EZTrace: a generic framework for performance analysis

François Trahay
Yutaka Ishikawa
Riken, University of Tokyo
{trahay, ishikawa}@il.is.s.u-tokyo.ac.jp

François Rue
Raymond Namyst
INRIA Bordeaux – Sud-Ouest
LaBRI, University of Bordeaux
{rue,namyst}@inria.fr

Mathieu Faverge
Jack Dongarra
University of Tennessee
{mfaverge,dongarra}@eecs.utk.edu

***Abstract*—Modern supercomputers with multi-core nodes enhanced by accelerators, as well as hybrid programming models, introduce more complexity in modern applications. Exploiting efficiently all the resources requires a complex analysis of the performance of applications in order to detect time-consuming or idle sections. This paper presents EZTRACE, a generic trace generation framework that aims at providing a simple way to analyze applications. EZTRACE is based on plugins that allow it to trace different programming models such as MPI, pthread or OpenMP as well as user-defined libraries or application. This framework uses two steps: one to collect the basic information during execution and one *post-mortem* analysis. This permits tracing the execution of applications with low overhead while allowing to refine the analysis after the execution of the program. We also present a simple script language for EZTRACE that gives the user the capability to easily define the functions to instrument without modifying the source code of the application. The evaluation of EZTRACE shows that the framework offers a convenient way to analyze applications without impacting the execution.**

I. INTRODUCTION

Numerical simulation has become one of the pillars of the process of science in many domains: numerous research topics now rely on computational simulation for modeling physical phenomenons. The needs for simulation in various computer power hungry research areas such as climate modeling, computational fluids dynamic or astrophysics have led to designing massively parallel computers that now reach petaflops.

Given the cost of such supercomputers, high performance applications are designed to exploit the available computing power to its maximum. During the development of an application, the optimization phase is crucial for improving the efficiency. However, this phase requires extensive understanding of the behavior and the performance of the application. The complexity of supercomputer hardware, due to the use of NUMA architectures or hierarchical caches, as well as the use of various programming models like MPI, OpenMP, MPI+threads, MPI+GPUs, PGAS models, makes it more and more difficult to understand the performance of an application.

Due to the complexity of the hardware and software stack, the use of convenient analysis tools is a great help for understanding the performance of an application. Such tools permit the user to follow the behavior of a program and to spot its problematic phases. However, the variety of scientific libraries and programing models makes it mandatory for such tools to be generic. Allowing easy instrumentation of any kind of library or application is crucial in order to work on most modern platforms and to meet the requirements of emerging programing models.

This paper describes the design and implementation of EZTRACE, a generic framework for performance analysis. EZTRACE uses a two phase mechanism based on plugins for tracing applications. This permits easy specification of the functions to analyze as well as the way they should be represented. Moreover, EZTRACE provides an easy to use script language that allows the user to instrument functions without modifying its source code. The remainder of this paper is organized as follows: in Section II, we present various research related to performance analysis. The design of EZTRACE is described in Section III. Section IV provides an overview of a script language for EZTRACE that permits users to easily instrument libraries and applications. The results of experiments conducted on EZTRACE are discussed in Section V. Finally, in Section VI we draw a conclusion and introduce future work.

II. RELATED WORK

Since the advent of parallel programming and the need for optimized applications, such research on performance analysis tools has been conducted. Two main directions were explored: application profiling and program tracing.

Research on profiling applications have led to tools capable of gathering statistics on the execution of a program and creating a non temporal report. Profiling tools such as GPROF [1] provide extensive information such as the list of time consuming functions. However these tools do not provide temporal performance data. Detecting typical abnormal behaviors – such as several MPI processes being blocked waiting for one particular message – is thus difficult.

Numerous research were conducted on tracing the execution of applications. This type of performance analysis tool permits the collection of temporal information on the behavior of a program. The resulting traces can then be visualized with tools such as VAMPIR [2], PARAVER [3] or VITE [4].

A. Manual instrumentation

The basic idea of application tracing is to record *events* when the application reaches specific key points. At the end of the application, a trace file that contains the recorded events is written on disk. This trace file can then be visualized or analyzed. Tracing the execution of an application can be achieved by manually instrumenting its source code. Specialized tools such as FXT [5] provide a convenient way of recording events during the execution of multithreaded applications.

In order to avoid reinventing the wheel each time an application is analyzed, various tools permit the use of pre-instrumented libraries. The MPI Profiling Interface [6] allows tracing tools to interface with MPI calls in a portable way. Specific libraries automatically intercept MPI functions [7], calls to pthread functions [8] or OpenMP directives [9]. This allows for easy analysis of any application based on these major programming models.

