A tool for selecting the right target machine for Parallel Scientific Applications

Javier Panadero¹, Alvaro Wong, Dolores Rexachs, Emilio Luque∗

Computer Architecture and Operating System Department, Universitat Autonoma de Barcelona, Barcelona, SPAIN.

Abstract
Analyzing and predicting performance in parallel applications is a great challenge for scientific programmers due to its complexity. Analyzing parallel application behavior is not a trivial process and it requires spending a large amount of time and effort to understand the behavior of the application algorithms during execution. We have developed PAS2P toolkit from PAS2P methodology. This methodology strives to characterize the behavior of MPI applications to identify and extract representative phases and create a signature, which will be used to analyze the application behavior and predict its execution time in different target systems. Applying this methodology is a non-trivial process for users, for this reason we have developed the proposal toolkit, which allows users to make the whole process, from creating a signature to executing it on target systems, in user-space in an easy and fully automatic way. PAS2P toolkit has been validated, making clear the advantages of the signature, with its execution time being much lower than the whole application execution time (around 7% of the total execution time), with a high quality prediction of around 96%.

Keywords: Performance toolkit, Performance Prediction, Parallel Application Signature

1. Introduction
The continuous progress in parallel machines as well as in parallel programming models have allowed the development of scientific applications with more complexity and accuracy. These kinds of applications are very difficult to analyze, due to their high level of complexity, which centers on the difficulty to understand the behavior of the application during its execution in order to find and isolate performance problems. Moreover, parallel applications can be executed using different systems and different performance results can be obtained, according to the machine configuration. In performance prediction, the best way for measuring the performance is to use the application itself, but the time required to run it thoroughly is a heavy requirement; especially if we want to predict in different systems. In this sense, performance tools become necessary to determine the most suitable system on which to execute the application.

The Parallel Application Signature for Performance Prediction (PAS2P) methodology[1] strives to solve the complexity of the behavior analysis making an application signature, which represents the relevant behavior of a
message passing application. The signature is built by selecting a set of relevant parts (phases) and their weights, which are the frequency each phase repeats. When the signature is constructed, it can be executed on real target machines in order to analyze its behavior, as well as to predict the execution time with a high degree of accuracy.

We propose to make a toolkit in order to automate the PAS2P methodology to facilitate its usability, because to obtain and execute the signature from the methodology is a non-trivial process for the users. In this sense, we have developed a toolkit from PAS2P methodology named PAS2P toolkit, which automatically generates an application signature transparently in order to be executed in real target machines.

PAS2P toolkit can be employed by system administrators, users and scientific programmers. System administrators need foresight which allows them to make better use of the available resources, through mapping policies with the purpose of reducing economic and energy costs. On the other hand, users want to execute their applications as fast as possible in order to obtain their results, thus they need to know what system is more appropriate to execute their applications, whereas scientific programmers need to know the performance and the behavior of an application in order to optimize it and solve any performance problem. Due to the fact that the signature represents the application kernel, programmers can focus on analyzing the signature behavior, which contains the important pieces of code, with the aim of detecting application performance problems.

To develop the PAS2P toolkit, all stages of the PAS2P methodology have been implemented. The toolkit we propose is composed of four modules: Instrumentation, Analysis, Signature construction and Performance prediction. In the instrumentation module, we collect the data and characterize compute and communication behavior to generate an application trace log. The analysis module is developed to obtain the application model to identify and extract the most relevant phases and assign them a weight from the number of times they occur. For the Signature construction, we generate an executable signature, which contains the relevant phases and their weights. Finally, the Prediction module was developed, which executes the signature to predict the execution time in an automatic way.

To evaluate the correct implementation of the proposed PAS2P toolkit, a set of experiments were carried out extracting the signatures from benchmarks such as CG, BT and LU from NPB [2], Sweep3D [3] and SMG2000 [4], and the applications Moldy [5] and M.DynaMix [6]. The signatures generated with the PAS2P toolkit for all the applications tested have an average execution time that represents about 7% of the application execution time, and the prediction quality is about 93% accurate, ensuring the two objectives of PAS2P methodology.

