Automatic Adaption of the Sampling Frequency for Detailed Performance Analysis

Michael Wagner*† and Andreas Knüpfer†
*Barcelona Supercomputing Center (BSC), 08034 Barcelona, Spain
†Center for Information Services and High Performance Computing (ZIH), 01062 Dresden, Germany
Email: michael.wagner@bsc.es

***Abstract*—One of the most urgent challenges in event based performance analysis is the enormous amount of collected data. Combining event tracing and periodic sampling has been a successful approach to allow a detailed event-based recording of MPI communication and a coarse recording of the remaining application with periodic sampling. In this paper, we present a novel approach to automatically adapt the sampling frequency during runtime to the given amount of buffer space, releasing users to find an appropriate sampling frequency themselves. This way, the entire measurement can be kept within a single memory buffer, which avoids disruptive intermediate memory buffer flushes, excessive data volumes, and measurement delays due to slow file system interaction. We describe our approach to sort and store samples based on their order of occurrence in an hierarchical array based on powers of two. Furthermore, we evaluate the feasibility as well as the overhead of the approach with the prototype implementation OTFX based on the Open Trace Format 2, a state-of-the-art Open Source event trace library used by the performance analysis tools Vampir, Scalasca, and Tau.**

I. Introduction

High performance computing (HPC) systems yield tremendous computational resources, which comes, however, with more and more complexity, as well. Today's leading edge HPC systems comprise millions of processing elements accompanied by specialized vendor hardware, networks, and heterogeneous accelerators [1]. Building on top of these architectures there is a variety of different parallel programming models such as message passing (MPI), threading and tasking (OpenMP), one-sided communication (PGAS), and architecture specific models like interfaces to incorporate hardware accelerators such as GPUs. Hence, supporting tools have become an inevitable part of application development to utilize those highly parallel and complex systems.

Performance analysis tools assist developers not only in identifying performance issues within their applications but also in understanding their behavior on complex heterogeneous systems. Profiling and tracing form the two main approaches in performance analysis. While profiling gathers aggregated information about different performance metrics, tracing records runtime events or samples together with a precise time stamp and further specific metrics.

Profiling with its nature of summarization decreases the amount of data that needs to be stored during runtime. However, profiles may lack essential information and hide dynamically occurring effects. In contrast, tracing keeps each event or sample of a parallel application in detail. Thus, it allows capturing the dynamic interaction between concurrent processing elements and enables the identification of outliers from the regular behavior.

While individual events and samples are rather small, event-based tracing frequently results in huge data volumes. In fact, the large amount of collected data, in particular, for massively parallel or long running applications is one of the most urgent challenges in event-based performance analysis. The combination of event-based tracing and periodic sampling has been a successful approach to allow a detailed and accurate event-based recording of MPI communication and a coarse recording of the remaining application with periodic sampling to reduce the overall data volume. In this paper, we present a novel approach to automatically adapt the sampling frequency during runtime to the given amount of buffer space, releasing users to find an appropriate sampling frequency themselves. This way, the entire measurement can be kept within a single memory buffer, which avoids disruptive intermediate memory buffer flushes, excessive data volumes, and measurement delays due to slow file system interaction. We describe our approach to sort and store samples based on their order of occurrence in an hierarchical array based on powers of two. Furthermore, we evaluate the feasibility as well as the overhead of the approach with the prototype implementation OTFX based on the Open Trace Format 2, a state-of-the-art Open Source event trace library used by the performance analysis tools Vampir, Scalasca, and Tau. First, the number of resulting trace files limits scalability since the collected data is usually stored in one file per processing element. While HPC parallel file systems are highly optimized for data throughput, the simultaneous creation of hundreds of thousands or even millions of event tracing files overwhelms any parallel file system. Second, the aggregated size of the resulting trace files quickly swallows up storage capacities and overstrains analysis capabilities. Third, the bias caused by intermediate memory buffer flushes. Recorded event data is typically buffered before it is written to the file system to reduce expensive file system interactions. Whenever such an internal memory buffer is exhausted, the content is transferred to the file system; usually in an unsynchronized fashion. Such uncoordinated intermediate memory buffer flushes during a measurement introduce extensive bias and lead to a falsification of the recorded program behavior, which prevents a correct and meaningful analysis.

