

# Event-Action Mappings for Parallel Tools Infrastructures

Tobias Hilbrich<sup>1,4</sup>(✉), Martin Schulz<sup>2,4</sup>, Holger Brunst<sup>1,4</sup>, Joachim Protze<sup>3,4</sup>,  
Bronis R. de Supinski<sup>2,4</sup>, and Matthias S. Müller<sup>3,4</sup>

<sup>1</sup> Technische Universität Dresden, 01062 Dresden, Germany  
{tobias.hilbrich,holger.brunst}@tu-dresden.de

<sup>2</sup> Lawrence Livermore National Laboratory, Livermore, CA 94551, USA  
{schulzm,bronis}@llnl.gov

<sup>3</sup> RWTH Aachen University, 52056 Aachen, Germany

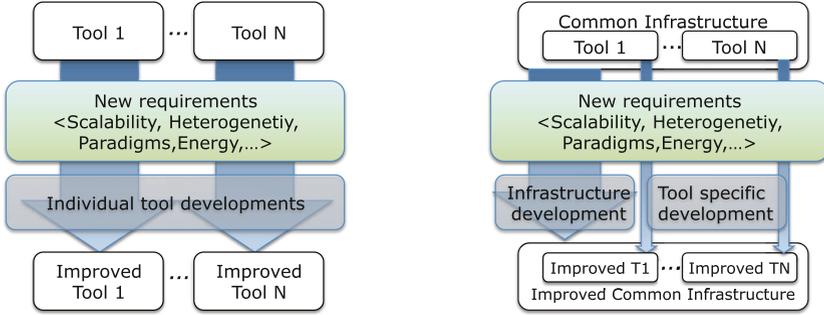
<sup>4</sup> JARA – High-Performance Computing, 52062 Aachen, Germany  
{protze,mueller}@rz.rwth-aachen.de

**Abstract.** The development of applications for High Performance Computing (HPC) systems is a challenging task. Development steps such as optimization, tuning, porting, and debugging often motivate the use of tools, many of which operate at application runtime. Current trends in the HPC community, such as increasing compute core counts and the advent of new programming paradigms challenge the development of applications, as well as the development of runtime tools. Parallel tools infrastructures can help to simplify the development and adaption of runtime tools by reducing development time and increasing applicability. They can provide reusable tool components, communication services, and abstractions for scalable tools, which preserve lessons learned from existing tools projects.

This paper defines an abstraction for a highly integrated infrastructure, which we implement in a prototype that targets MPI applications. Our abstraction enables an incorporation of common tasks such as instrumentation, i.e., observing application behavior, with existing concepts for tool communication, while at the same time enabling scalability. A formal description of our abstraction allows us to highlight its design and to differentiate it from alternatives, so tool developers have a clear understanding of the high-level approach that our infrastructure follows. Existing prototype tools that are based on this infrastructure demonstrate applicability at 1,024 and 16,384 processes respectively.

## 1 Introduction

The development of efficient applications for today's and future High Performance Computing (HPC) systems is a challenging process that involves important steps such as debugging and performance optimization. Tools play a critical role in aiding developers during these steps. Increasing HPC system size, in terms of parallel processing elements, primarily impacts runtime tools, i.e., tools that operate while an application is running on an HPC system. Additionally, deeper



(a) With stand-alone tools each tools requires large development efforts to adapt to new requirements.

(b) With a tools infrastructure, a single development effort can adapt the infrastructure, adaptations of individual tools require a small effort.

**Fig. 1.** Tool development with a common tools infrastructure can drastically simplify the adaption of tools to novel requirements.

system hierarchies for parallelization, caused by hybrid parallelization schemes, e.g., by adding threading models or targeting accelerators such as GPGPUs, impact tools since they must be able to track system execution across these hierarchies. Future systems of an exascale level are likely to add further challenges on the development environment. In order to be widely applicable, portable and future-proof, tools must both be able to handle the increasing scale, adapt to changes in parallelization paradigms, and be capable to deal with limitations on individual systems without losing their portability.