Pre-instrumenting a library also permits the user applying optimization during the construction of traces. For example, limiting the instrumentation to MPI functions allows for compressing the recorded events [10], [11]. Such techniques thus permit reducing the size of the output traces while preserving the information. However, this is only possible for a set of pre-defined functions: these techniques cannot be applied to user-defined functions.

Using manual instrumentation for analyzing an application requires heavy modifications of its source code unless the instrumentation is restricted to widely used libraries. If such source code modification can be performed for some applications, it can be problematic for large source codes.

B. Automatic instrumentation

Various work has focused on reducing the effort needed for instrumenting user-defined functions. Automatic instrumentation of applications allows the user for inserting event recording instructions in all the functions defined in a program. SCALASCA [12], TAU [13] or VAMPIRTRACE [14] are able to intercept calls to widely used libraries – MPI, OpenMP, etc. – and to automatically instrument the application functions during the compilation of the program. This technique works with most compilers and makes instrumenting the application functions easy. These tools also support binary modification – using the DYNINST API [15] – for instrumenting applications and can use the PAPI API [16] for collecting performance counters during the execution of the program.

However, recording events at each function call is time consuming. It thus causes a significant overhead on the performance of the application. Moreover, fully instrumenting an application may result in very large trace files. In order to avoid degrading the performance, these tools also allow manual instrumentation of the application. The user can thus focus on a set of interesting functions while reducing the size of the trace files and overhead.

In order to ease the analysis of applications, various work has focused on providing a complete software suite that permits tracing the execution of programs. INTEL TRACE ANALYZER AND COLLECTOR and OPEN|SPEEDSHOP [17] are able to instrument applications and to analyze the resulting traces. The PABLO [18] performance analysis environment also provides a graphical interface for instrumenting the source code of the application.

C. Discussion on current application tracing tools

Longstanding research efforts have led to numerous solutions for performance analysis. Thanks to these solutions, users can now trace applications based on widely used libraries such as MPI or OpenMP with little effort. The overall good performance of current tracing tools allows for collecting execution traces without disturbing the application.

However, while instrumenting MPI functions with current tools is straightforward, analyzing functions defined by the application requires modifying and recompiling the source code. Instrumenting manually an application is thus feasible, but this method is not convenient for large source code. Moreover, listing the instrumented functions requires browsing the source code in search of instrumentation instructions. A convenient way to specify the functions to instrument would ease the analysis of application.

The way *post-mortem* analysis is organized is another issue; once the execution of the application is over, it is not possible to interact with the traces. Once the execution trace is recorded, it is difficult to refine the analysis for focusing on a particular point.

Moreover, current application tracing tools do not permit choosing the representation of an event; the output trace shows that a function was called at one point, but representing non temporal events is tricky. Tracking the values of variables is also hard to achieve with current solutions.

We believe that performance analysis tools should provide a simple yet flexible way to instrument function and to customize their representation in the output trace. Such a solution would greatly ease the analysis of the performance of an application.

III. EZTRACE: A GENERIC FRAMEWORK FOR PERFORMANCE ANALYSIS

EZTRACE has been designed to tackle the limitations of current performance analysis tools described in the previous section. This framework relies on *plugins* in order to offer a generic way to analyze programs; depending on the application to analyze or on the point to focus on, several modules can be loaded. EZTRACE provides pre-defined *plugins* that permit the analysis of applications that use MPI libraries, GNU OpenMP, Pthreads or the PLASMA [19] library. However, user-defined *plugins* can also be loaded in order to analyze application functions or custom libraries.

EZTRACE uses a two phase mechanism for performance analysis. During the first phase that occurs while the application is executed, functions are intercepted and events are recorded. After the execution of the application, the *post-mortem* analysis phase is in charge of interpreting the recorded events. This two phase mechanism permits the library to

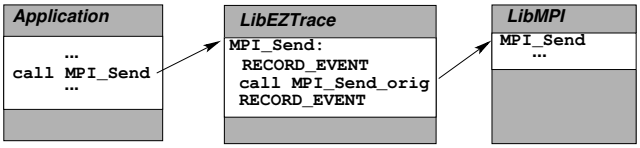

Fig. 1. Instrumentation of library functions using `dlsym`

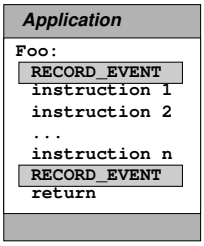

Fig. 2. Instrumentation of application functions using DYNINST

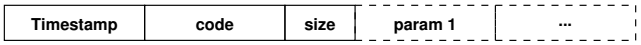

Fig. 3. FXT event format.

separate the recording of a function call from its interpretation. It thus allows the user to interpret a function call event in different ways depending on the point he/she wants to focus on. For example, a call to `MPI_Recv` can be interpreted by representing a communication – if the user focuses on the communication pattern of the application – or by notifying the modification of a buffer if the analysis focuses on memory accesses. Separating the recording of events from their interpretation also allows for reducing the overhead of profiling a program; during the execution of the application, the analysis tool should avoid performing time-consuming tasks such as computing statistics or interpreting function calls.

