Unifying the Analysis of Performance Event Streams at the Consumer Interface Level

Jean-Baptiste Besnard, Allen D. Malony, Sameer Shende, Marc Pérache, Patrick Carribault and Julien Jaeger

Abstract Several instrumentation interfaces have been developed for parallel programs to make observable actions that take place during execution and to make accessible information about the program's behavior and performance. Following in the footsteps of the successful profiling interface for MPI (PMPI), new rich interfaces to expose internal operation of MPI (MPI-T) and OpenMP (OMPT) runtimes are now in the standards. Taking advantage of these interfaces requires tools to selectively collect events from multiples interfaces by various techniques: function interposition (PMPI), value read (MPI-T), and callbacks (OMPT). In this paper, we present the unified instrumentation pipeline proposed by the MALP infrastructure that can be used to forward a variety of fine-grained events from multiple interfaces online to multi-threaded analysis processes implemented orthogonally with plugins. In essence, our contribution complements "front-end" instrumentation mechanisms by a generic "back-end" *event consumption interface* that allows "consumer" callbacks to generate performance measurements in various formats for analysis and transport. With such support, online and post-mortem cases become similar from

J.-B. Besnard (✉)
ParaTools SAS, Arpajon, France
e-mail: jbbesnard@paratools.fr

A. D. Malony · S. Shende
ParaTools Inc., Eugene, USA
e-mail: malony@paratools.com

S. Shende
e-mail: sameer@paratools.com

M. Pérache · P. Carribault · J. Jaeger
CEA, Arpajon, France
e-mail: marc.perache@cea.fr

P. Carribault
e-mail: patrick.carribaul@cea.fr

J. Jaeger
e-mail: julien.jaeger@cea.fr

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2017*,
https://doi.org/10.1007/978-3-030-11987-4_4

an analysis point of view, making it possible to build more unified and consistent analysis frameworks. The paper describes the approach and demonstrates its benefits with several use cases.

1 Introduction

There has always been an intimate *pas de trois* between the need for greater computational power in scientific discovery, the evolution of HPC hardware and software to provide next-generation computing potential, and the nature of parallel performance on HPC platforms that determines what can be achieved. By now, the dance is well-known. New domain applications try to maximize parallelism and handle larger problems, pushing beyond the limits of present high-performance computing (HPC) capabilities. In response, HPC architectures and technology advance, resulting in greater programming complexity in order to access the potential these new HPC machines have to offer. However, the performance complexity is also in flux. For instance, a parallel application might be able to expose a high level of concurrency for a large number of threads to execute on a many-core processor. The problem is that the optimal number of threads to use and even cores to allocate at any point in the program depends on the performance interactions involved. The interplay of the application and the HPC resources with respect to performance factors is subtle and not easily understood. Performance consequences include:

- a decreasing memory per thread jeopardizes a pure distributed memory model due to both the memory replication (halo cells) and communication overhead [4];
- smaller cores result in lower sequential performance, requiring the programs to exhibit more parallelism to achieve the same performance;
- larger vectorial units requiring optimizing compilers and a correct expression of computing loops (making them *vectorizable*), possibly new optimized instructions; and
- complex memory hierarchies demanding a careful data-placement and tracking (first-touch), including important NUMA effects, stacked memory, deep cache architectures and meshed processors.

A strategy in response to these complexity challenges has been to advance parallel programming languages to expose sufficient parallelism that can be mapped to multiple shared-memory cores via multi-threading and to distributed nodes via message passing. Unified programming abstractions and hybrid ones are commonly explored. The hybrid approach has been the most widely adopted due to the progressive shift it allows from a pure MPI program to an *MPI + X* one. In general, X is some form of shared-memory parallelism, such as provided by OpenMP directives for parallelizing loops or expressing tasks. The standardization and portability of MPI and OpenMP, plus the large corpus of multi-million line programs, makes it a desirable combination. In general, hybrid parallel programming methodologies attempt to deal

with the important problem of application engineering in the face of rapid hardware evolutions. The hope is to minimize the impact on the code which is costly to develop and often represents a multi-year investment spanning over machine generations.