With this in mind, it becomes a necessity to limit the volume of recorded data during runtime. For a performance analysis that targets applications using MPI as main parallel programming model, existing tools like Score-P [2] allow to restrict the event recording to MPI events, which results in much smaller trace size and, therefore, drastically reduces measurement bias. Recording only MPI communication events can be a valuable method to study communication, for instance, communication patterns and communication inefficiencies. However, the recorded communication events lose their context within the application behavior, for instance, it cannot be determined anymore, which application context triggered an inefficient communication pattern. Thus, the missing context of communication events makes it difficult or even impossible to infer information for performance optimization.

In particular, combining event tracing and periodic sampling has been a common approach among tools like Extrae [3] or Score-P [4]. Both tools allow a detailed event-based recording of MPI communication by instrumenting the MPI library and a coarse recording of the remaining application with periodic sampling. Instead of runtime events, sampling record the current state of an application, usually, with a fixed frequency. The current state of an application that is recorded is typically the current calling context and adequate performance metrics. Recording the state of an application with a fixed frequency provides the inherent benefit that the recorded data rate can be approximated and regulated with the sampling frequency. In addition, sampling introduces less overhead on the measurement, except for applications with particularly low event rates. However, finding an optimal sampling frequency is a virtually impossible task: Using a too low sampling frequency maps the application behavior very coarse. Using a too high sampling frequency results in huge data volumes. In addition, new approaches not only refer to time intervals to record a sample but to intervals of hardware performance counters [5], e.g., with every n-th cache miss or floating point operation. For those approaches it is even more difficult to find useful sampling intervals because their progress is not predictable like a frequency based on time intervals.

Our contribution in this paper is a new approach to automatically adapt the sampling frequency to the available amount of memory buffer space. Our approach allows to start recording in a very high sampling frequency and, whenever the recording memory buffer is exhausted, one half of the already stored samples (every second sample) is discarded and the sampling frequency is halved, as well. This way, the entire measurement can be kept within a single memory buffer, which avoids disruptive intermediate memory buffer flushes, excessive data volumes, and measurement delays due to slow file system interaction. Such an approach releases the user to find an appropriate sampling frequency on its own by automatically adapting the sampling frequency to utilize the recording memory buffer. Even more effective is such an approach for not time-based and, therefore, not predictable sampling intervals, for instance, those based on hardware performance counters.

Our prototype implementation, called OTFX, is an in-memory event tracing extension to the well-established event tracing format and access library OTF2 [6], [7]. For our new approach we build on existing capabilities of OTFX to support a hierarchy based event representation and runtime reduction [8], [9] and extended it to support the new features for hierarchically storing and removing of samples.

In the following section we distinguish our work from other approaches and in Section III, we highlight the tools and libraries providing the starting point of our extensions. In Section IV we describe our approach to allow a low-overhead reduction of already stored samples and discuss aspects of the implementation in Section V. In Section VI we evaluate the overhead of our approach and define a model to verify its feasibility. At the end, we summarize the presented work.

II. Related Work

The Open Trace Format 2 (OTF2) [6] is an event tracing format and access library used by the monitoring environment Score-P [2] and by the trace analyzers Vampir [10], Scalasca [11], and Tau [12]. As it is the origin for the OTFX in-memory extension to OTF2 it shares many similarities with our OTFX prototype, e.g., similar interfaces and event definitions.