In this Section, we first describe the way EZTRACE records events while the application runs. Then we explain how the *post-mortem* analysis phase works. We conclude this Section by discussing the benefits brought by EZTRACE.

A. Recording events

During the execution of the application, EZTRACE intercepts calls to the functions specified by *plugins* and records events for each of them. Depending on the type of functions, EZTRACE uses two different mechanisms for interception. The functions defined in shared libraries are instrumented using a `dlsym` mechanism whereas functions defined by the application are intercepted using the DYNINST [15] tool.

The `dlsym` mechanism uses LD_PRELOAD for overriding the functions to instrument as illustrated in Figure 1. When the EZTRACE library is loaded, it retrieves the addresses of the functions to instrument. When the application calls one of these functions, the version implemented in EZTRACE is called. This function records events and calls the actual function.

The `dlsym` mechanism cannot be used for functions defined in the application since there is no symbol resolution. It is thus necessary to patch the application. EZTRACE uses the DYNINST [15] tool for instrumenting the program on the fly. As depicted in Figure 2, EZTRACE uses DYNINST for inserting calls to `RECORD_EVENT` at the beginning and/or at the end of each function to instrument. When the application calls an instrumented function, EZTRACE can thus record events.

For recording events, EZTRACE relies on the FXT library [5]. Each process being instrumented by EZTRACE generates a trace file using FXT. In order to keep the trace size as compact as possible, FXT records events in a binary format that contains only the minimum amount of information. As depicted in Figure 3, FXT only records a timestamp, an *event code* and optional parameters. Since FXT instrumentation is fully reentrant, multiple threads can record events simultaneously. The use of atomic operations in FXT for handling concurrent event recording allows EZTRACE to maintain the integrity of the output trace without using expensive synchronization primitives.

EZTRACE plugins associate *event codes* to the functions that they intercept. Thus, when intercepting a function, EZTRACE provides FXT with the *event code* corresponding to this function and the identifier of the calling thread. The automatic recording of the thread identifier provides useful information to the analysis phase, especially for multithreaded applications. For example, it allows the library to determine which threads are sending or receiving messages using MPI. Depending on the function to instrument, EZTRACE may add optional parameters. For example, the arguments passed to `MPI_Send` are reported to the FXT trace so that the *post-mortem* phase can reconstruct the application communication pattern.

B. Post-mortem analysis

After the execution of the application, the trace files are analyzed during the *post-mortem* phase. Recorded events are read in the execution order and are interpreted by *plugins*. During this interpretation phase, the plugins can generate a merged trace file – in the OTF [20] or PAJÉ [21] formats – or compute statistics. Since each *event code* corresponds to a function that was intercepted during the recording phase, a *plugin* has to interpret all the *event codes* that it defines. However, a single *event code* may be interpreted by different plugins. For example, the code associated with the `MPI_Recv` function may be used for representing a communication, or for notifying a memory access depending on the interpretation plugin.

While the FXT trace consists of a set of unrelated events, the *post-mortem* analysis phase adds semantics that allows the user to understand the behavior of the application. For doing this, EZTRACE acts on the *trace modifiers* provided by the GTG library [22]; a set of containers is defined – processes, threads, GPUs, etc. – and interpretation plugins modify their information over time. Plugins can modify the *state* of a container for pointing out a particular phase in the

application – a thread entered a critical section, a particular function was called, etc. Plugins can also notify non lasting *events* – submission of a new job, lock released, beginning of a non-blocking communication, etc. The value of *variables* can also be tracked. This can be useful for representing the number of pending jobs, the total allocated memory or any quantity that may give hints about the behavior of the application.

Moreover, depending on the semantics of the recorded events, EZTRACE plugins can extract statistics. This can provide the user with useful information. For example, statistics on MPI messages can reveal the amount of communication between each pair of processes. This information, as well as the communication pattern of the application, may help the user improve the placement of processes in order to improve the performance of the application [23].

