



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Memory Usage Optimizations for Online Event Analysis

T. Hilbrich, J. Protze, M. Wagner, M. Mueller, M.
Schulz, B. de Supinski, W. Nagel

March 20, 2014

EASC2014: Solving Software Challenges for Exascale
Stockholm, Sweden
April 2, 2014 through April 3, 2014

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Memory Usage Optimizations for Online Event Analysis

Tobias Hilbrich¹, Joachim Protze^{2,3}, Michael Wagner¹, Matthias S. Müller^{2,3},
Martin Schulz⁴, Bronis R. de Supinski⁴, and Wolfgang E. Nagel¹

¹ Technische Universität Dresden, D-01062 Dresden, Germany,
{tobias.hilbrich, michael.wagner2, wolfgang.nagel}@tu-dresden.de

² RWTH Aachen University, D-52056 Aachen, Germany,
{protze,mueller}@rz.rwth-aachen.de

³ JARA – High-Performance Computing, D-52062 Aachen, Germany

⁴ Lawrence Livermore National Laboratory, Livermore, CA 94551, USA
{schulzm,bronis}@llnl.gov

Abstract. Tools are essential for application developers and system support personnel during tasks such as performance optimization and debugging of massively parallel applications. An important class are event-based tools that analyze relevant events during the runtime of an application, e.g., function invocations or communication operations. We develop a parallel tools infrastructure that supports both the observation and analysis of application events at runtime. Some analyses—e.g., deadlock detection algorithms—require complex processing and apply to many types of frequently occurring events. For situations where the rate at which an application generates new events exceeds the processing rate of the analysis, we experience tool instability or even failures, e.g., memory exhaustion. Tool infrastructures must provide means to avoid or mitigate such situations. This paper explores two such techniques: first, a heuristic that selects events to receive and process next; second, a *pause* mechanism that temporarily suspends the execution of an application. An application study with applications from the SPEC MPI2007 benchmark suite and the NAS parallel benchmarks evaluates these techniques at up to 16,384 processes and illustrates how they avoid memory exhaustion problems that limited the applicability of a runtime correctness tool in the past.

1 Introduction

High Performance Computing (HPC) architectures feature increasing compute core counts, such as the Sequoia system at the Lawrence Livermore National Laboratory with more than 1.5 million cores. This trend challenges both developers of HPC applications as well as the maintainers of tools that aid these developers. Especially tools that operate at application runtime must provide sufficient scalability to be applicable for application runs with large core counts.

We develop the Generic Tools Infrastructure (GTI) [8] to simplify the development of such scalable runtime tools, in particular tools that analyze large

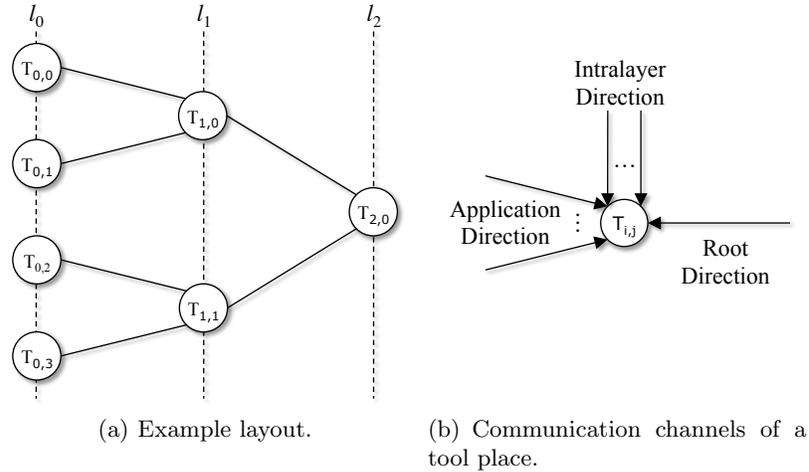


Fig. 1: Illustration of a runtime tool with a TBON layout.

numbers of events (function invocations or communication events) in Message Passing Interface (MPI) [15] applications. Those tools analyze events for use cases such as performance optimization or debugging. Performance analysis tools like Vampir [17] and Scalasca [5] use traces to store events during the runtime of an application and then apply a post-mortem analysis. However, tool exclusive computing resources and a Tree-Based Overlay Network (TBON) abstraction allow tools built upon GTI to analyze such event data already during the runtime of an application; in other words online.