This paper is organized as follows: Sections 2 and 3 present related work and the PAS2P methodology, respectively. Section 4 presents the PAS2P toolkit, Section 5 presents the analysis and discussion of the prediction results obtained, and finally, we present the conclusions and future work.

2. Related Work

There are other tools which are closely related to analysis and performance prediction. WARPP [7] is a prototype semi-automatic performance prediction simulator, which was designed to support the automated generation of performance models and it is composed of four steps: (1) model construction which requires significant work by the user, (2) machine benchmarking an instrumented version of the application, (3) post-execution analysis of machine benchmarking results to produce simulator inputs and finally (4) simulation. This approach differs from PAS2P toolkit because with our toolkit, users can generate and execute the application signature in a fully automatic way in order to obtain the performance prediction.

There are other approaches such as BSIM [8] and PHANTOM [9] which also use simulation in order to predict the performance. BSIM is a simulator that predicts the performances for a parallel application by the simulation of only the communications, the computing parts of the program are modeled separately. In PHANTOM, the performance estimation is based on the real execution of computational parts of all processes of the parallel application on a single machine of the platform. The resulted computing performances will be combined with the network performances by the use of MPI-SIM [10] in order to predict the parallel execution time. This approach differs of PAS2P toolkit since PAS2P generates an application signature which contains the most significant application parts (communication and computation) in order to be executed in real target machines.

SWAPP [11] is a framework for performance prediction based on computation and communication projections and using benchmark data in target systems. The framework assumes that the target system is not available for
running applications, only benchmark. Projections are developed using the performance profiles of the benchmarks and application on the base system and the benchmark data for the target system. SWAPP projects the performances of compute and communication components separately, then they are combined to get the full application projection. This approach differs from PAS2P toolkit since PAS2P does not use benchmark information to predict the application performance, instead the application signature is executed in the target system.

Snavely et al. [12] proposes a framework to construct application and machine signatures to simulate application behavior across different system or processor architectures. In order to project the MPI communication performance, the Dimemas simulator [13] is used. This approach differs of PAS2P toolkit, since our toolkit uses a real network in order to obtain the communication performance.

3. Overview about PAS2P methodology

Applications typically possess highly repetitive behavior and parallel applications are no exception. PAS2P makes an analysis to characterize the computational and communications-related behavior of parallel applications by identifying these repetitive portions. It is important to notice that this is a two-step methodology. The first step is to analyze the application, build the application model to extract its phases and weights, and use that information to build the signature, which is an executable that contains the relevant phases with instrumentation. This is in order to have information about their behavior and their weights to predict the application performance on the target machines. The second step is to execute the signature in a target system, to measure the execution time of each phase and predict the execution time of the application.

3.1. Application analysis and signature construction

In order to obtain the behavior of computation and communication, the application is instrumented on a base machine in order to intercept and collect communication events of the parallel application. With this collected data an application trace log is generated. The communication events are ordered by means of a logical global clock according to causality relations between communication events. The machine-independent application model can be obtained from this trace. Once we have the application model, the methodology strives to identify the application patterns in order to find a representative behavior of the application. It is processed using a technique that searches for similarity to identify and extract the most relevant event sequences (phases) and assign them a weight based on the number of times the phases occur. Afterwards, in order to construct the signature, the last step is to re-run the application to create the coordinated checkpoints before each relevant phase happens. Therefore, the executable signature will be defined by a set of relevant phases and their weights.

3.2. Performance prediction model

Once we have constructed the application signature, we can run it on real target machines to analyze the application behavior and predict the application execution time. In order to execute the phases, we restart the checkpoints of the application before the phase begins and measure its execution time until the phase ends. In order to predict the application execution time equation 1 is used, where PET is the Predicted application Execution Time, n is the number of phases, TPhase is the Phase i Execution Time and Wi is the weight of the phase i.