However attractive this strategy is, it is incomplete because performance aspects are not fully considered. Beyond providing a means to enable instrumentation, most parallel programming systems do not provide direct support for performance analysis or optimization. Certainly, parallel performance tools exist, but there is a strong motivation to make sure performance technology can interoperate effectively. Interestingly, the rise of hardware targeted to HPC will make this more of a challenge. Dedicated ARM processors in Japan and the Sunway processors [10] in China increase the number of architectures tools will need to support. Similarly, custom high-performance networks such as the Tofu interconnect [2] in Japan and the Bull Exascale Interconnect [7] in France add communication performance factors specific to those platforms. Infrastructure and power monitoring with the Bull power-tracking FPGA results in additional components needing to be integrated. All these evolutions will need (often already lack) suitable tools to both allow and assess them. Yet, the shift to such systems will likely require a substantial porting effort, which will be further complicated by the dearth of performance tools.

The purpose of this paper is to look at the problem of tool interoperability and propose an interface that could be leveraged more efficiently to address certain complexity challenges of HPC hardware and software evolution. In particular, we consider tools with complementary instrumentation mechanisms and pose the open question of how to couple their data streams for measurement and downstream processing. Indeed, several leading performance tool systems have *instrumentation chains* that are redundant, while their actual value comes from the analysis insight they provide. Thus, we propose a component-based view of the *instrumentation chain* by defining a coupling at the consumer-level interface instead of at the transport-level, as is commonly done.

To support this idea, the rest of the paper is organized as follows. Section 2 first describes our component-based instrumentation chain and how it is articulated. The feasibility, extensibility, and a possible implementation of our proposed common consumer interface is discussed in Sect. 3. Section 4 illustrates an instance previous ideas in the context of the Multi-Application Online Profiling tool (MALP). Concluding remarks and future directions are then given.

2 Components in the Instrumentation Chain

The typical parallel program *instrumentation chain* focuses on "observation" points and the mechanisms to make these visible, versus potential "coupling" opportunities for interoperability. Our goal is to determine where coupling interfaces could be enabled exist and how to support them. To do so, we will first have to describe a general model of the instrumentation chain and then consider each of its components. With this groundwork, we propose a consumer-level interface and discuss its prototype implementation.

2.1 General Model

The *instrumentation chain* is a general term describing all the steps needed by a tool to obtain and forward a target program state to intelligible analysis consumers, thereby providing the end-user with a clear view of what is the actual behavior of its program. This model applies for debugging, validation, and performance assessment with only little variations.

Figure 1 shows the major components in the instrumentation chain that are used to build tools. From left to right, we first have the *event-source* where the event is generated, for example, by a function called or being called. Second, the data associated with what was observed at the source is structured as an *event* represented by different fields encoding observable parameters. In the middle of the figure, these events are *transported* to the analysis. This process can take several forms and possibly converts the event to an intermediate representation suitable for the transport layer. Then, on the analysis side, the transported event is first decoded to a state similar to the original *event* representation. Eventually, events are projected to the final *analysis* before being presented to the end-user.

The instrumentation chain shown in Fig. 1 is the way support tools explore parallel program state, systematically intercepting, encoding, forwarding, decoding, and eventually projecting what happened in the program. Events by nature are imperceptible, thus requiring this chain to be actualized for analysis. Also, note that we do not see this chain as exclusively feed-forward. It is also possible for the end-user to *consume* events from the event source in a "pull" model. In this case, for example, when considering a debugger, the analysis is directly consuming events through the *ptrace* transport layer.

Let us further describe the components that we have just introduced in more detail. This gives us the opportunity to also point to the rich related work implementing these various building blocks.

2.2 Event Sources

The *event source* is where the observable is generated. In all cases, the goal is to extract information from program's state, but this may happen in many different forms. There are several possibilities when it comes to instrumenting a parallel program [23], which we attempt to sort in categories before discussing them more generally. Expert tool developers are quite familiar with these methods.

Fig. 1 Macroscopic view of instrumentation chain building blocks

- **Manual instrumentation** is the process of manually adding probes in the target program. It can be, for example, tools call outlining phases, perhaps using *printf* to output arbitrary values. Moreover, the user may expose an interface to make visible the state of its program.
- **Source-to-source instrumentation** consists of parsing the source code in order to insert probes and writing out the instrumented code in a source form to be compiled. This was the case for the Opari instrumenter [20] before the advent of the OMPT [8] interface. Included in this category are preprocessors that define redirecting calls to instrumented libraries.
- **Compiler instrumentation** can be used to instrument functions using directives such as `-finstrument-functions`. Moreover, pragmas or functions attributes can be used to create weak symbols to be used at link or load time, as done for example in the MPI profiling interface. An instrumented program is produced as the result of the compilation process.
- **Linker and loader** mechanism can be used to alter symbol resolution either by changing library order at link time, wrapping symbols (using `-Wl,-wrap`) or by inserting at runtime (with `LD_PRELOAD`) a library superseding existing ones. At this level, weak symbols can also be replaced by instrumented functions.
- **Runtime tools** can provide verbose information source for tools. The MPI tools interface with MPI-T [16], the OpenMP tools and debugging interface (OMPT and OMPD) and the CUDA profiling interface (CUPTI) [19] are examples of facilities provided by the runtime to enable tool support.
- **Indirect measurements** in this last category we consider measurements which deal with system-wide parameters such as memory, network bandwidth, system load. This approach can also be extended to interpreted languages. Similarly, when considering virtualized environments or embedded hardware such interface could be exposed through a serial port (either virtual and physical).