Tool development efforts that implement all components of a tool themselves are challenged by these trends, since addressing them requires tremendous effort from tool developers. Figure 1(a) illustrates this situation. Each tool must individually be adapted for the wide range of new requirements that novel HPC systems and advancements in parallel programming yield. One way to reduce this dramatic development cost, which is finding increased adoption in the tools community, is the use of tool infrastructures and frameworks, which can provide many types of common functionality to tool developers, as well as basic portability layers across platforms and programming paradigms. The use of such common functionality decreases development efforts, since it allows tool developers to reuse adaptations across multiple tools. Figure 1(b) illustrates this situation: The tool infrastructure needs to be adapted for new requirements, tool specific adaptations can require drastically reduced effort then. Benefit increases with the number of tools that adopt an infrastructure.

Following this motivation, we present an abstraction that enables a highly integrated tools infrastructure. The goal of this abstraction is to maximize the amount of common functionality that the infrastructure provides and to minimize tool specific implementation. As Fig. 1(b) illustrates, fulfilling this goal directly reduces the development effort to adapt tools to new requirements.

Naturally, wide adoption of a tools infrastructure does not only depend on the features and capabilities of an infrastructure, but also on a multitude of strategic considerations. Thus, we try not to solely promote a specific infrastructure, but focus on describing an abstraction that enables highly integrated tools development. Existing or novel infrastructures can utilize our concepts, at the same time we provide an open source prototype implementation called GTI [4]. Our contributions include:

- A high-level abstraction that enables quick prototyping and efficient tool development;
- Event-flow definitions that define the meaning of our abstraction for both its usage and implementation; and
- A comparison to existing tools infrastructures to highlight how we achieve a high degree of integration.

Section 2 summarizes related work. We then detail our tools infrastructure abstraction in Sect. 3. Section 4 compares our abstraction to existing infrastructures to highlight the increased degree of integration that we achieve with our abstraction. Finally, we shortly present existing tools that use our abstraction in Sect. 5.

## 2 Related Work

We classify runtime tool developments for HPC systems into: (1) developments that largely use a custom-made implementation; (2) developments that reuse existing components; and (3) developments with parallel tools infrastructures.

Efforts such as the development of Score-P [6] largely employ custom-made source code. Motivations include highest performance requirements, lack of existing components, or redesign efforts. Usually, such developments reuse little existing source code and must cope with trends in HPC by themselves, in order to remain applicable. In practice this can increase development costs and is only suitable for large efforts with significant developer support for maintenance.

Component-based developments reuse existing packages to implement parts of the tool. This can be compared to using support libraries, such as solvers or I/O libraries, in application code. Examples include instrumentation services, wrapper generators, tracing libraries, and stack tracing utilities. Existing components for common types of tool functionality allow developers to reuse lessons learned from other developers. As an example, if a developer improves the scalability of a component, other developers that use the same component can reuse this improvement.

Tool development with parallel tools infrastructures or frameworks, called *tools infrastructures* in the following, provide wider-ranging services for the development of tools and is similar to developing applications in larger frameworks that provide the basic workflow and only require the addition of the actual application components, often in the form of plugins. Examples include P<sup>n</sup>MPI [8] as an infrastructure that combines and connects multiple MPI tools; OCM [10] as

**Table 1.** Tool infrastructure comparison.

	Abstraction	Instrumentation	Topology	Means of Communication	Component Migratability	Component Dependencies
OCM	Event-Action	✓	Centralized		✗	✗
P <sup>n</sup> MPI	Modules	✓	None	MPI	✓	✗
MRNet	Filters	✗	TBON	TCP sockets	✗	✗
STCI	Filters	✗	TBON	MPI subsystem	✗	✗
CBTF	Dataflow	✗	TBON	MRNet (extensible)	✓	✓
Proposed (GTI)	Distrib. Event-Action	✓	Layers	Flexible	✓	✓

an early online tools infrastructure; STCI [2], and MRNet [7] as infrastructures that provide Tree Based Overlay Network (TBON) services; and CBTF [3] and GTI [4] that target a development with a higher degree of integration. TBON services such as provided by STCI, MRNet, CBTF, and GTI are important since they enable tool scalability. The TBON concept uses a hierarchy of processing nodes that allows tools to condense information as it progresses towards the root of the hierarchy. Aggregations and filters on all hierarchy layers provide a step-by-step means to condense information. TBON-based tools such as STAT [1] and Allinea DDT operate for applications that run on close to, or even more than, one million compute cores.