GTI uses extra processes as additional compute resources for the tool itself. These tool processes—called *places* in GTI—can analyze events outside of the critical path of the application. Additionally, GTI organizes places in hierarchy layers that can apply stepwise event analysis (TBON layout), e.g., all application processes provide an event with an integer value and the hierarchy layers sum these events up until the root of the layout retrieves a global sum. This combination of event offloading, analysis outside the critical path, and hierarchic event analysis enables wide ranges of scalable tools. Figure 1(a) illustrates the layout of a GTI tool for four application processes—represented as circles with labels $T_{0,0}$ – $T_{0,3}$ —and three tool places $T_{1,0}$, $T_{1,1}$, and $T_{2,0}$. The lines between the circles indicate the communication channels for events, e.g., the application process $T_{0,0}$ would usually forward events to tool place $T_{1,0}$ for analysis. The tool places can analyze events from the application processes, but also use the communication capabilities of the layout to exchange information with each other.

The GTI-based tool MUST [7] analyzes all communication operations of an application to reveal MPI usage errors. The tool applies a comparatively expensive event analysis as part of its deadlock detection scheme. Thus, the event handling and analysis cost of MUST may exceed the original cost of the

communication operations on the application. Under such a scenario an online event analysis tool like MUST can consume increasing amounts of memory and may fail due to memory exhaustion. Even on a compute system with 24GB of main memory per compute node—shared between 12 cores—MUST repeatedly exhausted memory for one benchmark application in a study of its deadlock detection capabilities [7]. This paper describes and studies two techniques for TBON-based event analysis tools to avoid memory exhaustion problems. Specifically, these techniques avoid storing data into files since the use of the I/O subsystem imposes further challenges at scale [11, 22]. This research may particularly enable new tool workflows for exascale level compute systems that increase challenges around massively parallel I/O system use. An increasing use of online tools could circumvent the challenges that these systems impose onto traditional post-mortem tools.

Section 2 first presents related work and Section 3 then details our assumptions for the communication channels of a TBON and refines our problem statement. Section 4 contains our first technique, a heuristic that provides tool places a communication channel selection that offers a tune-able selection between performance and memory consumption. Section 5 then describes our second technique that temporarily pauses the execution of an application to let a tool “catch up” with its event analysis. We implement these techniques in our tool infrastructure GTI and evaluate it with the runtime MPI correctness tool MUST that previously failed for some SPEC MPI2007 benchmarks. An application study with MPI2007 and the NAS Parallel Benchmarks (NPB) evaluates our techniques at up to 16,384 processes and avoids memory exhaustion in practice (Section 6).

2 Related Work

We describe techniques that overcome deficiencies [7] in the GTI-based tool MUST. These deficiencies result from online event analysis on large event counts where the analysis requires increasing amounts of memory for some series of events. The techniques that we describe apply to tools that handle events in TBONs. Besides MUST, various existing tools and tool infrastructures for high performance computing use TBONs, but often operate on very few events per MPI process. Examples for performance optimization include Periscope [6] that applies an analysis on profiling data for application phases; and TAUoverMRNet [18] that analyses profiling data at user specified execution points or for periodic time intervals. Debugging tools like STAT [1] retrieve call stack information from all processes to represent a global execution state, this data could hardly exhaust memory on any node of the TBON layout. Implementations of our techniques are not bound to GTI, but can also be used to improve the reliability of infrastructures such as MRNet [20], CBTF [13], STCI [4] or SCI [12].

MALP [3] also targets the analysis of large event counts at scale. However, its analyses provide profiling-based performance reports for which a constant amount of memory suffices to handle any event series. Event sizes that increase

with application scale [14] are a related problem that can limit the applicability of an online tool.

File system traces represent an alternative to our techniques that target reduced memory needs during event analysis. Our analyses could store temporary event information into traces to avoid memory exhaustion. Tools such as Vampir [17] and Scalasca [5] successfully employ traces for their performance analysis. However, file systems can impose scalability challenges [11, 22] as well. Various approaches exist to mitigate the effect of this bottleneck, e.g., trace reduction [21], trace compression [19], and I/O forwarding [11].