The above non-exhaustive list illustrates the number of inputs an instrumentation chain may have to gather. Given the different interfaces offered by the various sources, transposing the "state" of an application to a unified form becomes a challenge. To do so it has to *encode* what it "sees" through these varying sources in a common intermediate representation.

2.3 Intermediate Representations

A tool interested in the various event sources has to ultimately forward data to the analysis consumers. Intuitively, it must convert from the source to what we describe below as an *intermediate representation*. Note that such representation is far from being fixed. Some approaches could adopt actual events, carefully describing what is observed. Other approaches may simply call functions encoding program state as parameters operating the conversion from the source to the transport layer (see next section) immediately in the instrumentation code. Eventually, such representation

could be even useless in the case of a direct data consumption—directly feeding the analysis at wrapper level.

We propose to develop the intermediate representation idea that can serve to create a common intermediate layer between tools components. As we just evoked, we have seen that events can be either partially or even completely diluted in function of the instrumentation chain construction. However, by preserving the "event" abstraction, it is possible to enable interesting scenarios that are not yet fully exploited. Eventually, the reader should notice that this representation is present in two phases of the instrumentation chain (see Fig. 1), both before and after the transport layer.

2.4 Event Forwarding

Given events extracted and then encoded in an intermediate representation, we consider how his representation is transposed to where it should be processed. At this step, there are also various approaches which depend on data verbosity and analysis cost. It is also here where the most impact on tool's scalability arises, clearly of high importance for tool designers. Consider the following transport layers:

- **In-place instrumentation** is when the analysis is directly done at the event-source level. In this case, the analysis is generally not distributed and must remain lightweight so as not to impact the target program.
- **Post-mortem instrumentation** consists in storing events in a trace (generally in the file-system) for later analysis. The advantage of this approach is that analysis is completely decoupled from the target program and that data can be processed multiple times, for example, to be compared. However, the cost associated with storing events has to be mitigated requiring a careful design of a storage medium commonly called a trace-format. Note that this format is generally different from the intermediate representation mentioned in Sect. 2.3.
- **Online instrumentation** is a compromise between the two previous approaches. It relies on network resources to pass events to third-party processes in charge of performing the analysis. This can be done over a Tree-Based Overlay Network (TBON) to perform, for example, validation or continuous spatial reduction at runtime. Another model consists of forwarding events from the group of instrumented processed directly to the analysis—a model that we explore later in this paper in the context of the MALP [5] performance tool.

The purpose of the transport layer is to forward events to the analyzer. In our instrumentation chain model, it can be seen as a function taking events (at the intermediate representation level) and encoding them in a suitable format. On the analysis side, the event is decoded to the intermediate representation for processing. Note, the trace formats also have to account for meta-data (e.g., active threads, their location, their identifiers, and so on). Once forwarded, data are to be projected to the final analysis.

2.5 Event Analysis

There are as many types of event projection than performance tools. Indeed, it is generally at this step that tools produce views, analysis, and hopefully insights for the end-user. Naturally, some views require more input-data than others. However, as analysis are derived from the same input-set of source-event, it is possible to consider that any analysis presented with a sufficiently verbose input should be able to produce its output. In other words, the difference between a time-line and a profile is the level of data projection operated (and therefore its partial destruction). In practice, a given analysis is generally tied to a single instrumentation layer. This might be one reason that we see less attention to interoperability. One interest is to explore whether it is possible to decouple these two main components of the instrumentation chain, enabling cross-analysis.

2.6 Tools Interoperability

We have seen that despite variability in both methods and implementations, instrumentation chains exhibit a common architecture. Moreover, as instrumentation sources are a finite set and generally similar between performance tools, we believe it is reasonable to pose the question of tools interoperability. Some tools are already connected and able to share the same instrumentation layer, such as provided by Score-P [18]. However, this interoperability is still at the transport layer level—Score-P supports several output formats for profiling (TAU [23], Scalasca [12]) and OTF2 [9] for tracing, as well as exposing an online consumption interface for Periscope [3].