Modern event tracing tools provide different strategies to reduce the amount of collected data. For tools that use instrumentation for the entire application, the main focus is on limiting the effects of excessive event generation, which are typical for applications that use a lot of tiny helper functions or class methods like in C++. Score-P can filter function calls based on their occurrence, i.e., the user can specify a value n so that all calls to a function are filtered after this function is called n times [2]. In addition, Stolle et al. introduce a set of adaptive filter techniques that operate during runtime [13]. Score-P also supports a rewind feature that allows to statically or dynamically filter complete program phases, e.g., single iteration steps [2]. Scalasca offers a static code analysis prior to source code instrumentation to exclude functions with a short source code length [14]. In addition, OTFX [15] itself includes techniques to deal with excessive event generation, e.g. filtering functions based on their duration [16].

Next to these filter methods, approaches like compressed complete call graphs CCCG [17] use pattern recognition to accumulate recurring patterns to minimize trace data. While these techniques are capable of reducing the trace data to a nearly constant trace size (depending on the granularity of the aggregation), they may be very time consuming.

In the case of combined instrumentation and sampling methods, the Paraver tool suite uses folding [18] to increase the amount of information for low frequency sampling post-mortem. This approach relies on clustering to detect repeating program phases with similar behavior, e.g. the compute phases within each iteration in iterative codes. Based on the provided clustering, folding than maps multiple compute phases in the same cluster onto each other to use all samples of all similar phases, each with only a few samples, to create a single representative phase with a high density of samples. Furthermore, Paraver's monitoring tool Extrae allows recording uses cluster and spectral analysis to reduce the number of events in traces and, thus, the traces sizes during runtime [19]. This approach forwards performance data and analyzes it on a front-end during runtime. The results are broadcasted back to the monitoring nodes, which use the information to selectively record further events.

Although sharing a similar goal, our approach is distinct from the former mentioned approaches. First, we do not utilize any filtering techniques that, as sophisticated they are, always rely on certain assumptions about the application behavior. Second, techniques relying on filtering of events solve the issue of massive data volumes but not the issues regarding measurement slow down due to the instrumentation of small function calls. Third, while folding allows regaining detail out of a low sampling frequency, this approach is only applicable for iterative codes and highly depended on a successful and accurate clustering of the application phases. To the best of our knowledge there are no approaches that automatically adapt the sampling frequency during runtime to keep the measurement within a single memory buffer.

III. Background

Score-P is the joint measurement infrastructure for the analysis tools Vampir, Scalasca, Periscope, and TAU [2], [10], [11], [12]. It comprises the measurement functionality of these tools into a single infrastructure, which provides a maximum of convenience for users. The Score-P measurement infrastructure allows profiling, event tracing, and online analysis. It contains the code instrumentation functionality and performs the runtime data collection. For event tracing, Score-P uses the Open Trace Format 2 (OTF2) to store the event tracing data for a successive analysis [6]. The Open Trace Format 2 is a highly scalable, memory efficient event trace data format plus support library. It is the new standard trace format for Vampir, Scalasca, and TAU.

OTFX [15] is a prototype implementation based on the Open Trace Format 2 and includes different filters, enhanced encoding techniques, and runtime event reduction to dynamically adapt trace size during runtime to the given memory allocation. The by far most important aspect of OTFX's capabilities is the so-called event reduction. Event reduction dynamically adapts trace size during runtime to the given memory allocation by eliminating events already stored in the memory buffer. Hence, event reduction guarantees that an event trace of arbitrary size fits into a single fixed-size memory buffer. Supported methods for event reduction include a reduction by event class, e.g., functions, communication, or performance metrics; a reduction by the calling depth, i.e., start elimination with the events that have the highest calling depth; and a reduction by the duration of function calls. The different event reduction strategies are discussed in detail in [8]. A new data structure for event storage, called the hierarchical memory buffer, allows applying the event reduction operations with minimal overhead [9].

In the following two sections we detail our efforts to automatically adapt the sampling frequency during runtime to keep the measurement within a single memory buffer based on the previous work in OTFX. In Section IV we illustrate our concept to add an order-based hierarchy to periodic samples. After that, Section V highlights the extensions to the existing hierarchical memory buffer data structure and demonstrates how the order-based hierarchy for periodic samples can be used with the extended hierarchical memory buffer.