The mentioned infrastructures differ in the depth of their integration and in whether they provide TBON services or not. Table 1 compares these approaches. P<sup>n</sup>MPI and OCM provide no TBON services, which is a common requirement for scalable tools. However, in both cases TBON functionality could be added as an extension leveraging an existing TBON infrastructure. OCM uses event-action mappings that relate to the analysis-hook mappings that we subsequently present, but it considers no distributed and hierarchical processing. The infrastructures STCI [2] and MRNet [7] focus on providing TBON services and allow tool developers to specify *modules* as tool components. The modules run on the TBON nodes to aggregate or filter information. However, these infrastructures lack a deeper integration, e.g., they provide no instrumentation systems, components can not be used on the leaves or the root of the TBON, and no dependency tracking ensures that dependent components are present. Thus, tool developers must provide several common tool components themselves. CBTF and GTI provide such a deep integration and combine it with TBON services. In the following we present the abstraction behind GTI, while our previous publication [4] focused on an implementation of this abstraction. Section 4 compares MRNet—due to its wide usage—with CBTF and GTI in more detail.

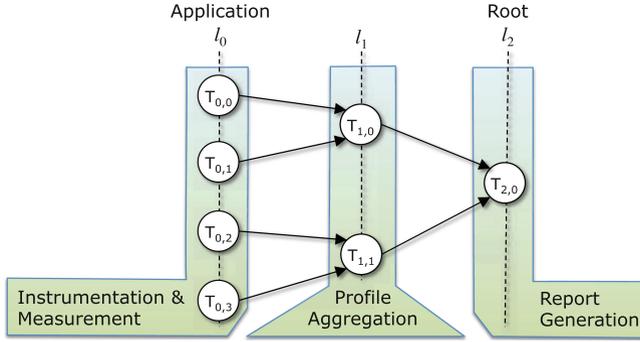


Fig. 2. Illustration of a tool layout and analyses for a profiling tool.

### 3 A Mapping-Based Tools Infrastructure

We base our abstraction for parallel tool development on the notion of:

**Events:** Occurrences of information that the runtime tool must know about, and  
**Analyses:** The types of processing that need to take place upon perceiving events.

The following examples show how both events and analyses map to common steps in well-known types of tools:

- Tracing tools for performance analysis store information on a function invocation (event) into a trace buffer, from where they are processed (analysis);
- A debugging tool retrieves a stack trace (analysis) due to a request from the graphical user interface (event); or
- A runtime correctness tool for MPI analyzes whether send and receive datatypes match (analysis) when a pair of send and receive operations is observed (events).

These examples highlight that events and analyses can represent the activities of a wide range of runtime tools. Following this notion, tool developers can create their tools by solely focusing on the two fundamental questions:

(1) *What activities must the tool implement and (2) which events trigger them?*

The first identifies the analyses of the tool and the second its events and their source. We follow this approach and let developers then specify *mappings* as the relation between events and analyses. This approach allows tool developers to specify the workings of their tool in a high level abstraction. In comparison to existing infrastructure approaches, this allows us to increase the amount of tool functionality that an infrastructure can implement.

We illustrate our abstraction with an example, a simplified profiling tool that creates basic execution profiles and maps them to source code regions. Irrespective of the parallel programming paradigm of the target application,

the tool observes when the application enters or leaves functions. Summarizing this information for all functions of equal name, the tool provides basic profiles that identify the functions in which the application spends most time. Figure 2 illustrates an instance of this tool for four application processes/threads depicted as nodes  $T_{0,0}$ – $T_{0,3}$ . Instrumentation, i.e., observing function enters and leaves, takes place directly on the application, along with adding up the time that each function consumes. The box with *instrumentation & measurement* in the figure illustrates the association of this tool functionality to the application processes. The tool in the figure employs a TBON layout for two purposes: First, the root  $T_{2,0}$  of the TBON can create a summary report that could average the profiling data from all processes. Second, the intermediate layer with  $T_{1,0}$  and  $T_{1,1}$  aids in averaging the profiling data, by applying a step-wise aggregation of the data. Consequently, specific analyses can run on different levels of a TBON layout and a mapping can specify this relation. Additionally, events must travel through the TBON layout from their origin to the analyses that are interested in them.

By specifying the tool functionality in these terms, instead of hard-coding the tool functionality and the tool layout, we allow the infrastructure to handle event instrumentation, of spawning the tool layout, and of handling event communication. This results in the desired deep integration that minimizes the amount of functionality that individual tools must implement.