3 Channels and Memory

Figure 1(a) illustrates a TBON layout. For GTI, application processes and tool places use up to three different communication directions as Figure 1(b) illustrates. The *application* direction allows a place to receive events that travel from the application processes towards the root, the *root* direction allows a place to receive events that travel from the root towards the application processes (usually control and steering), and the *intralayer* direction provides GTI tools a point-to-point communication means within a hierarchy layer. The latter communication direction facilitates tool analyses such as point-to-point message matching for which pure TBON layouts could limit scalability [10]. The arrows in Figure 1(b) illustrate that tool places can probe any communication channel from any of these three communication directions to receive a new event. Each communication channel is bidirectional and has a certain event capacity. That is, if an application process or a tool place sends an event over a channel it can continue its execution before the receiver side handled the event, as long as the capacity of the channel suffices to store the new event. If a communication channel reaches its capacity it will block any subsequent send operations until the receiver side drains some events from the channel. In GTI, this capacity depends on the selection of the communication system, which can either be optimized for bandwidth, offering high capacities, or latency, offering only low capacities.

Analysis algorithms such as point-to-point message matching [10] or deadlock analysis [7], as well as tool infrastructure services such as order preserving event aggregation [9] can consume increasing amounts of memory if newly received events do not satisfy certain conditions. In such scenarios, the channel selection of a tool place can heavily impact the memory consumption of a tool. We illustrate this with MPI point-to-point message matching as an example analysis that searches for pairs of send and receive events with matching message envelopes. If a new send/receive event arrives and no matching receive/send is available, then the analysis stores information on the new event in a matching table, i.e., memory consumption increases. Otherwise, if a new send/receive event completes a pair—a matching receive/send event was present in the matching table—the analysis can remove the latter event from the table. Thus, the memory consumption of the analysis decreases. This analysis enables correctness tools like MUST to implement MPI type matching checks that can reveal incorrect data transfers.

```

MPI_Comm_size(&p)
MPI_Comm_rank(&r)
assert (p%3 == 0)
for i ∈ {1, 2, ..., iterations} do
  switch r%3 do
    case 0
      MPI_Send(to:(r + 1))
    end
    case 1
      MPI_Recv(from:(r - 1))
      MPI_Recv(from:(r + 1))
    end
    case 2
      MPI_Send(to:(r - 1))
    end
  end
end
end

```

(a) Homogeneous.

```

MPI_Comm_size(&p)
MPI_Comm_rank(&r)
for i ∈ {1, 2, ..., iterations} do
  MPI_Isend(to:(r + 1)%p, &req)
  MPI_Recv(from:(r - 1)%p)
  MPI_Wait(&req)
end
end

```

(b) Process behavior differs.

Fig. 2: Communication pattern examples (pseudo code).

As an example, a single tool place could receive events from all application processes in order to match MPI point-to-point operations; in other words, the tool uses a TBON that consists of the application processes and a root. In that case, the single tool place exclusively uses the application communication direction and only needs to select which application process to receive an event from. A round-robin scheme efficiently handles homogeneous applications where all MPI processes execute similar events, such as the example pattern in Figure 2(a). Given that all channels provide an event when probed, the matching table of the point-to-point matching analysis would store at most p operations for a round-robin channel selection. The analysis reaches this peak after it handled an `MPI_Isend` event from each process. At the same time, application processes can exhibit different MPI operations such as in the communication pattern of Figure 2(b). This example⁵ uses process triples where two processes send to the third process, which in turn receives the two send operations. A round-robin scheme would behave poorly for this example since one process in each triple issues twice as many operations than the other processes. The matching table could use up to $iterations \cdot (\frac{p}{3})$ entries for unmatched send operations for the round-robin approach. In practices, functional decomposition and border processes for domain decompositions can cause different MPI operation workloads, such a in the example of Figure 2(b).

In summary, the memory consumption of an analysis depends on the channel selection scheme of the tool places, the communication pattern of the application, the capacity of the communication channels, and the analysis algorithm.

⁵ Uses numbers of processes that are a multiple of three.

The previous example illustrated the impact of the communication pattern. The capacity of a communication channel together with the number of synchronization points in the application also impacts the memory consumption of tool analyses. Once a channel reaches its capacity, no further events can be processed causing the application process to be blocked. This will then indirectly block other processes in their synchronization operations, leading to a cascading effect. Blocked processes can continue their execution once higher hierarchy layers of the tool drain some events from the communication channels.