IV. Hierarchical Sample Storage

As stated before, combining event tracing and periodic sampling can be useful to combine their benefits. Merging event-based recording of the MPI communication and periodic sampling of the remaining application allows to accurately store the parallel behavior and communication patterns while keeping the overall memory footprint and measurement overhead reasonably small. Moreover, the recording data rate can be approximated and regulated by adapting the sampling frequency. However, applying an optimal sampling frequency from the beginning of the measurement is a virtually impossible without prior knowledge about the application. This results in either a very coarse mapping of the application behavior when setting the sampling frequency too low or in huge data volumes when setting the sampling frequency to high. It becomes even more difficult with the addition of approaches that record samples based on intervals of hardware performance counters, e.g., cache misses or floating point operations, which are less predictable as time-based sampling.

To circumvent the requirement of finding a suitable sampling frequency a priori, we propose a new approach that automatically adapts the sample recording to the given memory quota. Our approach allows to start recording in a very high sampling frequency and whenever the recording memory buffer is exhausted, half the stored samples (every second sample) are discarded and the sampling frequency is halved, as well. This approach requires two main methods. First, being able to change the frequency in which a tool acquires samples and, second, the ability to remove already stored samples from the memory buffer in an efficient manner.

The first requirement is already met by some tools that provide feedback from the memory buffer to the monitor. For instance, the default setup of Score-P as the monitor with OTF2 as the tracing library carrying the buffer. Whenever a buffer is exhausted OTF2 triggers a callback provided by Score-P that decides how to proceed with the buffer and allows adjustments to the measuring process. Our prototype OTFX supports the same callbacks, which can be used to adapt the sampling frequency within Score-P.

The second requirement is much harder to fulfill, at least in an efficient way. OTFX already incorporates the hierarchical memory buffer data structure to allow an efficient removal of events from the memory buffer. Our contribution is to, first, apply an order-based hierarchy to periodic samples that allows to pre-order samples in a binary tree. Second, we extend the capabilities of the hierarchical memory buffer data structure and demonstrate how the order-based hierarchy for periodic samples can be used with the extended hierarchical memory buffer to efficiently remove already stored samples to meet the second requirement.

Since samples contain similar hierarchical information to events, e.g., the calling context, this hierarchy information can be used to store samples similar to events as described in OTFX's methods for event reduction [8]. However, in contrast to event-based trace data, hierarchical ordering in terms of time is much more relevant. Since samples are triggered by the sampling interval rather than application context, each sample by itself represents a randomly chosen application state. In particular, for any two individual samples it is infeasible to

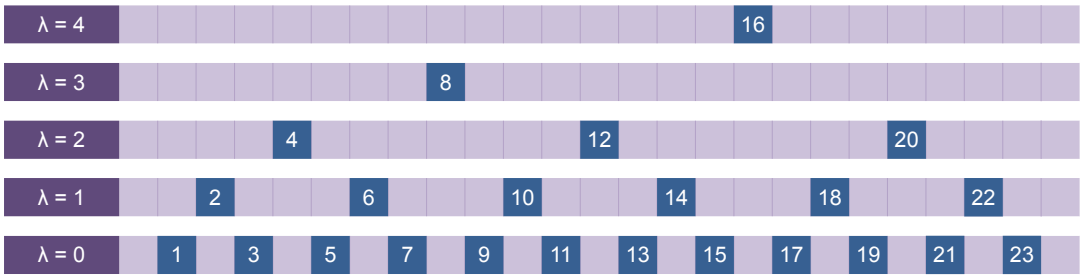


Fig. 1. Distribution of samples based on powers of two with aach level λ containing every $2^{\lambda+1}$th sample.

identify which one of these better represents the actual state of an application. In addition, each sample on its own contains the complete state of an application it represents; while most events are only meaningful in combination with further events, e.g., a code region is only represented with the according enter and leave event together. Thus, removing a fixed ratio of samples based on the order of occurrence, e.g., discarding every n-th sample provides a useful method.