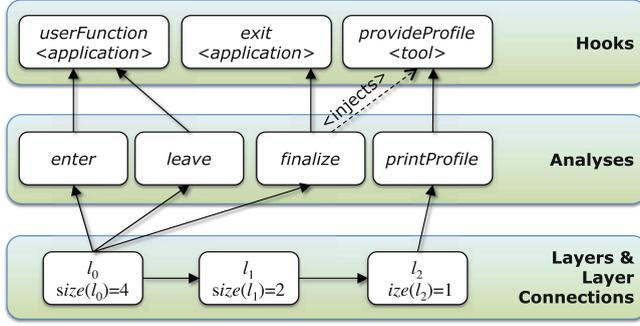
### 3.1 Terms and Abstraction

The tool infrastructure abstraction that we propose extends *events* and *analysis* with *layers*, *places*, *mappings*, *hooks*, and an *event-flow*. We formally define these terms to concisely present our abstraction and to simplify comparison and adoption. Figure 3 showcases how our abstraction represents the aforementioned profiling tool.

**Analyses.** Represent available tool actions. A selection from these analyses makes up the overall functionality of a *tool instance*, i.e., of a running representation of a tool. We use a set  $A$  to specify the available analyses. For the profiling tool, we have  $A = \{\text{enter}, \text{leave}, \text{finalize}, \text{printProfile}\}$ , where enter/leave observes when the application enters/leaves a function, finalize triggers creation and forwarding of the profiling data before the application exits, and printProfile writes the resulting profiling data into a file for investigation.

The example analyses highlight a need for dependencies: Both the enter and the leave analysis must be used together and they require the finalize analysis to forward their results. We provide analysis dependencies  $depends : A \rightarrow \mathcal{P}(A)$  for this purpose<sup>1</sup>. In our example, dependencies include  $depends(\text{enter}) = \{\text{leave}, \text{finalize}\}$  and so forth. If a tool consists of a large number of fine-grained analyses, dependencies ensure that all analyses can operate correctly. This feature facilitates effective component reuse.

<sup>1</sup> The power set  $\mathcal{P}(A)$  of  $A$  is the set of all subsets of  $A$ .



**Fig. 3.** Illustration of our mapping-based abstraction for the profiling tool example from Fig. 2.

**Layers and Places.** Figure 2 illustrates a possible layout for the example tool. Our abstraction defines such layouts with a set of *layers*  $L = \{l_0, l_1, \dots\}$ . We use a specifically marked layer  $l_0$  as the layer that consists of the application processes. We then use the remaining layers to build a hierarchy of layers as in a T BON layout. The layers both specify the hierarchy layers and they enable that distinct layers can execute distinct analyses, as Fig. 2 illustrates. Formally, a *layer tree*  $\mathcal{L} = (L, E_{\mathcal{L}} \subseteq L \times L)$  connects the layers to indicate which layers send information to which other layers. This forwarding represents a *towards-root* direction for scalable tools that apply event aggregation. In our example we have  $L = \{l_0, l_1, l_2\}$  and  $E_{\mathcal{L}} = \{(l_0, l_1), (l_1, l_2)\}$ . The lower third of Fig. 3 illustrates this layer tree.

Further, we use a size function to associate how many *places* each layer uses. Where a place represents an application process/thread or a tool process/thread. Thus,  $T_{1,0}$ ,  $T_{1,1}$ , and  $T_{2,0}$  in Fig. 2 could be tool owned processes and  $T_{0,0}$ – $T_{0,3}$  could be MPI processes.

We use connection rules to create a *tool topology graph*  $\mathcal{T} = (P, E_{\mathcal{T}})$  from a layer tree and the size function, i.e., to create the layout in Fig. 2 from the layer tree in Fig. 3 (bottom). This graph uses the set of all places  $P = \{T_{i,j} : l_i \in L \text{ and } 0 \leq j < \text{size}(l_i)\}$  as its node set. A connection rule then connects places of connected layers to create the set of arcs  $E_{\mathcal{T}} \subseteq P \times P$ .