4 Selection Heuristic

To avoid this kind of impact on application execution, we develop and implement two techniques in GTI. The first one is a heuristic solution to select a communication channel when a place tries to receive a new event. The heuristic targets low-overhead channel selection with a consideration of memory usage. On each tool place, a penalty score for each communication channel represents how often events from this channel increased memory consumption as well as how often the channel failed to provide an event when probed. The score starts at 0 and GTI adds a penalty of α when an event increases memory consumption and a penalty of β when a channel failed to provide an event. Places sort all channels with increasing penalty into a list. When a place probes for a new event it starts with the first channel in the list. Channels that fail to provide an event receive the penalty increase of β and the place advances to the next channel in the list. If a channel provides an event, the place processes the event and any analysis can return feedback whether the event increased their memory consumption via an API. If so, the place applies the penalty of α to the channel that provided the event, otherwise the score remains unchanged. Afterwards, a place reorders the list and probes the first channel in the list again.

This heuristic targets a flexible selection between low memory consumption and low overhead where the values of β and α allow an adaption between the two goals. A selection of $\alpha > 0$ and $\beta = 0$ would only organize channels based on their memory impact and a selection of $\alpha = 0$ and $\beta > 0$ would prefer channels that *usually* provide events as to avoid unsuccessful probes. Additionally, the number of channels along the application direction is usually low and about constant across scales (most TBON-based tools use constant fan-ins across scale), while the number of channels along the intralayer communication direction usually increases with scale. The organization of increasing numbers of channels in a priority list would impact the performance of the selection heuristic at scale. Thus, GTI uses a wildcard receive semantic for the intralayer channel and represents it as a single entry in its channel lists.

5 Application Pause

The channel selection heuristic attempts to receive events that will not increase memory, but bases its selection on past behavior. GTI incorporates a second tech-

nique to avoid memory exhaustion when the heuristic fails to restrict memory usage. GTI-based tools can request an application pause such that application processes will not generate new events. A place should invoke such a request if its memory usage exceeds a threshold σ . Once the application is paused, tool places can process all existing events to reduce their memory usage. For applications that synchronize within some regular interval, any intermediate execution state of the application should have a limited number of open operations (e.g., unmatched communications) for which analyses need to store information. As a result, memory consumption of analyses can decrease towards the memory demand for these open operations, which should be far below the original threshold that caused a place to request an application pause. Once the memory usage of a place that requested an application pause decreases below a second threshold σ' ($\sigma' < \sigma$), it will request that the application should be resumed.

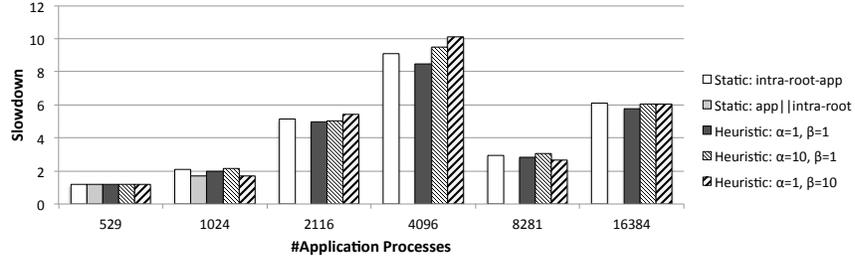
GTI handles this technique with events that any place can inject. These tool specific events travel either along the application or the root communication direction. Four events implement the technique:

- **requestPause:**
 - A tool place injects this event if an analysis exceeds its memory threshold,
 - Tool places forward these events towards the root of the TBON,
- **broadcastPause:**
 - The root of the TBON injects this event when it received one more **requestPause** events than **requestResume** events,
 - The root broadcasts the event towards the application processes,
 - When an application process receives this event it waits until it receives a *broadcastResume* event.
- **requestResume:**
 - Tool places inject this event if they injected a **requestPause** event and their memory usage decreases below σ'
 - Tool places forward these events towards the root of the TBON,
- **broadcastResume:**
 - The root of the TBON injects this event when it received as many **requestResume** events as it received **requestPause** events,
 - The root broadcasts the event towards the application processes.