C. Benefits brought by EZTrace

The use of a generic framework for performance analysis such as EZTRACE permits users to collect information on the execution of most HPC applications. Since EZTRACE relies on plugins, any application can be analyzed as long as plugins for its libraries are available.

The two phase mechanism used in EZTRACE allows to separated the recording of the execution of an application from its analysis. This permits to run the application only once and to work on the collected traces later. It is thus possible to refine the analysis of a set of traces. For example, the execution of an application mixing MPI and OpenMP can be recorded using EZTRACE. The analysis of this execution can then focus on either MPI, OpenMP or both paradigms depending on the user interest.

IV. ANALYZING USER-DEFINED FUNCTIONS

The instrumentation of functions implemented in widely used libraries can provide useful information for a large range of applications. For example, recording calls to MPI functions will help analyzing most MPI-based applications. When developing a framework for performance analysis, providing predefined mechanisms for such libraries is thus crucial. However, pre-defining mechanisms for all the existing libraries is not feasible, and convenient tools for analyzing less common libraries as well as applications are required. This way, the framework can be used for any application regardless of the libraries it uses.

In Section II, we have shown that instrumenting custom applications or libraries with available frameworks can be tedious. Since EZTRACE uses a plugin mechanism, analyzing an MPI program is not different from analyzing an application based on a random library, as long as a plugin for this library is available. It is thus important for the user that the framework provides a convenient way to create *plugins*.

A. Overview of the plugin mechanism

Since EZTRACE uses a two phase mechanism for application analysis, plugins are made up of two parts:

```
int foo(int arg1, int arg2) {
  EZTRACE_EVENT2(FOO_ENTRY, arg1, arg2);
  int ret = foo_callback(arg1, arg2);
  EZTRACE_EVENT0(FOO_EXIT);
  return ret;
}
```

Fig. 4. Example of function overriding.

```
void handle_foo_entry() {
  INIT_THREAD_ID(thread_id);
  pushState(CURRENT, "ST_Thread",
      thread_id, "foo");
  free(thread_id);
}
```

Fig. 5. Example of event interpretation.

a) Event collection: This part of the plugin intercepts calls to a set of specified functions. When this happen, events are recorded before and/or after the execution of the function. Depending on whether the functions are defined in a shared library or in the application, the plugin may use function overriding or dynamic instrumentation for intercepting function calls.

b) Trace interpretation: After the execution of the application, the plugin interprets the events that were recorded. For doing this, EZTRACE delegates the handling of each recorded event to the plugin that is then free to interpret it.

Writing a plugin for EZTRACE thus requires implementing both an event collection module and a trace interpreter module. An example of such modules is given in Figures 4 and 5. In this example, the `foo` function is intercepted and events are recorded before and after the call to the original function. During the *post-mortem* analysis, the `handle_foo_entry` function is called each time the `FOO_ENTRY` event code appears. This function modifies the state of the calling thread in the output trace.

Even if writing a plugin for EZTRACE is quite simple, this can be simplified in order to make it accessible to end-users that are not familiar with C programming. For that reason, EZTRACE provides a convenient way to create plugins using a simple script language. This language allows for easily describing the functions to instrument as well as their interpretation. User-defined scripts can then be converted into plugins using a source to source compiler.

B. A script language for plugin creation

Due to the two phase mechanism used in EZTRACE, writing a plugin requires specifying the functions to instrument and the interpretation of these functions calls in the output trace. However, providing EZTRACE with the interpretation is often sufficient for guessing how to instrument a function. For example, changing the *state* of a thread during the execution of a function implies recording an event at the beginning and at the end of the corresponding function, whereas notifying the

```
NAME foo
DESC "Plugin for the foo library"
LANGUAGE C
TYPE LIBRARY

int foo(int arg1, int arg2)
BEGIN
  RECORD_STATE("doing function foo")
END

void bar(int arg1)
BEGIN
  EVENT("function bar called")
END
```

Fig. 6. Example plugin.

occurrence of a function with an *event* only requires recording an event at the beginning of the function. The script language for describing the functions to analyze thus focuses on the interpretation of these functions in the output trace, leaving the instrumentation to EZTRACE.

As illustrated in Figure 6, creating a plugin using this script language boils down to describing the plugin and interpreting each function that has to be analyzed.

1) Describing a plugin: In order to generate a plugin, the source to source compiler needs some general information. For example, the *plugin type* describes whether the functions to intercept are defined in a shared library or in an application. Depending on the *plugin type*, the source to source compiler chooses between the DYNINST and the `dlsym` mechanisms.