Moreover, there are several trace-format converters that can pass data between tools. This process generally involves rewriting all events to the target's transport layer representation (i.e., trace format). This approach has the drawback of duplicating performance data, while being relatively expensive due to their potentially large trace sizes. The following sections explore an alternative approach by exploiting the intermediate representation itself.

3 Towards a Shared-Representation of Performance Events

As presented in Fig. 1, there are two steps where events are in an intermediate representation. During these steps, events are being either extracted or inserted from and to another interface. This representation does not suffer from the same constraints than the transport one. Indeed, these state events are mostly on the stack as structures or function parameters. Also, this representation does not have to be highly space efficient as it will be used for a very short period of time unlike, for example, a post-mortem trace event. An interesting question is whether we can use this to the benefit of unifying represetations.

3.1 Towards a Tool Network

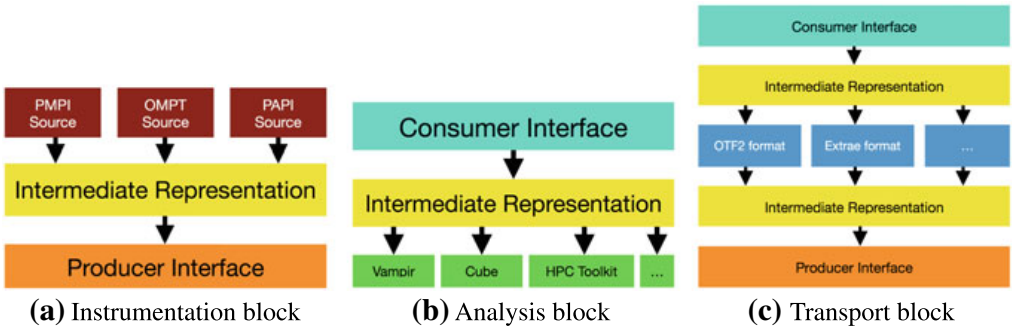
Consider instrumentation chains where this representation is unified. To do so, we suppose a shared intermediate *in-memory* representation sufficiently generic to describe any event type with retro-compatibility. This in-memory aspect is important as it mitigates approach drawbacks. Indeed, if, for instance, we consider the highly generic Pajé [17] trace format, it required ASCII encoding, making it inefficient in terms of storage and parsing.

What we attempt to describe here is close to the consumer/producer interface of any trace format. It can be illustrated with the EARL high-level trace consuming interface proposed in the KOJAK instrumentation chain [25], more recently the notion of Event-Action mapping [15] or ScrubJay [13] (internally relying on Caliper [6]). One point of interest in this related work is the exploration of an event meta-model. Interestingly, in order to make this unified approach possible tools would have to agree on what source events actually represent. If such in-memory event model could be achieved, we could provide trace formats with the same interface. Symmetrically, we could allow instrumentation blocks to produce events in this format.

We propose to model each building block of the instrumentation chain as graph components with inputs and outputs respectively matching these consumer and producer interfaces. As illustrated in Fig. 2, this model would turn support tools into a network of collaborating applications, allowing them to be interfaced at the *consumer interface* level. The following discusses what could be the design of such interface and what are the main constraints envisioned. The approach that we will correlate with our existing implementation of the MALP performance tools which exhibits some of these abstractions.

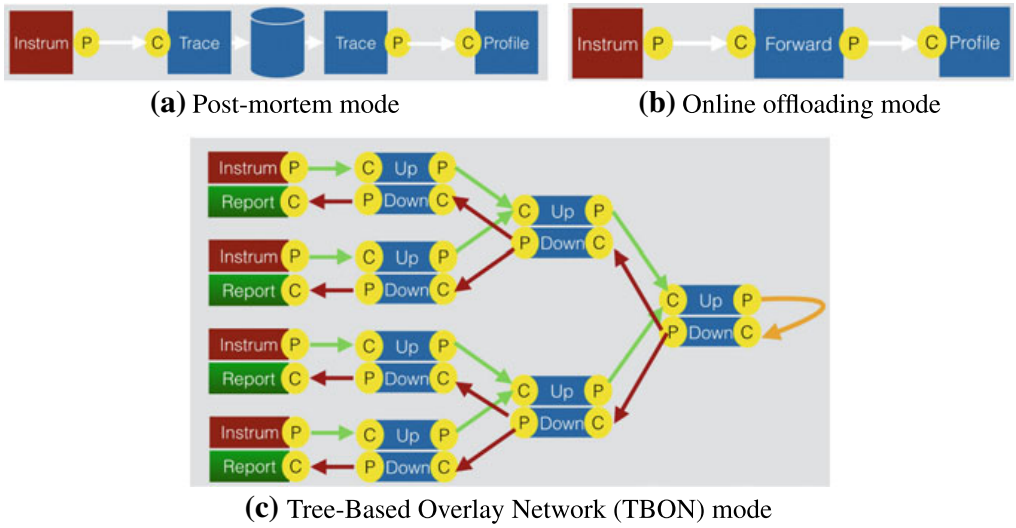
As presented in Fig. 3, the component model supports all the configurations described in Sect. 2.4. Thus, at first cut, this model fits most performance tools indifferently from their data-management policy.