Although a sample by itself does not contain sufficient hierarchy information to presort samples for a reduction based on the ratio, the order of occurrence can be used to distribute the samples to different hierarchy levels to allow an efficient later reduction. The distribution function $\lambda : \mathbb{N} \rightarrow \mathbb{N}$ that maps each sample to a hierarchy level based on the order of occurrence n can be expressed as:

$$\lambda(n) = max\left\{ \; p \in \mathbb{N} \; \middle| \; n \equiv 0 \; mod \; 2^p \; \right\} \tag{1}$$

This way each level λ contains every $2^{\lambda+1}$th sample; comparable to a binary tree representation. Figure 1 illustrates this mapping for the first few samples. The lowest hierarchy level λ_0 contains every 2nd sample, the second lowest hierarchy level holds every 4th sample, and so on, up to the highest hierarchy level $\lambda_{\text{max}} = \lfloor \ln n \rfloor$, which consists of only one sample. Since each level λ contains every $2^{\lambda+1}$th sample, the interval of levels $[\lambda, \infty)$ contains every 2^{λ}th sample. Therefore, the lowest hierarchy level λ_0 contains one half of the samples and the levels $[\lambda_1, \lambda_{\text{max}}]$ contain the other half of the samples.

In a later removal operation all samples on the lowest hierarchy level are discarded and the according memory is released. From that point the sampling frequency is divided in half. Due to the distribution based on powers of two, after the reduction operation the new lowest hierarchy level λ_1 contains one half of the remaining samples and the levels $[\lambda_2, \lambda_{\text{max}}]$ contain the other half of the samples. This way, the reduction operation can be applied iteratively whenever the memory buffer is exhausted; each time discarding every 2nd sample.

However, the maximum notation of the distribution function λ in Equation 1 may be very compute intensive, especially, for large numbers of samples. Since any natural number n can be uniquely decomposed in two-potencies in the form of

$$n = \sum_{p \; \in \; \mathbb{N}} \alpha_p 2^p, with \; \alpha_p \in \{0, 1\}$$

the distribution function λ of Equation 1 can be expressed as:

$$\lambda(n) = min\left\{ \; p \in \mathbb{N} \; \middle| \; n = \sum_{p \; \in \; \mathbb{N}} \alpha_p 2^p \wedge \; \alpha_p = 1 \right\} \tag{2}$$

This minimum notation provides a more efficient way to compute λ because the representation as two-potencies equals the binary representation of integer values.

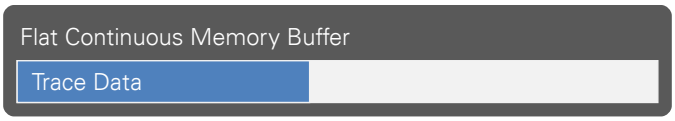
Therefore, the statement in Equation 2 is equal to the number of consecutive trailing zeros in a binary representation of n, which can be calculated with minimal costs using only five basic operations (see Figure 2). The AND comparison with the negation *(input & -input)* extracts the least significant 1 bit from *input*. Multiplying this expression with the De Bruijn sequence[1] 0x077CB531UL results in a unique pattern of bits into the five highest bits for each possible bit position it is multiplied with. When there are no bits set, it returns 0. With a 27-bit right shift this unique sequence can be used to look up *lambda* (i.e. the number of consecutive trailing zeros) in a small table [20]. This multiply and lookup method allows to compute λ within 2-2.5 clock cycles on most modern architectures.

```
unsigned int input;
int lamdba;

static const int lookup[32] =
{
  0, 1, 28, 2, 29, 14, 24, 3,
  30, 22, 20, 15, 25, 17, 4, 8,
  31, 27, 13, 23, 21, 19, 16, 7,
  26, 12, 18, 6, 11, 5, 10, 9
};

lambda = lookup[((input & -input) * 0x077CB531U) >> 27];
```

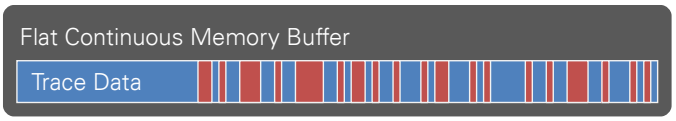
Fig. 2. Finding the number of consecutive trailing zeros in an integer.