Due to the complexity of the process and the huge quantity of information obtained during the analysis, we decided to automatize the methodology, allowing users to apply the whole methodology in an automatic and transparent way. The next section explains how the methodology was automated.

\[ PET = \sum_{i=1}^{n} (TPhase_i)(Wi) \] (1)

4. PAS2P Toolkit

We have developed a toolkit from PAS2P methodology, named PAS2P toolkit, because obtaining and executing the signature from the methodology is a non-trivial process for the users. PAS2P toolkit allows users to generate and automatically execute an application signature transparently.
The PAS2P methodology has been grouped into four independent modules. The first three modules: Instrumentation, Analyzer and Signature generation are related to the first step of the PAS2P methodology, which was explained in section 3.1, while the fourth module, the Performance prediction module, is related to the second step of the methodology, which was explained in section 3.2. All four modules have been integrated in a toolkit. Figure 1 shows a chart representing each module and the stage of the methodology that is developed in it.

![Fig. 1. Modules of PAS2P toolkit](image1)

PAS2P toolkit is composed by two binaries, *Pas2p_command* and *Pas2p_tool*. These binaries have been developed in C++ in order to take advantage of the efficiency of C and the facilities of C++, as the Standard Template Library (STD). The binary *Pas2p_command* allows us to automate the whole process until the executable *signature* is created. Fig. 2 shows all the options of *Pas2p_command* and how users can use it. The binary *Pas2p_tool* analyzes the log trace to obtain the relevant phases and their weights. Besides these two binaries, the PAS2P toolkit has a dynamic library called *Libpas2p*, which allows us to automatize the methodology by instrumenting the application, constructing and executing the signature.

```
alvaro@clus1:~:/bin/NPB3.3-MPI/bin$ ./pas2p_command --help
./pas2p_command -lcf 'mprun application'
user command line between apostrophes 'mprun application'

*************PAS2P INSTRUMENTATION LOG TRACE PARAMETERS*************
-1 PAS2P LOG
-c Construct Signature using checkpoints (DMTCP)
-r Construct Signature without checkpoints

*************PAS2P SIGNATURE Generation PARAMETERS*************

alvaro@clus1:~:/bin/NPB3.3-MPI/bin$
```

**Fig. 2. *Pas2p_command* user interface**

4.1. **Instrumentation module**

With the purpose of instrumenting the parallel message application, we generate an optimized trace log containing necessary information such as communication events and computational time. To do this transparently, the dynamic library named *Libpas2p* was developed and located between the parallel application and the MPI library. Its objective is to intercept and collect MPI communications primitives during the application execution time by interposition functions, without modifying the application source code. These primitives will be saved as communication events and computational time to build a trace log. In order to obtain the trace log, the binary *Pas2p_command* is used with the input parameter "-l " which loads the *Libpas2p* in user space automatically through the environmental variable *LD_PRELOAD*, as is shown in Fig. 3(a). Once the application has been executed, the trace log is obtained. As is shown in Fig. 3(b), the trace log contains information about the application process, on column 1 (process), the MPI event, on column 2 (Call), the source/destination of the process involved with, on column 3 (Send/Recv), the count events per process, on column 4 (Count), the communication volume
in bytes, on column 5 (Byte), the wall clock time in nanoseconds, on column 6 (Wtime), the computational time in nanoseconds, on column 7 (Compute Time), which is defined by the cpu time between two MPI events, and finally the type of the event (send: 1, recv: 0, collective: -1). This trace log will be used in the next module in order to construct the application model.

4.2. Analyzer module

In order to extract the relevant phases of the application and their weights, the analyzer module is used. This module is composed of the binary Pas2p_tool, which is executed automatically by Pas2p_command when the trace log is obtained, as is shown in Fig. 4(a).

![Pas2p_command loads the Libpas2p library to obtain the application trace log; (b) Trace log generated by Libpas2p library.](image)

**Fig. 3.** (a) Pas2p_command loads the Libpas2p library to obtain the application trace log; (b) Trace log generated by Libpas2p library.

**Pas2p_tool** has as input the application trace log created in the instrumentation module, as well as two user parameters: computation and communication similarity, as is shown in Fig. 4(b). The computation parameter compares the computational time between two phases, where a value of 100% represents the equality between the phases. As for the communication parameter, Pas2p_tool compares which type of communication there is and the number of similar communication events. The user can configure these two parameters, modifying their range of similarity. These ranges of similarity determine the total number of application phases.