Moreover, as shown in Fig. 4, enabling these common interfaces would allow straightforward interoperability. It would indeed be easy to convert to and from a



(a) Instrumentation block **(b)** Analysis block **(c)** Transport block

Fig. 2 Schematic illustration of *instrumentation chain* building blocks considering the unified producer/consumer model



(a) Post-mortem mode **(b)** Online offloading mode

(c) Tree-Based Overlay Network (TBON) mode

Fig. 3 *Instrumentation chain* configurations obtained from combining building blocks at the consumer level interface. "P" and "C" respectively stand for producer and consumer interfaces

(a) Example of a trace generator **(b)** Implementation of a trace converter

Fig. 4 Alternative building-block usage for interoperability. "P" and "C" respectively stand for producer and consumer interfaces

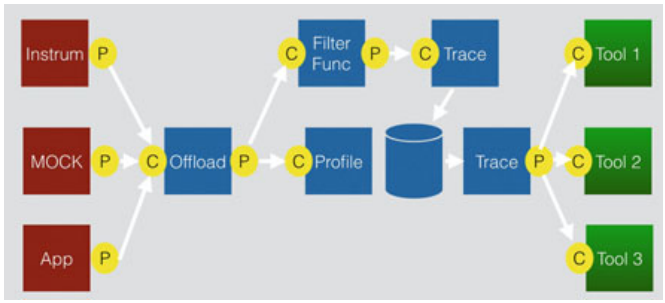


Fig. 5 Custom performance analysis work-flow build around the notion of unified consumer/producer interface. "P" and "C" respectively stand for producer and consumer interfaces

trace-format (Fig. 4), to generate a trace (Fig. 4), or simply to read a trace agnostically from its internal format.

Figure 5 illustrates the level of modularity which could be attained thanks to common interfaces. In this examples, all events are extracted from the source programs using an online coupling (forwarding) mechanism. Then the tool branches, first feeding a selective branch (see *filter func*) sent to the post-mortem trace. Meanwhile, all events are sent to a profile analysis which is less costly to implement and therefore able to handle all events without filtering. Moreover, note how the same trace could be read by different tools. Thanks to this feature users would be able

to see the exact same events instead of having to rerun the instrumented programs will all the bias it supposes (instrumentation overhead). In such configuration, tools could be used in a complementary manner leveraging their various capabilities on the same event records—saving repetitive steps when dealing with monolithic instrumentation chains. This would generalize what is already possible around the Score-P infrastructure.

3.2 The Producer/Consumer Interface

There are certain aspects of a unified interface that outline and motivate important design choices. First, if different tools choose to implement their own instrumentation chain, it is often because they intend to explore new kinds of source events and consequently cannot immediately use what is presently existing. However, with the rise of rich instrumentation interfaces (e.g., OMPT and MPI-T) and considering by itself the non-trivial task of instrumenting a parallel program, important efforts have to be directed to this relatively redundant work. The aim is to convert source events to a representation matching the analysis. instrumenting at event source from known interfaces and with common approaches (see Sect. 2.2). The interface, therefore, has to propose a highly-extensible event-model, both retro-compatible and future-proof. For example, if a new instrumentation interface rises, it should be possible to account for it without impacting tools. This first observation immediately forbids the use of the per-event function (as in OTF2), advocating instead for a single function taking an *event-object* as parameter. In this way, the consumer/producer interface can be fixed and this object later extended to match new requirements.

If we now consider this object representation in the context of a common language such a C, it will certainly be a `struct` pointer passed as a parameter through the interface. However, this struct should be able to represent an arbitrary number of events, each with their respective fields carrying metrics. In C, the `union` can be used to represent multiple data-layouts in the same object, this layout is chosen through the member name. As a consequence, it should be possible to determine the event "kind" which cannot be stored directly inside the union. Furthermore, any event occurring in a parallel application must be associated with meta-data, such as the thread ID where the event was triggered, the timestamp, the MPI rank, and so on. Like the event type, a "common" set of data has to be exposed for every event.