**Layer-Analysis-Mapping.** A *layer-analysis-mapping* specifies which analyses each layer executes. Formally, the function  $m_{L,A} : L \rightarrow \mathcal{P}(A)$  specifies this mapping. Figure 2 illustrates the association for the example tool, which could use  $m_{L,A}(l_0) = \{\text{enter}, \text{leave}, \text{finalize}\}$ ,  $m_{L,A}(l_1) = \emptyset$ , and  $m_{L,A}(l_2) = \{\text{printProfile}\}$ . Figure 3 illustrates  $m_{L,A}$  with arrows between layers and analyses. Based on the layer-analysis mapping, the tool infrastructure can compute dependent analyses according to *depends*.

**Hooks and Analysis-Hook-Mapping.** Infrastructure provided instrumentation and automatic event communication is a key requirement for the deeply integrated tool development that we target. This requires that the infrastructure is aware of what can be instrumented, which we represent with a set of *hooks*  $H$ . A hook  $h \in H$  is an activity that the infrastructure can instrument in order to observe it. When during tool runtime, an instrumented hook  $h$  is triggered, we create an *event* with information on the observation of hook  $h$ . Examples of hooks could be function calls, APIs such as the profiling interface of MPI, or callback mechanisms.

For the example case, we use  $H = \{\text{userFunction}, \text{exit}, \text{provideProfile}\}$ . The `userFunction` hook observes function calls of the application and could rely on a compiler-based instrumentation of the application. The `exit` hook observes when the application attempts to exit. When it is triggered, it must pass the profiling data to the `printProfile` analysis. To do so, an analysis can simply trigger a hook itself, i.e., analyses can inject events with hooks. The hook `provideProfile` is used in this manner and implements part of the tool functionality, rather than to observe application activities. With the `provideProfile` hook, the `finalize` analysis can inject an event that carries this data. Any analysis that is interested in the data, e.g., `printProfile`, can then observe it. Figure 3 (top) illustrates these hooks and highlights that the first two hooks serve for application activities, while the third hook implements part of the tool functionality.

The above example illustrates a notion of *an analysis is interested in events of a hook*. We use *analysis-hook-mappings*  $m_{A,H} : A \rightarrow \mathcal{P}(H)$  towards this end. The `enter` and `leave` analyses must observe the `userFunction` hook, i.e.,  $m_{A,H}(\text{enter}) = \{\text{userFunction}\}$  and  $m_{A,H}(\text{leave}) = \{\text{userFunction}\}$ . Additionally, the `finalize` analysis must observe the `exit` hook, while the `printProfile` analysis must observe the `provideProfile` hook. Figure 3 illustrates these mappings with arrows between the analyses and hooks.

**Event-Flow.** The previous specifications detail how we can represent a tool in the terms of a deeply integrated abstraction. However, we associated no semantics, yet. For a given tool specification, we use *event-flow* definitions to define the workflow within the tool. These definitions use a mapping  $\text{execute} : P \rightarrow \mathcal{P}(A)$  that represents the final layer-analysis mapping, e.g., to correctly consider module dependencies. The first event-flow definition specifies which analyses a place must trigger when it observes or receives an event for a hook:

**Event-Flow 1 (Trigger)** *A place  $T \in P$  that observes or receives an event for a hook  $h$  must trigger an analysis  $a \in \text{execute}(T)$  exactly if  $h \in m_{A,H}(a)$ .*

The definition requires that when a place receives or observes an event, it triggers any analysis that is mapped to the hook that created the event.

The remaining event flow definitions depend on *communication directions*. As in the previous examples, events can travel towards higher hierarchy layers, e.g., towards a root place. We call this the *primary direction*, which we cover in the following. The opposite direction is called the *broadcast direction* and

distributes information from a root towards application places. The function  $dir : H \rightarrow \{\text{primary}, \text{broadcast}\}$  assigns a communication direction with each hook to specify the direction that events of this hook use. In the example, all hooks use the primary direction.