This distribution function allows to efficiently order samples based on their occurrence in a way that separates samples to keep from samples to potentially discard. In the following section we extend the hierarchical memory buffer data structure to support this representation of samples and demonstrate how this hierarchical ordering for periodic samples can be used with the extended hierarchical memory buffer to perform the actual removal efficiently.

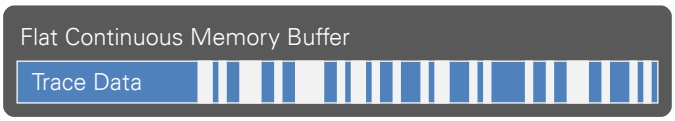
[1]A De Bruijn sequence is a cyclic sequence in which every possible string of length n out of an alphabet A occurs exactly once as a substring. For example, the De Bruijn sequence of $A = \{0, 1\}$ and $n = 2$ is 0011 since this sequence contains all strings of length $n = 2$ (00, 01, 10, 11) as substring exactly once.



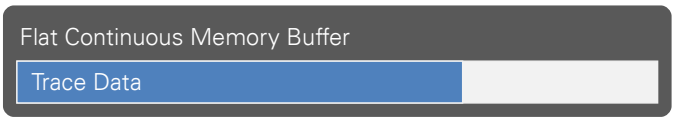
(a) Collecting samples and events until the memory buffer is filled.



(b) The memory buffer is filled and scanned for every second sample.



(c) The memory sections of these samples are marked as free.



(d) The remaining non-free memory is collapsed to one memory section to provide a continuous free memory section at the end for new samples or events.

Fig. 3. Removal of every second sample with a flat continuous memory buffer.

V. Extension of the Hierarchical Memory Buffer

The hierarchical memory buffer is a novel data structure that is designed to store events in a hierarchical representation rather than in a flat continuous way. This allows low overhead access methods to identify and eliminate items that are already stored in the memory buffer. To highlight its capabilities we compare the hierarchical memory buffer to a flat continuous memory buffer that is the default for all common tracing tools and libraries. We exemplarily execute one reduction operation that is triggered whenever the buffer is exhausted and which removes every second sample out of a mixed set of periodic samples and events that describe MPI communication; which is our initial target scenario that combines the benefits of periodic sampling with event-based tracing of the parallel MPI library.

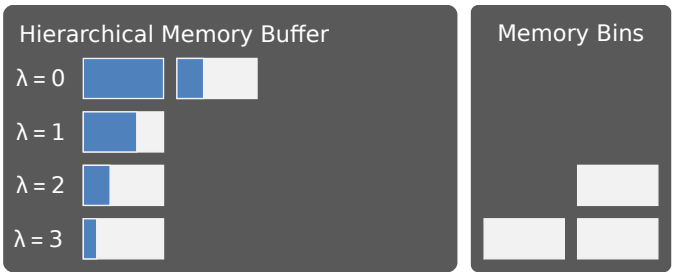
A flat continuous memory buffer stores the recorded samples and events in the order they occur until the memory buffer is exhausted (see Figure 3(a)). Although allowing the identification and elimination of items already stored in the memory buffer, a flat continuous memory buffer introduces an enormous overhead when engaged. Since all events are scattered over the memory buffer, the entire memory buffer needs to be scanned to find all items that match the criterion for reduction (see Figure 3(b)); in this case, every 2nd sample. When all samples matching the reduction criteria are found, they are discarded and the according memory sections are marked as free (see Figure 3(c)). As a result, there are plenty of small free sections scattered over the whole memory buffer. This leaves a highly fragmented memory buffer that cannot be used for writing further samples or events. Thus, all non-free memory sections need to be moved to collapse the fragmented memory buffer to a single continuous memory segment that leaves a continuous free memory section at the end to store further items (see Figure 3(d)) [15].