Figure 6 shows how a representation would allow multiple types to be encoded in the same struct without the constraint of the number of types supported thanks to the enum. However, the size of the C struct would be the header size plus the size of the largest enum member. Consequently, some "simple" events may have a footprint larger than they are. This would be a problem in the case of a trace format where storage size is crucial. This extra event footprint is then acceptable in comparison of the flexibility it provides when dealing solely with in-memory representations.

One aspect worthy of further consideration is meta-data. As shown in Fig. 6, events are contextualized to be associated with ranks and threads. However, the

Header Name	**The common event header**
timestamp	When the event happened
tid	Thread ID where the event happened
rank	MPI rank where the event happened
type	Event-type encoding how to look at the enum
Payload name	**An union encoding several types of events**
Collective Operation	**Collective Operation Description**
colltype	Collective operation type
root	Root rank of the collective
srcbuff	Source buffer
...	
P2P Message	**Point to Point Message Description**
p2ptype	Point-to-point operation type
remoterank	Remote MPI process rank
count	Number of elements
comm	Communicator
...	
...	

Fig. 6 Proposed data-model for an unified intermediate event representation

analyzer has to know of these threads and ranks and therefore must be informed of their existence. Most tools manage *shadow* contexts tracking all these identifiers to later retrieve representative state. Instead of this manually tracked state, the source events may simply emit an event for each new handle encountered (e.g., datatype, communicator, and so on). Thanks to logical event order for a given execution stream, the analyzer should be able to reestablish its *shadow* state based on the forwarded event, just as the dedicated instrumentation library would do when intercepting the events. This naturally outlines the need for dedicated meta-data forwarding events for new execution streams and handles and more specifically a common agreement on their semantics. This is probably the most difficult part in the design of a converging performance event meta-model as unlike for source-events, tools are free to choose an arbitrary state tracking approach.

Eventually, if we look at Fig. 5, the producer/consumer interface should be able to build a directed acyclic graph of tools. Producer interfaces should be exposed by name, allowing multiple tools (running in shared memory) to register and consume events. In particular, events should be repeated in each tool registered to the interface, for example, to both profile and trace in Fig. 5. Additionally, the producer interface should callback inside the consumer, as it simplifies data-parallelism which otherwise has to be manually implemented in the consumer. On this note, the single callback approach is very important to avoid the need to register several callbacks with a varying footprint to process events as in current OTF2 API. This idea is not new and has been explored in P^{n}MPI [22]. It has also been pursued in the generic tools interface (GTI) [14] and its transposition to tracing GTI-OTFX [24], together with TBON support. These related research efforts are examples of such modular and plugin-oriented infrastructures which adopt approaches close to the one we want to

motivate in this work. They clearly demonstrate the validity and interest of loading multiple components around an instrumented program. In fact, we see correlations in what is done in the GTI for validation/trace-analysis and MALP for profiling.

4 Practical Illustration with MALP

The Multi-Application Online Profiling (MALP) performance tool [5] has been designed around the concept of a `Generic Events` interface, matching the model we presented in Fig. 6. Our idea when we started the development was to find an alternative to verbose performance traces without sacrificing event verbosity too early in the instrumentation chain. In order to avoid I/O operations, we utilized an approach inspired from the P^{n}MPI [22] virtualization idea and forwarded a stream of events from the instrumented processes to the analysis. These events are similar to those you may find in a performance trace and rely on an intermediate representation. On the analysis side, a "blackboard" system allows plugins to register themselves to event types in order to perform data-reduction on incoming events. Eventually, reduced data are exported in a JSON file and presented in an HTML interface. In this description, one can see how MALP fits in the ideas we described in previous sections.