For the primary communication direction, the hierarchies of places, as illustrated in Fig. 2, determine which events are forwarded to a place. A place  $T$  must perceive events that are directly triggered on  $T$  and any event that a predecessor in the tool topology graph perceives. As an example, in the figure,  $T_{0,0}$  perceives its own events only, whereas  $T_{1,1}$  must perceive events from itself,  $T_{0,2}$ , and  $T_{0,3}$ . To minimize tool overhead, a place perceives only events for hooks to which some of its analyses are mapped or for which a descendant in the tool topology graph must perceive. The relation  $requiresInformation \subseteq P \times H$  formally defines the hooks whose events a place must observe as  $(T, h) \in requiresInformation$  exactly if  $\exists T' \in \{T\} \cup successors(T, \mathcal{T})$  and  $a \in A$  :

$$a \in execute(T'), h \in m_{A,H}(a), \text{ and } dir(h) = \text{primary}.$$

The relation includes pairs of a place and a hook if the place itself or a direct successor of the place executes at least one analysis that is mapped to the respective hook<sup>2</sup>. This relation directly allows us to define what hooks a place must instrument, i.e., observe, and towards which places a place forwards events along the primary communication direction:

**Event-Flow 2 (Observe)** *A place  $T \in P$  must observe (i.e., instrument) a hook  $h \in H$  exactly if  $(T, h) \in requiresInformation$ .*

**Event-Flow 3 (Forward)** *A place  $T \in P$  must forward an event of hook  $h$ , which it observes or receives from another place, to a direct successor  $T' \in P$  exactly if  $(T', h) \in requiresInformation$  (with  $(T, T') \in E_{\mathcal{T}}$ ).*

In the profiling tool example, these definitions require that places  $T_{0,0}$ – $T_{0,3}$  observe the provideProfile hook, even though no analysis mapping assigns these places an analysis that is mapped to this hook. This results from place  $T_{2,0}$  executing the analysis printProfile, which is mapped to provideProfile, while  $T_{2,0}$  is a successor of  $T_{0,0}$ – $T_{0,3}$  in the tool topology graph.

These three event-flow definitions formally define the workings of a tool that follows our abstraction.

### 3.2 Event-Aggregation

Our abstraction allows integration of event aggregation with so called *aggregation analyses*  $A_{agg} \subseteq A$ . These analyses have all properties of regular analyses, but they try to replace their input events with a single/few events. Hooks serve to inject the new events and return values of the aggregation enable the removal of the input events. Adaptions of the above event flow definitions can formalize this notion.

<sup>2</sup>  $successors(T, \mathcal{T})$  is the set of successor places of  $T$  in the tool topology graph  $\mathcal{T}$ .

## 4 Infrastructure Comparison

In the following we compare our abstraction with MRNet [7] as a widely used infrastructure for TBON-based tools and with CBTF [3] as a further highly integrated infrastructure.

An MRNet-based tool consists of back-end code running on the application processes, front-end code running on the root of the TBON, and module code that runs on the intermediate layers of the TBON. The modules are managed by MRNet—much like analyses in our abstractions—and get triggered when events of interest occur. Front-end and back-end code, however, are directly provided by the tool developer and use a specific MRNet API. If a tool developer wants to migrate tool functionality that is implemented as part of the front-end/back-end code to a module, or vice versa, then an adaption of the API in use or a redesign of that code is necessary. This imposes unnecessary restrictions on the use of available pieces of a tool implementation. Our abstraction uses a single concept to provide tool functionality instead, which are analyses. Migrating analyses between any hierarchy layers requires no adaptations as we note in Table 1.

Instrumentation services for MRNet-based tools reside in the back-end code. The infrastructure provides no services for the instrumentation of an application. Our abstraction incorporates instrumentation directly into the infrastructure to provide it to the tool developer directly. Portable instrumentation is often challenging for tools, thus our approach simplifies tool development. A further differentiation is the analysis dependency system that we integrate in our abstraction. MRNet has no such notion and tool developers can consequentially create non-functional tool instances if they are not careful.

The Component Based Tool Framework (CBTF) [3] is closely related to our efforts and is younger than the GTI implementation that realized our abstraction. Few studies on the applicability and scalability of CBTF are available, so we focus on a high-level comparison. As opposed to our event-action mappings, CBTF uses a dataflow-programming paradigm. A CBTF tool consists of components that could be compared to our analyses. Rather than using analysis-hook mappings, CBTF connects components towards a component network. The output of a component then forms the input of connected components. For scalability, CBTF allows component networks to employ MRNet to apply hierarchical aggregations or filters. Both approaches achieve a deeply integrated tool development and should support similar ranges of tools. A key differentiation is that CBTF lacks instrumentation services, which challenges portable tool development. An integration of such functionality would require a mapping of instrumentation sources to CBTF components, much as in our proposed abstraction.