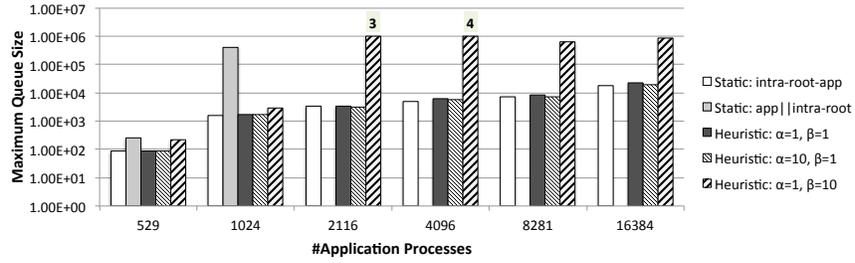
This handling continuously votes for an application pause. The root of the TBON manages the voting and holds an application pause until all places that previously requested a pause agree to resuming the application. The implementation in GTI uses a scalable event aggregation on all levels of the TBON to combine **requestPause** and **requestResume** events.

6 Application Study

We use the Juqueen system at the Forschungszentrum Jülich and the NAS Parallel Benchmarks (NPB) [2] (v3.3-MPI) for our measurements. This Blue Gene/Q

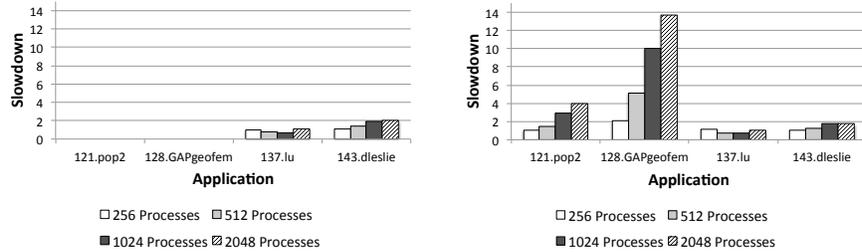


(a) Slowdown.

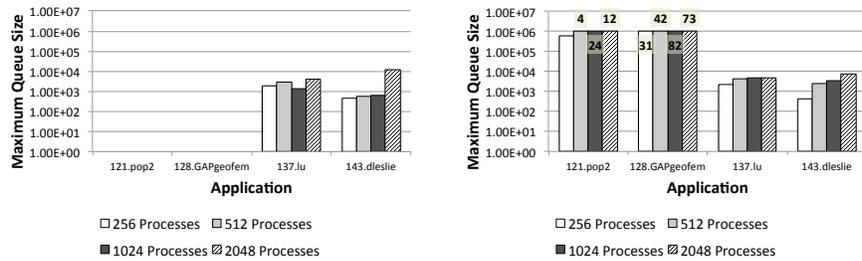


(b) Maximum analysis queue size.

Fig. 3: Channel selection strategy comparison for NPB *sp* on Juqueen.



(a) Slowdown for static selection with *intra-root-app*. (b) Slowdown for our approach with $\alpha = 1, \beta = 1, \sigma = 10^6$, and $\sigma' = \frac{\sigma}{2}$.



(c) Maximum analysis queue size for static selection with *intra-root-app*. (d) Maximum queue size for our approach with $\alpha = 1, \beta = 1, \sigma = 10^6$, and $\sigma' = \frac{\sigma}{2}$.

Fig. 4: Channel selection strategy comparison for MPI2007 on Sierra.

system features 28,672 nodes with 16 cores and 16 GB of main memory each. We implement our techniques in GTI and use the distributed deadlock detection in MUST as an expensive tool analysis that keeps a queue of active MPI operations for deadlock detection. We use the size of this queue to both apply the α penalty of our heuristic and to request an application pause, where we use values of $\sigma = 10^6$ events and $\sigma' = \frac{\sigma}{2}$ events in all runs with our techniques. As kernel we select *sp* since it combines high communication frequency with longer runtime. We use problem size D at up to 4,096 processes and size E at up to 16,384 processes; hence, the dip at 8,192 in Figure 3(a). Figure 3 shows the application slowdown (as runtime with MUST divided by the runtime of a reference run) and the maximum queue size of MUST’s analysis for increasing scales. We compare five different channel selections where we use two static approaches (previous version of GTI) and three selections with our new techniques that differ in their choices for α and β . The static selection *intra-root-app* selects channels in rounds where it first tries to receive an event from the intralayer direction, afterwards—irrespective of whether it received an event—it tries to receive from the root direction, and finally it tries to receive from the application direction. This scheme is a compromise between a performance impact due to unnecessary probes and serving all three directions. The second static selection *app|intra-root* receives events from the application direction whenever possible and only investigates the other directions if no application event is available. This scheme tries to avoid blocked application processes that satisfy their communication channel capacity towards low tool overhead. The selections with our techniques use $\alpha = \beta = 1$ as a compromise between performance and memory usage, $\alpha = 10$ with $\beta = 1$ to prefer lower memory use, and $\alpha = 1$ with $\beta = 10$ to prefer channels that usually provide events towards low tool overhead.