As presented in Fig. 7, we recently added support for new event-source interfaces such as MPI-T, Alinea MAP time-series, OTF2 trace format, and OMPT inside MALP. All events are then represented as `Generic Events` and forwarded over the network to be reduced inside analysis processes. Therefore, MALP is an illustration of the feasibility of the approach we proposed in this paper. It suggests that if we managed to find new inter-operability opportunities with existing tools we might be able to create bridges simplifying end-user experience. For instance, imagine that Score-P exposes a generic event interface. We would then be able to take advantage of

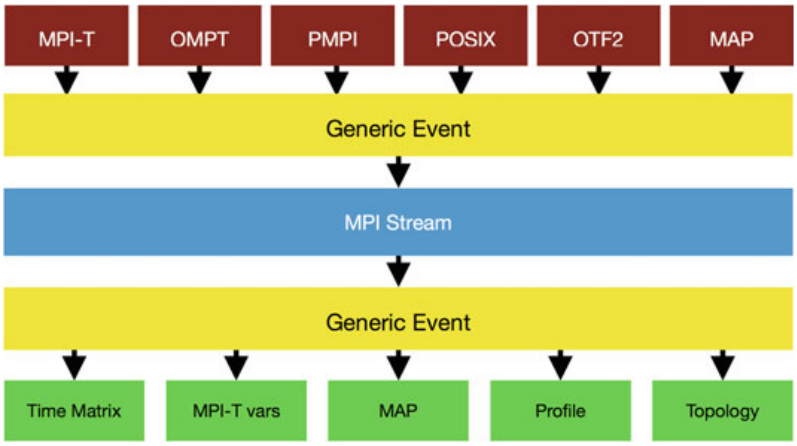


Fig. 7 Schematic overview of source-interfaces supported by the MALP performance tool

a state-of-the-art instrumentation substrate with a rich eco-system (packages, support, user-base), while enabling a robust connection to MALP for performance analysis. From a research and development point of view, it would have saved a lot of complexity in the tool implementation, saving redundant developments as instrumentation approaches are generally similar and well-known (see Sect. 2.2).

5 Conclusion

This paper proposes the concept of an intermediate event representation that that facilitates the development of performance tool workflows. Such workflows are based on a general instrumentation chain model, with components ranging from the instrumentation to the analysis. Our observation of transport-layer aspects of exemplar tools (e.g., post-mortem, in-place, offload, TBON) is that their data semantics is more or less invariant. Thus, we introduced the idea of modular instrumentation chains with an *intermediate representation* as mediation layer. In Sect. 3.1, we discussed the benefits that would arise from such data model, including new performance workflows and possible component reuse. We then suggested a C implementation using a struct of union and an unified handler interface, motivated by what we see as important design choices. Eventually, we discussed the implementation done in the MALP performance tools and showed how it related to our proposed model.

There is an interest in enabling tool inter-operability to simplify tools development and to provide more freedom to end-users who are currently forced to adopt the instrumentation chain for particular analysis tools. Related works such as Score-P, P^nMPI, the GTI, and various trace-converters show that there is a path for collaboration and modularity both inside and between tools. Our hope is that one day we will be able to compose performance tools workflows from a variety of components—TAU [23], Score-P, Paraver [21], HPCToolkit [1], Cube [11], MALP [5]. We believe that performance tools are currently not very far from being capable of this, in fact, a simple *in-memory* intermediate layer approaching the one we described in this paper for MALP would be sufficient. In the line of callback oriented instrumentation layers being developed for OpenMP and MPI, we are confident that infrastructures implementing this semantic will naturally overcome monolithic instrumentation chains.