2) Interpreting a function: The script describes the functions to instrument by specifying each function prototype and its interpretation. A set of keywords permits describing how to generate the output trace by acting on the *trace modifiers* described in Section III-B:

- `PUSH_STATE(new_state)` changes the *state* of a thread.
- `POP_STATE()` reverts the *state* of a thread to its value before the previous call to `PUSH_STATE`.
- `RECORD_STATE(new_state)` changes the *state* of a thread during the execution of a function. This is equivalent to calling `PUSH_STATE` at the beginning of a function and `POP_STATE` at the end of it.
- `EVENT(event_name)` records an *event*.
- `SET_VAR(var_name, value)` assigns a value to a *variable*.
- `ADD_VAR(var_name, value)` increases the value of a *variable*.
- `SUB_VAR(var_name, value)` decreases the value of a variable.

These keywords allow action on the output trace, however in order to control more precisely the generation of a trace it is necessary to distinguish the actions that should be performed before the function call from the ones that should be done after it. The `CALL_FUNC` keyword allows the user to specify at which point the function should be called and thus permits

```
int submit_jobs(int nb_jobs)
BEGIN
  ADD_VAR("Number of jobs", nb_jobs)
  CALL_FUNC
  EVENT("New jobs")
END
```

Fig. 7. Example of function instrumentation using the script language.

the library to guess when an action should be performed. For example, in the function listed in Figure 7, the *variable* `Number of jobs` is increased before the function call, whereas the *event* `New jobs` is set when the function ends.

The script language also provides conditional statements. It is thus possible to take the result of a function call or the value of a parameter into account.

In order to simplify the development of plugin, it is also possible to skip the interpretation instructions. In that case, the source to source compiler implicitly interprets the function using `RECORD_STATE`. It is thus possible to create a plugin by providing only a list of function prototypes.

C. The source to source compiler

The creation of a plugin suitable for EZTRACE relies on a source to source compiler that interprets the script and generates the C files described in Section IV-A. The compiler fills template files with the information contained in the script. It generates the source codes that instrument the functions provided by the user and that interpret events during the *post-mortem* phase.

In order to minimize the overhead of the function instrumentation, unnecessary recording of events are avoided; if the script specifies that a function should be interpreted by increasing a *variable* in the output trace, only one event is recorded during the event collection phase.

D. Discussion about the script language

The script language provides a convenient way to create EZTRACE plugins. By simply listing the functions to instrument it is possible to create a minimal plugin that shows the function calls during an application execution. The interpretation keywords allow the user to describe more precisely the generation of the output trace while minimizing the overhead of the instrumentation of functions. In most cases, it is thus possible to describe how to depict a function call in the output trace.

The small set of interpretation keywords permits acting on all the *trace modifiers* provided by the trace formats. Thus, the simplicity of the script language does not restrict the possibilities of event interpretation. Moreover, since the source to source compiler generates C files, it is possible to use them as a basis and to modify them in order to control more precisely the interpretation of a function.

Creating a plugin for EZTRACE can thus be achieved easily by providing a set of functions to instrument. The interpretation of function calls can be controlled precisely. As a result,

Method	Open MPI	VampirTrace	EZTrace	# of events
Automatic	4.99	6.12	5.68	121 000
Manual	4.99	5.71	5.67	80 800

TABLE I
RESULTS OF THE 16-BYTES LATENCY TEST

the script language is powerful enough for generating plugins for most users, while leaving the possibility for the advanced users to analyze more precisely an application execution.

V. EVALUATION

When analyzing the performance of an application with a framework such as EZTRACE, it is important that the application run during which the data is collected is representative of usual runs.Thus, the analysis framework should not modify the behavior of the application; the overhead of instrumenting the application should be as inexpensive as possible.

In this Section, we assess the impact on performance of using EZTRACE. We evaluate the raw performance of the `dlsym` and DYNINST mechanisms before comparing EZTRACE with VAMPIRTRACE on application kernels. Then we show how the script language described in Section IV-B can be used for analyzing the NAS SP kernel.

The results that we show in this Section were measured on the CLUSTER0 platform that is composed of 32 nodes. Each box is equipped with two 2.2 GHz dual-core OPTERON (2214 HE) CPUs featuring 4 GB of memory. These nodes are running Linux 2.6.32 and are connected through MYRINET MYRI-10G NICs. These results were obtained using OPEN MPI [24]. We compare EZTRACE with VAMPIRTRACE in its 5.9 version.

A. Overhead of trace collection

In order to evaluate the overhead of collecting event during the execution of an application, we use a MPI ping pong program. We measure the latency obtained for 16-bytes messages.