The static selection *app|intra-root* already uses exhaustive amounts of memory at 2,116 processes and causes an out-of-memory crash for this scale. This selection fails to probe communication channels that offer events that would decrease memory usage in practice. A selection with *intra-root-app* provides low queue sizes for the homogeneous communication pattern of *sp*, but issues many irrelevant probes on communication channels. Thus, it causes higher overheads than the heuristic selection with $\alpha = \beta = 1$, especially at 4,096 and 16,384 processes. The latter heuristic selection provides the best results for *sp* overall. It causes marginally higher queue lengths than *intra-root-app* or $\alpha = 1$ with $\beta = 10$, but has the lowest overall slowdown. A selection of $\alpha = 10$ with $\beta = 1$ can provide good performance, e.g., at 1,024 processes, but quickly causes excessive queue lengths that trigger the application pause technique at 2,116 and 4,096 application processes with 3 and 4 pauses respectively. The application pauses along with the increased memory usage increase tool overheads for scales above 1,024 processes.

A second set of experiments uses the Sierra system at the Lawrence Livermore National Laboratory, a Linux cluster with 1,944 nodes of two 6 core Xeon 5660 processors each (24 GB of main memory per node, and a QDR InfiniBand interconnect). We run the *lref* data set of the SPEC MPI2007 [16]

(v2.0) benchmark suite on up to 2,048 cores⁶ on this system to study less homogeneous applications. Particularly, these applications are derived from real world applications and provide a challenging test case. We select the applications *121.pop2*, *128.GAPgeofem*, *137.lu*, and *143.dleslie* for our runs since they particularly stress MUST or even caused memory exhaustion previously. Figures 4(a) and 4(c) present application slowdown and maximum queue length for our previous version of GTI and MUST that uses the static selection *intra-root-app*. The irregular communications in both *121.pop2* and *128.GAPgeofem* cause MUST to exhaust memory even at 256 processes. Figures 4(b) and 4(d) present application slowdowns and maximum queue sizes for our techniques with $\alpha = \beta = 1$. The heuristic suffices to handle *121.pop2* at 256 processes without the application pause technique, i.e., it adapts better than *intra-root-app* to the communication pattern of this application. The application pause technique avoids memory exhaustion for the remaining runs of *121.pop2* and *128.GAPgeofem*. The numbers above/below the bars in Figure 4(d) indicate the number of pauses that each run uses. The figure also highlights that processing all remaining non-application events during an application pause does not cause excessive increases in the maximum queue size for the MPI2007 applications. The highest queue size for these runs was about 5% above σ .

7 Conclusions

We present two techniques to avoid memory exhaustion in online analysis tools for high performance computing. These techniques facilitate use cases where complex tool analysis algorithms are used to process a large numbers of events. Our first technique provides a heuristic that selects a communication channel by using feedback from the tool infrastructure as well as the analysis itself to rank channels in a priority list. A performance study with up to 16,384 application processes shows that this heuristic provides an event selection that causes no memory exhaustion for homogeneous applications and that it reduces tool overhead compared to static selection approaches. Notably, this technique allows the tool to analyze applications such as *121.pop2* at 256 processes where the static selection already exhausts memory.

Our second technique uses the management capabilities of a TBON to pause the execution of all application processes if a tool analysis uses large amounts of memory. Once the application pauses its execution a tool can analyze all events in the system in order to reduce the memory consumption of the analyses. This mechanism handles cases where the heuristic channel selection would exhaust memory otherwise and application studies on two different compute systems show its practicability. Particularly, this technique allows MUST to handle applications for which it previously failed, e.g., *121.pop2* and *128.GAPgeofem*. Thus, our approach increases the applicability of runtime correctness tools such as MUST.

⁶ The *lref* data set operates with up to 2,048 processes (<http://www.spec.org/mpi/docs/faq.html#DataSetL>)

We implement both techniques in the open source tool infrastructure GTI that targets efficient development of online tools. Increased scalability and availability of online tools for tasks such as performance analysis and debugging are an essential step to provide an alternative for trace-based tool workflows, which are increasingly impacted by I/O limitations.