The parser of the analyzer module reads the trace log, where all events are collected into an event vector, which will be analysed later to obtain the parallel application model. Once the event vector has been filled, it needs to relate the events that belong to the same message. This id relation is done using bit fields. If two events contain the same id relation, then they will be related to each other, thus relation being the source, destination and/or size of each event.
To obtain the application model, we first have to assign the Logical Time (LT). To assign Logical Times to each event, the algorithm proposed in the PAS2P methodology is used. This algorithm was improved by eliminating searches, creating structures that allow pointing the memory position where events are aiming to decrease the time to assign logical time to each event. Once the LT of the events are assigned, a new vector is created, named Logical Model, with a specific size, defined by the number of processes per the maximum LT in which all events are inserted in a position depending on their LT. Once the events are inserted, the index of the vector itself is used as Logical Clock (Tick).

Taking the algorithm proposed in the methodology, the Logical Trace vector goes forward to generate phases and compare them. In order to obtain the phases, we generate another vector called Phases vector, which contains the following information:

- **Communication Pattern:** The processes (source / destination) involved in the phase.
- **Communication Volume:** The volume of communication sent or received in the phase.
- **Startpoint:** The tick where the phase begins
- **Endpoint:** The tick where the phase ends
- **Phase Execution Time:** The computational time from the startpoint to the endpoint.
- **Phase Weight:** Number of times that the phase repeats

To compare the phases to decide if two phase are similar, Pas2p tool considers the next fields of the Phases vector: communication pattern, communication volume and phase execution time. If these three fields are similar, the phases will be the same and the phase weight field increases one unit.

Once the phases have been obtained, the next step is to select the relevant phases. These phases are selected by their percentage of representativeness. Pas2p tool generates three ranges of predictions (85-90%, 90-95%, 95-100%) selecting different phases, as is shown in Fig. 5(a). The prediction error will be given by the number of phases, a higher number of phases implies less error, but a higher signature execution time. Users can select any of these prediction ranges, by default Pas2p tool selects the highest range of prediction (95%-100%).

Finally, the relevant phases and their weights are saved in a phase table. Fig. 5(b) shows an example of this table, which was obtained from the execution of an application with 4 processes and contains the startpoint and endpoint of each phase that will be used to measure its execution time. Each row of the table represents a phase, whose startpoint and endpoint are defined by the number sent where the phase occurs. The last two columns show the Phase ID and the weights of the phase. After the phase table is generated, Pas2p tool indicates if it is necessary to construct the signature making checkpoints or not depending on where the relevant phases are located in the application.
4.3. Signature generation module

Once Pas2p_tool has obtained the relevant phases and their weights, users can use this module to generate the executable signature.

The first step to construct the signature is instrument the application, due to the interaction between the application with external libraries. If the signature has to be constructed with checkpoints, we have to guarantee a correct warm-up time for the machine components. Another issue is how to detect relevant phases during the application execution. Due to the complexity of this step, we add the process of signature generation to the Libpas2p library to make it transparent to users. To construct the signature, the binary $\text{Pas2p\_command}$ is used, which re-runs the application loading the Libpas2p library and the phase table to instrument and detect where the phases occur. Once the signature has been constructed, an executable named signature is generated in order to predict the performance prediction. Now we describe both processes.

- **Signature with checkpoints.** To create the signature with checkpoints the binary $\text{Pas2p\_command}$ is used with the input parameter “-c”. The checkpoint operation is done before the Startpoint of the phase to guarantee the warm-up. Previous versions of the first PAS2P implementation used the BLCR library[14], but it requires being installed at the kernel level, therefore we changed to the DMTCP [15] checkpointing library, because of its user-level transparency. As shown in Fig. 6(a), the Libpas2p detects a phase and calls the DMTCP library to make the checkpoint. Once it is done, the DMTCP returns the control to the Libpas2p to add the checkpoint to the signature. This process is repeated until the last relevant phase. Finally, the executable signature is generated, with the information needed to measure each phase. Fig. 6(b) shows an example of signature generation using checkpoints. We know that the size of the checkpoint could be a big problem, depending of the application memory requirements. In order to reduce the checkpoints size, the DMTCP library has an option to compress the size of checkpoint, which the module achieves reducing the checkpoint size significantly. Moreover, in the cases that the relevant phases are at the beginning of the application, the signature can be constructed avoiding the use of checkpoint.