We run this program with automatic and manual instrumentation and we compare to the performance obtained without any instrumentation. The automatic instrumentation is obtained by using VAMPIRTRACE MPI trace capabilities and the EZTRACE MPI plugin. For VAMPIRTRACE, the program is linked against the VAMPIRTRACE library so that it uses its hooks to MPI functions, while EZTRACE uses the `dlsym` mechanism for intercepting calls to MPI functions.

The manual instrumentation with VAMPIRTRACE is obtained by adding calls to VT_USER_START and VT_USER_END when invoking MPI_Send and MPI_Recv. The automatic instrumentation of MPI functions is disabled by setting the `VT_MPITRACE` environment variable to `no`. For EZTRACE, we use the script language described in Section IV-B for instrumenting manually with DYNINST the functions that send and receive messages.

Table I shows the results that we obtained for automatic and manual instrumentation. Using VAMPIRTRACE automatic function tracing causes an overhead of 1.1 μs while the manual instrumentation degrades the latency by 700 ns. This difference is due to VAMPIRTRACE's handling of these function. The manual instrumentation generates events at the entry and the exit of functions while VAMPIRTRACE MPI hooks also create *SendMessage* or *ReceiveMessage* events. Recording one event with VAMPIRTRACE thus costs approximately 350 ns.

Instrumenting the application using EZTRACE `dlsym` or DYNINST mechanisms causes an overhead of 700 ns. In both cases, events are recorded at the entry and the exit of functions. The EZTRACE pre-defined MPI plugin creates the *SendMessage* and *ReceiveMessage* during the *post-mortem* analysis phase, avoiding additional overhead during the execution of the application. Recording an event with EZTRACE thus costs 350 ns, just as with VAMPIRTRACE.

EZTRACE generates the same number of events as VAMPIRTRACE in both cases. However, the output traces obtained with manual and automatic instrumentation do not contain the same number of events. This is due to the way VAMPIRTRACE and EZTRACE work. While the manual instrumentation only results in *EnterFunction* and *LeaveFunction* events, the MPI modules also generate *SendMessage* and *RecvMessage* events.

The overhead of recording an event with EZTRACE is the same as with VAMPIRTRACE. However, EZTRACE limits its instrumentation to the minimum. Only entry and exit of functions are recorded, leaving the interpretation of these events – the creation of *SendMessage* or *ReceiveMessages* – to the *post-mortem* analysis phase.

B. NAS parallel benchmarks

Beside the previous raw performance experiments, we also evaluate NAS [25] application kernels. The experiments were carried out with 4 computing processes on Class A and 32 processes on Class B. Kernels requiring a square number of processes – BT and SP – were run on 36 processes for Class B. We used 8 machines (9 for BT and SP on Class B) for these experiments. Thus, during the Class A measurements only one process runs on a node, whereas for the Class B experiments 4 processes are executed on each node.

Table II summarizes the results that we obtained. The number of OTF events generated by the LU kernel for Class B is not available due to problems in the current implementation that cause EZTRACE to crash during the *post-mortem* analysis in this case. The results show that instrumenting the kernels with VAMPIRTRACE or EZTRACE causes little variation in the execution time. In all cases, the difference is less than 2 %. The numbers of events show that intensive event recording – such as during the MG kernel for Class B – does not affect the performance significantly.

C. Analysis of an application

We used the script language described in section IV-B for instrumenting the SP kernel from the NAS Parallel Benchmark suite. This FORTRAN program, based on MPI, solves scalar pentadiagonal systems of equations.

The SP kernel calls the `adi` function at each iteration. This function invokes the `copy_faces` function that exchanges

Kernel	Class	# Processes	OpenMPI	VampirTrace		EZTrace		# Events	# Events / s
				Execution (s)	Overhead	Execution (s)	Overhead		
BT	A	4	70.57	70.58	0.01 %	70.39	-0.26 %	58 120	825
CG	A	4	2.64	2.68	1.52 %	2.68	1.52 %	33 624	12 546
EP	A	4	9.61	9.69	0.83 %	9.72	1.14 %	48	5
FT	A	4	6.63	6.67	0.55 %	6.62	-0.20 %	144	22
IS	A	4	0.63	0.64	2.13 %	0.62	-1.06 %	299	482
LU	A	4	42.08	42.15	0.17 %	41.39	-1.64 %	508 360	12 282
MG	A	4	5.04	5.06	0.46 %	5.07	0.66 %	15 096	2978
SP	A	4	166.25	165.94	-0.18 %	166.32	0.04 %	115 704	696
BT	B	36	26.08	25.83	-0.97 %	26.37	1.10 %	1 565 064	59 350
CG	B	32	16.29	16.46	1.02 %	16.60	1.88 %	3 198 272	192 667
EP	B	32	4.81	4.79	-0.42 %	4.76	-1.04 %	384	81
FT	B	32	11.76	11.61	-1.30 %	11.55	-1.81 %	2 944	255
IS	B	32	0.97	0.96	-1.03 %	0.96	-1.03 %	2 427	2 580
LU	B	32	33.75	34.11	1.07 %	33.67	-0.24 %	–	–
MG	B	32	2.14	2.16	0.78 %	2.13	-0.62 %	450 528	215 515
SP	B	36	51.18	51.98	1.57 %	52.07	1.75 %	3 120 120	59 922