## 5 Tools Enabled by Our Abstraction

The tools infrastructure GTI implements our abstraction and extends it with additions for practicality [4]. This includes a packaging of multiple tool analyses into so called *modules* that provide data sharing between closely related

analyses. The runtime correctness tool MUST [5] and the trace-based online performance analysis prototype OTFX-GTI [9] are both based on GTI and demonstrate applicability to multiple compute systems at up to 16,384 and 1,024 processes respectively. Both tools focus on MPI applications and use a TBON layout in their standard configurations. MUST uses a total of 358 analyses (in 59 GTI modules) and OTFX-GTI uses 168 analyses (in 20 GTI modules). Large numbers of these analyses (276 and 126 respectively) execute on the application layer. This situation highlights that with abstractions such as MRNet, which provides no concept for tool components on the application layer, a large portion of the tool functionality might not be as reusable. Whereas our abstraction enables component reuse on all hierarchy layers of the tool.

## 6 Conclusions

Development of HPC tools must consider increasing system scale and the rise of novel parallel programming paradigms. Developing portable tools that handle these challenges well is time consuming and often requires similar solutions across tools. Tool infrastructures can provide common tool services to wide ranges of tools. Such that development investments into the infrastructure can benefit all tools that utilize them. In such a situation, the total development effort that is needed to adapt tools to new requirements could be drastically reduced.

We extend upon TBON-oriented tools infrastructures with a mapping-based abstraction. This abstraction allows tool developers to implement their overall tool as fine-grained analyses. The abstraction carefully connects the concept of tool hierarchies with a high-level thinking in terms of events and analyses. This targets a deeply integrated development that lets the infrastructure provide as much functionality as possible. If we compare our approach with existing infrastructures we see increased opportunity for component reuse, as well as simplified development for tool developers. The infrastructure CBTF, which follows similar goals as our approach, compares closely, but lacks portable support for instrumentation.

If deeply integrated tools infrastructures receive widespread use, tool development could be drastically simplified. Developers could reuse valuable lessons learned that are embedded in a repository of existing tool components. A large community could then maintain these modules to adapt them to ongoing trends in HPC. Developing a novel tool becomes a reuse of existing modules with an addition of tool specific modules, towards reduced time to solution. Experience with tools that are based on our abstraction are promising and enable tools that operate for 1,024 and 16,384 application processes respectively.

**Acknowledgments.** We thank the ASC Tri-Labs for their friendly support. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-PROC-670945). This work has been supported by the CRESTA project that

has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703.

## References

1. Arnold, D.C., Ahn, D.H., de Supinski, B.R., Lee, G.L., Miller, B.P., Schulz, M.: Stack trace analysis for large scale debugging. In: Proceedings of the 2010 IEEE 21th International Parallel and Distributed Processing Symposium. IPDPS 2007. IEEE Computer Society, Los Alamitos (2007)
2. Buntinas, D., Bosilca, G., Graham, R.L., Vallée, G., Watson, G.R.: A scalable tools communications infrastructure. In: Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications, HPCS 2008, pp. 33–39. IEEE Computer Society, Washington, DC (2008)
3. Galarowicz, J.: Project Final Report: Building a Community Infrastructure for Scalable On-Line Performance Analysis Tools around Open—SpeedShop. Technical report, Krell Institute (2014)
4. 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: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012, pp. 1364–1375, Washington, DC (2012)
5. Hilbrich, T., de Supinski, B.R., Nagel, W.E., Protze, J., Baier, C., Müller, M.S.: Distributed wait state tracking for runtime mpi deadlock detection. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2013, pp. 16:1–16:12. ACM, New York (2013)
6. Knüpfer, A., Rössel, C., an Mey, D., 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. Tools for High Performance Computing 2011, pp. 79–91. Springer, Heidelberg (2012)
7. Roth, P.C., Arnold, D.C., Miller, B.P.: MRNet: A software-based multicast/reduction network for scalable tools. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, SC 2003, ACM, New York (2003)
8. 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 (2007)
9. Wagner, M., Hilbrich, T., Brunst, H.: Online performance analysis: an event-based workflow design towards exascale. In: To appear. In: The 16th IEEE International Conference on High Performance Computing and Communications, HPCC 2014 (2014)
10. Wismüller, R., Trinitis, J., Ludwig, T.: OCM-A monitoring system for interoperable tools. In: Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT 1998, pp. 1–9. ACM Press (1998)