- **Signature without checkpoints.** In the case that the relevant phases appear at the beginning of the application and they are repeated throughout all the execution, it is possible to avoid checkpoints to construct the signature. In that case, users can use $\text{Pas2p\_command}$ with the input parameter “-r”. It is important to remark that although only a relevant phase appears distant at the beginning of the application, it would be necessary to do checkpoints in order to avoid executing the application until this relevant phase arrives during the performance prediction stage. The signature is generated with the phase table loaded with the Libpas2p obtained during the analysis module to detect where each phase begins and ends.

![Fig. 6. (a) Libpas2p calls to DMTCP API to make checkpoint; (b) Pas2p\_command re-run the application to construct the signature.](image-url)
4.4. Performance prediction module

Once the **signature** has been constructed, users can run the executable **signature** on target machines. It is important to remark that in order to predict the application execution time, the **signature** will execute and measure each phase. The execution time obtained in each phase will be multiplied by the weight of its phase, as shown in equation 1 in section 3.2. It is for this reason that the **signature** is composed of the phases and their weights. Besides, indistinctively from the system where it is executed, users can execute the executable **signature** using different mapping policies. In that case, users can use a machine file if they want to predict the performance by changing the mapping policies of the processes on different nodes or cores.

![Image of diagram]

Fig. 7. (a) **Libpas2p** restarts checkpoints; (b) Execute the **signature** in order to predict.

- **Execution with checkpoints.** The checkpoints of the **signature** are restarted from the saved states and start measuring from the point where a phase begins until its end. In order to detect a phase the library **Libpas2p** is used. When a phase is measured, the **signature** finalizes its execution and restarts the next checkpoint using the DMTCP library. This method and process is repeated for all phases, as is shown in Fig. 7(a). The PAS2P toolkit uses a non-blocking method once the phase ends to finalize the checkpoint execution. This is done generating messages where all processes inform to process 1 that they have finished. When process 1 has been measured, a MPI testall is carried out to check if all processes have sent the message to finalize the execution. Finally, the **signature** applies the equation 1 in order to predict the execution time. Fig. 7(b) shows the **signature** execution output where the performance prediction using checkpoints can be observed.

- **Execution without checkpoints.** In this case, the **signature** is built without checkpoints. This is because the relevant phases are located at the beginning of the application. The **signature** runs the parallel application from the beginning and is measured from where a phase begins until its end. When the last phase has been measured, the **signature** finalizes the application and applies the equation 1 in order to predict the execution time.

5. Experimental results

With the objective to demonstrate the operation of PAS2P toolkit, we show some experimental results on different clusters used as base and target machines (see Table 1). We present the results obtained for the following parallel programs: CG, BT and SP from the NPB using class D, Sweep3D (workload sweep 150), SMG2000 (workload -n 250 -solver 3 - iterations 1200), Moldy (Workload control.tip4p with nsteps=100000) and M.DynaMix (iterations=150 and 2000 molecules H2O). All the applications have been compiled using 256 processes.
Table 1. Cluster characteristics

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Cluster</td>
<td>Dual-Core Intel(R) Xeon(R) CPU 5160 2.66GHz 4MB L2 (2x2), 12 GB Fully</td>
</tr>
<tr>
<td>(Target Machine)</td>
<td>Buffered DIMM 667 MHz, Interconnection Gigabit Ethernet, 128 cores.</td>
</tr>
<tr>
<td>Nova Cluster</td>
<td>4 Intel Xeon quad-core E7350 2.66Ghz Tigerton Processors L2 cache = 2x4 MB</td>
</tr>
<tr>
<td>(Base Machine)</td>
<td>16 cores (total = 256 cores) 48 GB DDR2 SDRAM, Interconnection Infiniband ConnectX IB Mellanox card</td>
</tr>
</tbody>
</table>

In order to construct the *signature*, the binary *Pas2p command* was used. *Pas2p tool* has been executed with the input parameters of similarity by default, considering a relevant phase when its time is 1% greater than the application execution time. When we analyzed the mentioned applications, *Pas2p tool* informed us that we could construct the *signature* without checkpoints, since all these applications have the relevant phases at the beginning of the execution, but in order to show the difference between both options, we constructed the *signature* both with and without checkpoints.