The computational complexity of the removal operation is in $\mathcal{O}(n)$, with n being the number of stored samples and events. Since a memory buffer, depending on its size, can contain several million items such a removal operation introduces a remarkable overhead when using a traditional flat continuous memory representation.

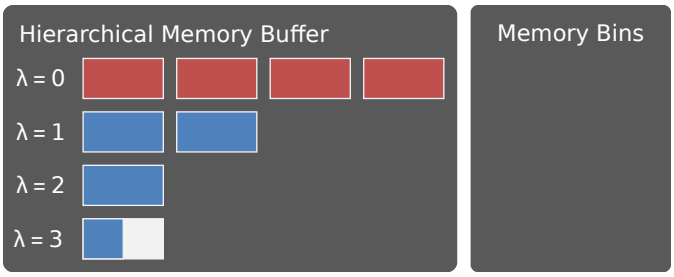
In contrast to a flat continuous memory buffer, the hierarchical memory buffer is organized as a multi-dimensional array, where each *hierarchy dimension* represents one possible hierarchical order with a flexible number of different values within that hierarchical order, called *hierarchy levels*. We extended the existing prototype within OTFX [15] to further distinguish samples from events and to implement the above described ordering of samples based on two-potencies, as described in Figure 2. In the context of the given scenario, one dimension is used to distinguish periodic samples from events describing MPI communication. The second dimension for samples implements the different levels for $\lambda = 0, 1, 2, ...$ to store every sample according to the distribution function. Instead of one huge memory chunk, the total memory allocation for the according memory buffer is divided in plenty of small memory sections, called *memory bins*. These memory bins can be dynamically distributed to any hierarchy level in any dimension. Whenever a sample or event needs to be stored at a certain hierarchy level and there is either no memory bin assigned yet or the current memory bin is exhausted, a free memory bin is distributed to this hierarchy level.

Figure 4 demonstrates the removal of samples with such a hierarchical representation of the memory buffer. For simplification this example considers only the storage and removal of samples. Thus, the according memory buffer's layout is an one-dimensional array. The second dimension distinguishing between samples and MPI events can be imagined as dimension into the depth but is left out to keep the illustration clearly readable.

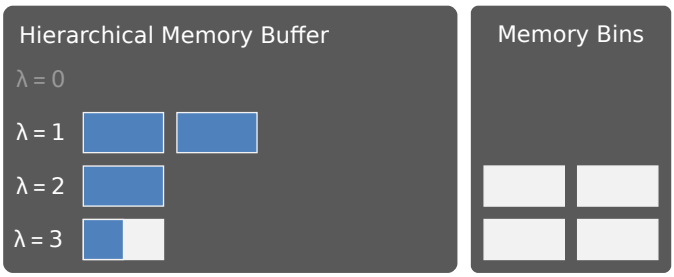
When the first sample needs to be stored, λ-level 0, no memory bin has been assigned to this hierarchy level, so far. Thus, the memory buffer checks if there is a free memory bin available, which is true in this case, and one memory bin is assigned to the hierarchy level $\lambda = 0$, so, the sample can be stored. If a sample or event needs to be stored on a different hierarchy level (a different λ-level or an MPI event), a free memory bin is assigned the same way. The same applies, when on any hierarchy level the current memory bin is exhausted. After some time, this leads to a situation like in Figure 4(a): Five memory bins are assigned to the hierarchy λ-levels $0-3$ and three free memory bins are available. Hence, three additional memory bins can be assigned to the hierarchy levels. After that, all memory bins are assigned and there are no free memory bins available anymore. This leads to the situation in Figure 4(b): A sample needs to be stored at the λ-level 0 but there are no free memory bins available. At this point, the sampling frequency is automatically adapted, i.e. halved. In the monitoring tool the new sampling frequency is set for all further probes. In the hierarchical memory buffer all samples of the lowest λ-level (in this case level λ = *0*) are