6 Future Work

The MPI-T and OMPT source-events and simple analysis were implemented in MALP. A point which also needs attention is backtrace representation which requires dedicated storage and possibly complex in-place analysis to attach a given call-stack to events. We plan to explore the possibility of a unified space-efficient backtrace abstraction fitting in our proposed intermediate representation.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: Hpctoolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**(6), 685–701 (2010). https://doi.org/10.1002/cpe.1553
2. Ajima, Y., Inoue, T., Hiramoto, S., Uno, S., Sumimoto, S., Miura, K., Shida, N., Kawashima, T., Okamoto, T., Moriyama, O., Ikeda, Y., Tabata, T., Yoshikawa, T., Seki, K., Shimizu, T.: Tofu Interconnect 2: System-on-Chip Integration of High-Performance Interconnect, pp. 498–507. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_35
3. Benedict, S., Petkov, V., Gerndt, M.: PERISCOPE: An Online-Based Distributed Performance Analysis Tool, pp. 1–16. Springer, Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-11261-4_1
4. Besnard, J.B., Malony, A., Shende, S., Pérache, M., Carribault, P., Jaeger, J.: An mpi halo-cell implementation for zero-copy abstraction. In: Proceedings of the 22Nd European MPI Users' Group Meeting, EuroMPI 2015, pp. 3:1–3:9. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2802658.2802669
5. Besnard, J.B., Pérache, M., Jalby, W.: Event streaming for online performance measurements reduction. In: 2013 42nd International Conference on Parallel Processing, pp. 985–994 (2013). https://doi.org/10.1109/ICPP.2013.117
6. Böhme, D., Gamblin, T., Beckingsale, D., Bremer, P., Giménez, A., LeGendre, M.P., Pearce, O., Schulz, M.: Caliper: performance introspection for HPC software stacks. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13–18, 2016, pp. 550–560 (2016). https://doi.org/10.1109/SC.2016.46
7. Derradji, S., Palfer-Sollier, T., Panziera, J.P., Poudes, A., Atos, F.W.: The bxi interconnect architecture. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 18–25 (2015). https://doi.org/10.1109/HOTI.2015.15
8. Eichenberger, A.E., Mellor-Crummey, J., Schulz, M., Wong, M., Copty, N., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis, pp. 171–185. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_13
9. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open trace format 2: the next generation of scalable trace formats and support libraries. PARCO **22**, 481–490 (2011)
10. Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., Yang, G.: The sunway taihulight supercomputer: system and applications. Sci. China Inf. Sci. **59**(7), 072,001 (2016). https://doi.org/10.1007/s11432-016-5588-7
11. Geimer, M., Kuhlmann, B., Pulatova, F., Wolf, F., Wylie, B.J.N.: Scalable collation and presentation of call-path profile data with cube. In: Parallel Computing: Architectures, Algorithms and Applications: Proceedings Parallel Computing (ParCo07, Jlich/Aachen, pp. 645–652. IOS Press
12. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurr. Comput. Pract. Exp. **22**(6), 702–719 (2010). https://doi.org/10.1002/cpe.1556
13. Giménez, A., Gamblin, T., Bhatele, A., Wood, C., Shoga, K., Marathe, A., Bremer, P.T., Hamann, B., Schulz, M.: Scrubjay: deriving knowledge from the disarray of hpc performance data. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, pp. 35:1–35:12. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3126908.3126935

14. Hilbrich, T., Müller, M.S., de Supinski, B.R., Schulz, M., Nagel, W.E.: Gti: a generic tools infrastructure for event-based tools in parallel systems. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp. 1364–1375 (2012). https://doi.org/10.1109/IPDPS.2012.123
15. Hilbrich, T., Schulz, M., Brunst, H., Protze, J., de Supinski, B.R., Müller, M.S.: Event-Action Mappings for Parallel Tools Infrastructures, pp. 43–54. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-0_4
16. Islam, T., Mohror, K., Schulz, M.: Exploring the capabilities of the new MPI_T interface. In: Proceedings of the 21st European MPI Users' Group Meeting, EuroMPI/ASIA 2014, pp. 91:91–91:96. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2642769.2642781
17. de Kergommeaux, J.C., de Oliveira Stein, B.: Pajé: An Extensible Environment for Visualizing Multi-threaded Programs Executions, pp. 133–140. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-44520-X_17
18. Knüpfer, A., Rössel, C., Mey, D.a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir, pp. 79–91. Springer, Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-31476-6_7
19. Malony, A.D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., Dietrich, R., Poole, D., Lamb, C.: Parallel performance measurement of heterogeneous parallel systems with gpus. In: 2011 International Conference on Parallel Processing, pp. 176–185 (2011). https://doi.org/10.1109/ICPP.2011.71
20. Mohr, B., Malony, A.D., Shende, S., Wolf, F., et al.: Towards a performance tool interface for openmp: an approach based on directive rewriting. In: Proceedings of the Third Workshop on OpenMP (EWOMP01) (2001)
21. Pillet, V., Pillet, V., Labarta, J., Cortes, T., Cortes, T., Girona, S., Girona, S., Computadors, D.D.D.: Paraver: a tool to visualize and analyze parallel code. Technical report, In WoTUG-18 (1995)
22. Schulz, M., de Supinski, B.R.: PNMPI tools: A whole lot greater than the sum of their parts. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC 2007, pp. 30:1–30:10. ACM, New York, NY, USA (2007). https://doi.org/10.1145/1362622.1362663
23. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006). https://doi.org/10.1177/1094342006064482
24. Wagner, M., Hilbrich, T., Brunst, H.: Online performance analysis: an event-based workflow design towards exascale. In: 2014 IEEE International Conference on High Performance Computing and Communications, 2014 IEEE 6th International Symposium on Cyberspace Safety and Security, 2014 IEEE 11th International Conference on Embedded Software and System (HPCC,CSS,ICESS), pp. 839–846 (2014). https://doi.org/10.1109/HPCC.2014.145
25. Wolf, F., Mohr, B.: EARL—A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs, pp. 503–512. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/BFb0100611