TABLE II
NAS PARALLEL BENCHMARK PERFORMANCE FOR CLASS A AND B

```
BEGIN_MODULE
  NAME plugin_nas_sp
  DESC "Plugin for the NAS SP kernel"
  LANGUAGE FORTRAN
  TYPE APPLICATION

   adi()
   BEGIN
     EVENT("New loop")
   END

   copy_faces()
   txinvr()
   x_solve()
   y_solve()
   z_solve()
   add()
END_MODULE
```

Fig. 8. Plugin used for the NAS SP kernel.

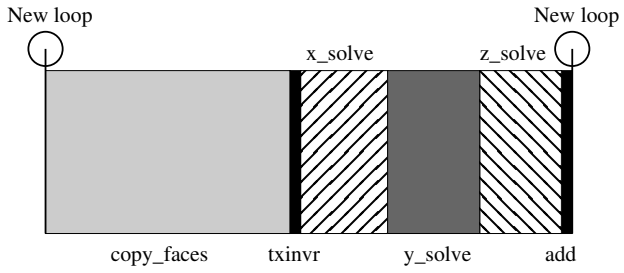

Fig. 9. Execution pattern of NAS SP

data with the neighboring processes. It then calls `txinvr` for performing a matrix-vector multiplication, `x_solve`, that computes a solution in the x-direction, as well as `y_solve` and `z_solve` that do the same for y and z directions.

Figure 8 shows the script we used for analyzing the SP kernel. The generated plugin instruments the functions called during each iteration and, in order to locate more easily the beginning of iterations, a "New loop" event is generated when `adi` is called.

Instrumenting the SP kernel with VAMPIRTRACE requires modifying its source code by inserting calls to `VT_USER_START` and `VT_USER_STOP`. This instrumentation adds 24 lines of codes in 7 differents files. Using an EZTRACE script for instrumenting the application thus permits regrouping all the instrumentation code in a single file while only requiring a few lines of instructions.

With EZTRACE we run the SP kernel with 4 processes on Class W. The output trace shows the repeated pattern depicted in Figure 9. The output trace also reveals which functions are the more time consuming. In this experiment, 32 % of the execution time was spent in the `copy_faces` function, each of the `x_solve`, `y_solve` and `z_solve` functions took 20 % of the time whereas less than 5 % of the time was spent in the `txinvr` and `add` functions. Analyzing an application with EZTRACE thus provides useful clues on which functions are worth optimizing: in the case of the SP kernel, tuning the `txinvr` function is likely to bring little improvement and the application developer should focus on the `copy_faces`, `x_solve`, `y_solve` or `z_solve` functions.

This experiment shows that the script language provided by EZTRACE allows users to analyze an application with little effort. Creating a *plugin* only requires a few lines of script and the application does not need to be modified nor recompiled. A simple *plugin* allows EZTRACE to reveal the general execution pattern of an application, to locate the time consuming functions and to gather statistics on the application execution.

VI. CONCLUSION AND FUTURE WORKS

Modern applications are more and more sophisticated by the use of different programming models like MPI, OpenMP or pthread but also because of modern architectures of supercom-

puters which include different units of computations like cores and accelerators, hierarchical levels of caches and memory and sometimes different networks. All these characteristics make it more and more difficult to efficiently exploit current supercomputers. Application developers thus need convenient tools for understanding the performance of their programs and to detect the phases that can be improved. Several tools offer the capability to trace one or several components, usually MPI and OpenMP, but tracing a set of user-defined functions can be tedious and requires users to modify the source code of the application. The second problem is how to analyze these data. There are many ways to interpret an execution trace depending on what is being searched for.

