reverse engineering. Based on that set of instructions, data and code information, the framework generates an application in a high-level programming language (C) that fulfils the performance requirements. Finally, the framework performs a semi-random obfuscation for avoiding the possibility of generating similar code as the original application, but with smaller number of instructions.

Clone morphing [17] is different from Clone workloads. The second one tries to copy in an application its performance behaviour, without providing real information about the original application. The first one proposes systematic changes to clone the behaviour of the application focusing on certain features. In this case, this program copies the cache/memory patterns for each application. Their main contribution is the systematic method for producing new proxies with performance behaviours that are the result of the combination of more than one application. The main weakness of this work is that the authors focus their work on the cache and memory patterns.

In [12] the authors proposed PerfProx, another alternative to build proxies based on real applications. However, this related work differs from other previous works because their proxy generator tries to replicate the performance of the applications based on the performance counters (similar to our proposal), and it is only focused on database processes. PerfProx directly genetares a general-purpose proxy executable. They have evaluated their proposal on Casandra, MongoDB and MySQL running both the data-serving and data-analysis on different platforms.

In [1] the authors proposed SynFull, a synthetic traffic generator that captures both applications and cache coherence behaviour to evaluate NoCs (Networks on chips). SynFull provides a novel technique for modelling real application traffic without the need for expensive, detailed simulation of all levels of the system. The authors determined the key traffic attributes that a cache-coherent application-driven traffic model must capture, including coherence-based message dependencies, application phase behaviour and injection process. So, this work is focused on modelling the network and the cache coherence traffic. As a result, SynFull attains an overall accuracy of 10.5% across the three configurations for all benchmarks relative to full-system simulation.

In contrast to the previous related works, the objective of this proposal is not to replicate the applications accurately. Instead, our goal is to reproduce the performance of the applications without extracting data from the binaries, without the necessity of dealing with code or data from the applications, and

without a big penalty in terms of overhead. However, this research line is relevant because more accurate proxies will produce more accurate performance counters, and they will increase the accuracy of the predictors. Moreover, the LIMITLESS' proxies are malleable using FlexMPI, which allows dynamic reconfiguration of the number of processes in run time. Due to this, the system is able to analyze different configurations of the same application proxy to discover its scalability and its impact on other applications (interference).

## 6 Conclusion

In this paper, we introduce a new feature on LIMITLESS, a lightweight monitoring and scheduling framework that was designed to monitor and schedule the execution of the applications on large-scale computing infrastructures. This feature consists of creating synthetic micro-benchmarks (proxies) from the applications executed in the cluster, based on the performance models that LIMITLESS already produces in an iterative fashion. One of the main characteristics of the framework is the performance prediction, which allows the scheduler to improve its tasks. It generates proxies based on the collected data and then uses those proxies to generate new execution data, which are included in the dataset to train the networks and the machine learning algorithms. With this proposal, LIMITLESS can predict the application performance with one execution and without user intervention. Besides, those application proxies can be shared to exhibit the performance of the real applications that have been running in the platforms because they do not contain proprietary information nor include any piece of code of the original application. Moreover, the system uses different application proxies to perform interference studies between applications, which allows the scheduler to share nodes between compatible applications. Note that, in case of performance degradation (interference) during the execution of real applications, the system will detect that situation, avoiding it by means of application migration or increasing or decreasing the processes using the malleability features.

As future work, we are studying the possibility of including a more precise application characterization to replicate, not only the performance metrics but the performance behaviour: FLOPs, IPC, syscalls, I/O phases, etc. Currently, it is not an option because we want to keep the overhead in the compute nodes as lower as possible.

## References

1. Badr, M., Jerger, N.E.: SynFull: synthetic traffic models capturing cache coherent behaviour. ACM SIGARCH Comput. Architect. News **42**(3), 109–120 (2014)
2. Cascajo, A., Singh, D.E., Carretero, J.: Performance-aware scheduling of parallel applications on non-dedicated clusters. Electronics **8**(9), 982 (2019)

3. Cascajo, A., Singh, D.E., Carretero, J.: Limitless - light-weight monitoring tool for large scale systems. In: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 220–227 (2021). https://doi.org/10.1109/PDP52278.2021.00042
4. Ganesan, K., Jo, J., John, L.K.: Synthesizing memory-level parallelism aware miniature clones for SPEC CPU2006 and implant bench workloads. In: ISPASS 2010 - IEEE International Symposium on Performance Analysis of Systems and Software, pp. 33–44 (2010)
5. Ganesan, K., John, L.K.: Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors. IEEE Trans. Comput. **63**, 833–846 (2014)
6. Gormley, C., Tong, Z.: Elasticsearch: the Definitive Guide: a Distributed Real-Time Search and Analytics Engine. O'Reilly Media, Inc. (2015)
7. Joshi, A., Bell, J., Ibm, R.H., John, L.K.: Distilling the essence of proprietary workloads into miniature benchmarks. TACO - ACM Trans. Archit. Code Optim. **5**(2), 1–33 (2008). https://doi.org/10.1145/1400112.1400115
8. Joshi, A., Eeckhout, L., Bell, R.H., John, L.: Performance cloning: a technique for disseminating proprietary applications as benchmarks. In: Proceedings of the 2006 IEEE International Symposium on Workload Characterization, IISWC - 2006, pp. 105–115 (2006)
9. Luk, C.K, et al.: Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Not. **40**(6), 190–200 (2005)
10. Martín, G., Marinescu, M.-C., Singh, D.E., Carretero, J.: FLEX-MPI: an MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 138–149. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40047-6_16
11. NASA Advanced Supercomputing (NAS) Division: NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html
12. Panda, R., John, L.K.: Proxy benchmarks for emerging big-data workloads. In: Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT 2017-September, pp. 105–116 (2017)
13. University, P.: PARSEC - CSWiki, http://wiki.cs.princeton.edu/index.php/PARSEC-Blackscholes
14. University, P.: The PARSEC Benchmark Suite. https://parsec.cs.princeton.edu/
15. Van Ertvelde, L., Eeckhout, L.: Benchmark synthesis for architecture and compiler exploration. In: IEEE International Symposium on Workload Characterization, IISWC 2010, pp. 1–11 (2010)
16. Van Ertvelde, L., Eeckhout, L.: Dispersing proprietary applications as benchmarks through code mutation. In: ACM SIGPLAN Notices, pp. 201–210 (2008)
17. Wang, Y., Awad, A., Solihin, Y.: Clone morphing: creating new workload behavior from existing applications. In: ISPASS 2017 - IEEE International Symposium on Performance Analysis of Systems and Software, pp. 97–108 (2017)