Acknowledgments

We thank the ASC Tri-Labs and the Los Alamos National Laboratory 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-CONF-?????). 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. D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. *International Parallel and Distributed Processing Symposium*, 2007.
2. D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. NAS Parallel Benchmark Results. Technical report, IEEE Parallel and Distributed Technology, 1992.
3. J.-B. Besnard, M. Pérache, and W. Jalby. Event Streaming for Online Performance Measurements Reduction. In *42nd International Conference on Parallel Processing, ICPP '13*, pages 985–994, 2013.
4. D. Buntinas, G. Bosilca, R. L. Graham, G. Vallée, and G. R. Watson. A Scalable Tools Communications Infrastructure. In *Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications, HPCS '08*, pages 33–39, Washington, DC, USA, 2008. IEEE Computer Society.
5. M. Geimer, F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, Apr. 2010.
6. M. Gerdndt, K. Furlinger, and E. Kereku. Periscope: Advanced Techniques for Performance Analysis. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *John von Neumann Institute for Computing Series*. Central Institute for Applied Mathematics, Jülich, Germany, 2005.
7. T. Hilbrich, B. R. de Supinski, W. E. Nagel, J. Protze, C. Baier, and M. S. Müller. Distributed Wait State Tracking for Runtime MPI Deadlock Detection. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, SC '13*, pages 16:1–16:12, New York, NY, USA, 2013. ACM.
8. T. Hilbrich, M. S. Müller, B. R. de Supinski, M. Schulz, and W. E. Nagel. 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 '12*, pages 1364–1375, Washington, DC, USA, 2012. IEEE Computer Society.

9. T. Hilbrich, M. S. Müller, M. Schulz, and B. R. de Supinski. Order Preserving Event Aggregation in TBONs. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 19–28. Springer Berlin Heidelberg, 2011.
10. T. Hilbrich, J. Protze, B. R. de Supinski, M. Schulz, M. S. Müller, and W. E. Nagel. Intralayer Communication for Tree-Based Overlay Networks. In *42nd International Conference on Parallel Processing (ICPP)*, Fourth International Workshop on Parallel Software Tools and Tool Infrastructures, pages 995–1003, Los Alamitos, CA, USA, 2013. IEEE Computer Society Press.
11. T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole. Enabling Event Tracing at Leadership-Class Scale through I/O Forwarding Middleware. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 49–60, New York, NY, USA, 2012. ACM.
12. T. H. Jun and G. R. Watson. Scalable Communication Infrastructure. <http://wiki.eclipse.org/PTP/designs/SCI>. Last visited on 30/04/2013.
13. Krell Institute. The Component Based Tool Infrastructure. <http://sourceforge.net/projects/cbtf/>. Last visited on 19/01/2014.
14. G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons learned at 208K: towards debugging millions of cores. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 26:1–26:9, Piscataway, NJ, USA, 2008. IEEE Press.
15. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0. <http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf>, 2012. Last visited on 27/11/2013.
16. M. S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, H. Feng, and C. Ponder. SPEC MPI2007 – An Application Benchmark Suite for Parallel Systems using MPI. *Concurrency and Computation: Practice and Experience*, 22(2):191–205, 2010.
17. W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
18. A. Nataraj, A. D. Malony, A. Morris, D. C. Arnold, and B. P. Miller. A Framework for Scalable, Parallel Performance Monitoring. *Concurrency and Computation: Practice and Experience*, 22(6):720–735, 2010.
19. M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski. Scalable Compression and Replay of Communication Traces in Massively Parallel Environments. In *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '07, pages 69–70, 2007.
20. P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, New York, NY, USA, 2003. ACM.
21. M. Wagner, A. Knüpfer, and W. E. Nagel. Hierarchical Memory Buffering Techniques for an In-Memory Event Tracing Extension to the Open Trace Format 2. In *42nd International Conference on Parallel Processing*, ICPP '13, pages 970–976, 2013.
22. B. J. N. Wylie, M. Geimer, B. Mohr, D. Böhme, Z. Szebenyi, and F. Wolf. Large-Scale Performance Analysis Of Sweep3D with the Scalasca Toolset. *Parallel Processing Letters*, 20(04):397–414, 2010.