We proposed in this paper a generic framework for application analysis based on two steps. During the execution of the program, a set of functions specified by plugins are instrumented and trace files are generated. After the application run, a *post-mortem* analysis interprets the traces. This also permits generating execution traces with a low impact on the performance of the application while allowing refinement of the analysis after the program run. The selection of the information we are interested in gives a precise analysis of the trace without surfeit of information. The script language developed with EZTRACE provides a simple way for the user to describe which functions he/she wants to follow and how to analyze the data collected. This way a simple user as well as an expert can easily generate a new plugin to study what is time consuming in his/her application or why one scheduling is better than another. The experiments on NAS benchmarks show that the overhead on the application execution is less than two percent and the use of binary files to collect information makes it small. These files can later be interpreted in different ways regarding what is interesting for the user. The evaluation also shows that the script language provided by EZTRACE permits users to easily analyze an application without modifying its source code.

EZTRACE is available as an open source project online [26]. Our future work is to integrate hardware counters collected by libraries such as PAPI [16] into our trace generation to follow the evolution of these counters during execution. We also plan to extend the plugin generator language to take this into account and to provide to the user a simple way to access these counters via this language.

REFERENCES

[1] S. Graham, P. Kessler, and M. Mckusick, "Gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[2] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Müller, and W. Nagel, "The Vampir Performance Analysis Tool-Set," *Tools for High Performance Computing*, pp. 139–155, 2008.

[3] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Transputer and occam developments: WoTUG-18: proceedings of the 187th world occam and Transputer User Group Technical Meeting, 9th-13th April 1995, Manchester, UK*. Ios Pr Inc, 1995, p. 17.

[4] "Visual Trace Explorer." [Online]. Available: http://vite.gforge.inria.fr/

[5] V. Danjean, R. Namyst, and P. Wacrenier, "An efficient multi-level trace toolkit for multi-threaded applications," *Euro-Par 2005 Parallel Processing*, pp. 166–175, 2005.

[6] E. Karrels and E. Lusk, "Performance analysis of MPI programs," in *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing*. SIAM Publications, 1994, pp. 195–200.

[7] J. Vetter and B. de Supinski, "Dynamic software testing of MPI applications with Umpire," in *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, 2006, p. 51.

[8] S. Bull, "NPTL Stabilization Project," in *Linux Symposium*, p. 111.

[9] J. Caubet, J. Gimenez, J. Labarta, L. DeRose, and J. Vetter, "A dynamic tracing mechanism for performance analysis of OpenMP applications," *OpenMP Shared Memory Parallel Programming*, pp. 53–67, 2001.

[10] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. 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.

[11] K. Vijayakumar, F. Mueller, X. Ma, and P. Roth, "Scalable i/o tracing and analysis," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM, 2009, pp. 26–31.

[12] M. Geimer, F. Wolf, B. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[13] S. Shende and A. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, p. 287, 2006.

[14] M. Muller, A. Knupfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. Nagel, "Developing scalable applications with Vampir, VampirServer and VampirTrace," *Proceedings of the Minisymposium on Scalability and Usability of HPC Programming Tools at PARCO*, 2007.

[15] B. Buck and J. Hollingsworth, "An API for runtime code patching," *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[16] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, p. 189, 2000.

[17] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open| SpeedShop: An open source infrastructure for parallel performance analysis," *Scientific Programming*, vol. 16, no. 2, pp. 105–121, 2008.

[18] D. Reed, P. Roth, R. Aydt, K. Shields, L. Tavera, R. Noe, and B. Schwartz, "Scalable performance analysis: The Pablo performance analysis environment," in *Scalable Parallel Libraries Conference, 1993., Proceedings of the*. IEEE, 2002, pp. 104–113.

[19] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," in *Journal of Physics: Conference Series*, vol. 180. IOP Publishing, 2009, p. 012037.

[20] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel, "Introducing the open trace format (OTF)," *Computational Science–ICCS 2006*, pp. 526–533, 2006.

[21] J. Chassin de Kergommeaux, B. de Oliveira Stein, and G. Mounié, "Pajé input data Format," Tech. Rep., 2003.

[22] "GTG: Generic Trace Generator." [Online]. Available: http://gforge.inria.fr/projects/gtg/

[23] G. Mercier and J. Clet-Ortega, "Towards an efficient process placement policy for mpi applications in multicore environments," in *EuroPVM/MPI*, ser. Lecture Notes in Computer Science, vol. 5759. Espoo, Finland: Springer, Sep. 2009, pp. 104–115.

[24] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 353–377, 2004.

[25] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," Tech. Rep., 1995.

[26] "EZTrace: easy to use trace generator." [Online]. Available: http://eztrace.gforge.inria.fr/