We constructed the *signatures* in the Nova Cluster (Base machine) and executed the *signature* of each application in the same cluster and in the IBM Cluster as target machines to predict the application performances.

The execution results on Nova Cluster are illustrated in Table 2. The SET column shows the Signature Execution Time, while AET column shows the Application Execution Time. In the SET vs. AET column is shows the percentage value of SET with respect to the AET. Then, the PET column shows the Predicted Execution Time, once the signature has been executed and finally results in the PET and AET columns being compared to obtain the Prediction Error. With these results, we obtain the SET around 6% of the Application Execution Time (AET), with a minimum prediction quality of 93.6%.

We also executed the *signature* of each application in the IBM Cluster, used as target system, to predict the Application Execution Time, as shown in Table 2. The applications have been fully executed in order to evaluate the quality of the prediction, and the prediction results are very similar to the execution in the Nova Cluster with a maximum predicted error of 6.4%.

Table 2. Predictions in cluster NOVA and cluster IBM

<table>
<thead>
<tr>
<th>Application</th>
<th>Cores</th>
<th>Signature Exec T. (SET) (Sec)</th>
<th>SET vs. AET (%)</th>
<th>Predicted Exec. T. (PET) (Sec)</th>
<th>Prediction Error (%)</th>
<th>App. Exec. T. (AET) (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG-256</td>
<td>256</td>
<td>16.81</td>
<td>4.04%</td>
<td>400.80</td>
<td>3.4%</td>
<td>415.15</td>
</tr>
<tr>
<td>BT-256</td>
<td>256</td>
<td>6.38</td>
<td>1.37%</td>
<td>492.31</td>
<td>6.4%</td>
<td>462.47</td>
</tr>
<tr>
<td>SP-256</td>
<td>256</td>
<td>5.72</td>
<td>0.62%</td>
<td>877.79</td>
<td>3.4%</td>
<td>908.36</td>
</tr>
<tr>
<td>SMG2000-256</td>
<td>256</td>
<td>134.30</td>
<td>1.7%</td>
<td>7305.97</td>
<td>3.8%</td>
<td>7596.21</td>
</tr>
<tr>
<td>Sweep3D-256</td>
<td>256</td>
<td>61.72</td>
<td>7.35%</td>
<td>1042.01</td>
<td>6.2%</td>
<td>1110.93</td>
</tr>
<tr>
<td>Moldy-256</td>
<td>128</td>
<td>31.72</td>
<td>3.29%</td>
<td>942.62</td>
<td>2.05%</td>
<td>962.25</td>
</tr>
<tr>
<td>M.DynaMix-256</td>
<td>128</td>
<td>16.38</td>
<td>3.17%</td>
<td>506.48</td>
<td>1.9%</td>
<td>516.32</td>
</tr>
<tr>
<td>CG-256</td>
<td>128</td>
<td>29.52</td>
<td>2.03%</td>
<td>2001.10</td>
<td>1.6%</td>
<td>2002.24</td>
</tr>
<tr>
<td>BT-256</td>
<td>128</td>
<td>17.78</td>
<td>1.48%</td>
<td>1182.67</td>
<td>1.5%</td>
<td>1200.85</td>
</tr>
<tr>
<td>SP-256</td>
<td>128</td>
<td>17.53</td>
<td>0.77%</td>
<td>2411.35</td>
<td>6.4%</td>
<td>2265.40</td>
</tr>
<tr>
<td>SMG2000-256</td>
<td>128</td>
<td>120.17</td>
<td>1.75%</td>
<td>6783.47</td>
<td>3.8%</td>
<td>6858.17</td>
</tr>
<tr>
<td>Sweep3D-256</td>
<td>128</td>
<td>82.28</td>
<td>7.62%</td>
<td>1043.01</td>
<td>3.5%</td>
<td>1079.13</td>
</tr>
<tr>
<td>Moldy-256</td>
<td>128</td>
<td>38.05</td>
<td>2.35%</td>
<td>1525.17</td>
<td>3.6%</td>
<td>1613.93</td>
</tr>
<tr>
<td>M.DynaMix-256</td>
<td>128</td>
<td>19.48</td>
<td>2.11%</td>
<td>953.61</td>
<td>3.5%</td>
<td>921.27</td>
</tr>
</tbody>
</table>

Table 3 shows the information generated by PAS2P toolkit to obtain and execute the signatures. Column 3 shows the trace log size obtained in the instrumentation module. Column 4 shows the table of phases size obtained with *pas2p tool*, which contains the relevant phases of the application, column 5 shows the number of relevant phases and column 6 shows the time required to analyze and obtain those relevant phases. It is important to remark that, although the time required to analyze can be long, depending on the Log trace size, this process is only carried out once in the base machine. Finally, the last two columns present the signature sizes without and with checkpoint respectively. Clearly, there is a reduction of the signature size when constructed without checkpoints.

6. Conclusions and future work

This paper proposes the PAS2P toolkit in order to automate the PAS2P methodology due to its complexity, providing users a way to make the whole process, from creating a signature to executing it on target systems, in
Table 3. Information generated by PAS2P toolkit by pasp2 tool

<table>
<thead>
<tr>
<th>Application</th>
<th>Cores</th>
<th>Log. Trace</th>
<th>Relevant Phases</th>
<th>Time required by Signature W/O. Pas2p Tool (Sec)</th>
<th>Signature W/O. Checkpoint</th>
<th>Signature W. Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG-256</td>
<td>256</td>
<td>428 MB</td>
<td>5.8 KB</td>
<td>4</td>
<td>297.03</td>
<td>7.8 KB</td>
</tr>
<tr>
<td>BT-256</td>
<td>256</td>
<td>573 MB</td>
<td>6.6 KB</td>
<td>13</td>
<td>222.27</td>
<td>8.1 KB</td>
</tr>
<tr>
<td>SP-256</td>
<td>256</td>
<td>1200 MB</td>
<td>72 KB</td>
<td>13</td>
<td>441.31</td>
<td>14.2 KB</td>
</tr>
<tr>
<td>SM2000-256</td>
<td>256</td>
<td>912 MB</td>
<td>67.4 KB</td>
<td>7</td>
<td>398.64</td>
<td>11.6 KB</td>
</tr>
<tr>
<td>Sweep3D-256</td>
<td>256</td>
<td>4.5 GB</td>
<td>67.3 KB</td>
<td>5</td>
<td>1342.46</td>
<td>4.1 KB</td>
</tr>
<tr>
<td>Moldy-256</td>
<td>128</td>
<td>2.7 GB</td>
<td>27 KB</td>
<td>4</td>
<td>732.83</td>
<td>4.2 KB</td>
</tr>
<tr>
<td>M.DynaMix-256</td>
<td>128</td>
<td>1.9 GB</td>
<td>14 KB</td>
<td>2</td>
<td>497.23</td>
<td>5.2 KB</td>
</tr>
</tbody>
</table>

user-space in an easy and fully automatic way.

The executable Pas2p tool is a sequential tool that analyzes the trace log file generated by our instrumentation. However, depending on the trace log size, the tool requires more memory in order to process the data. For this reason, we are working on the development of the parallel Pas2p tool. This code will allow us to use the same number of processes that the application actually used to be executed, ensuring the reduction of the analysis time.

Nowadays, the signatures constructed in base machines can be executed in target machines only if they have the same ISA (Instruction Set Architecture). We are developing a synthetic signature, which can be executed independently to the machine architecture.

Acknowledgements

This research has been supported by the MICINN Spain under contract TIN2007-64974, the MINECO (MICINN) Spain under contract TIN2011-24384, the European ITEA2 project H4H, No 09011 and the Avanza Competitividad I+D+I program under contract TSI-020400-2010-120.

References