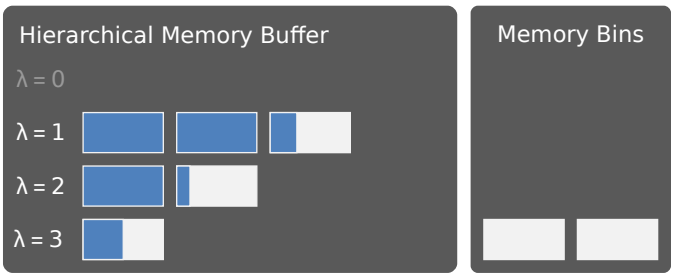
(a) Collecting samples until the memory buffer is filled. Whenever a memory bin is filled a free one is assigned



(b) Current memory bin on level $\lambda = 0$ is filled and there are no free memory bins left, i.e. the memory buffer is full. All samples of the lowest λ-level (in this case level $\lambda = 0$) are grouped together.



(c) All memory bins assigned to level $\lambda = 0$ are revoked and all samples are automatically discarded.



(d) One of the free memory bins is assigned to level $\lambda = 1$ and $\lambda = 2$ to store further samples.

Fig. 4. Storage and removal of already stored samples in the hierarchical memory buffer.

grouped together and all memory bins assigned to level $\lambda = 0$ are revoked and all samples are automatically discarded. This way, the remaining samples (every second sample) match the new sampling frequency, as well. After that, the four revoked memory bins are available again (see Figure 4(c)). Therefore, two of them can be assigned to the λ-levels 1 and 2 and new samples can be stored (see Figure 4(d)). In addition, the hierarchy level $\lambda = 0$ is marked as closed, so, all future samples on this hierarchy level are discarded right away. This allows sorting out future samples with the previous frequency even if the sampling frequency in the monitoring tool has not been adapted. If it is adapted in the monitoring tool, the distribution function λ is simply increased by one, so, all future samples are assigned to the levels $\lambda \geq 1$.

This way, the computational complexity of the removal operation is reduced to be in $\mathcal{O}(b)$, with b being the number of memory bins to revoke. Since, due to the nature of the distribution function, the number of bins to revoke is less than or equal to half the number of total memory bins, the removal operation introduces only a small overhead.

VI. Evaluation

Comparing different trace monitors and libraries is not a trivial task. Event tracing libraries such as OTF2 and OTFX are usually bundled with their according monitoring tools, which blends effects caused by the trace monitor from those caused by the tracing library. In addition, many parameters like timestamps or the execution details deviate in each measurement run. To clearly distinguish the effects of the tracing library and to provide maximum comparability, we do not use the OTF2 and OTFX libraries directly for the measurements. Instead, we generate an OTF2 trace with Score-P and use this trace as a baseline. For the comparison run with OTFX, we replay each application from its baseline trace. This method eliminates the tracing monitor and ensures that both traces (OTF2 and OTFX) use exactly the same input data eliminating the effects of runtime deviations.

The evaluation is based on traces of the molecular dynamics package *Gromacs* [21] including 10,000 iteration blocks; the cloud simulation model system *COSMO-SPECS+FD4* [22] recording two simulation iterations; the computational fluid dynamics solver *Nek5000* [23] using two examples from the source code: *3dbox* that runs 600 iterations on a three-dimensional box with 262,144 grid points and *pipe* that runs 400 iterations on a pipe with 86,400 grid points; and the molecular dynamics simulator *LAMMPS* [24], [25] using three examples from the source code: *colloid* running for 50,000 time steps and simulating interactions between 90,000 atoms on a two-dimensional plane, *rigid* running for 3,000 time steps and simulating interactions between 5,600 atoms in a three-dimensional orthogonal box, and *Lennard-Jones* used for official benchmarks running 1,000 time steps for 32 million atoms on a two-dimensional plane.

A. Runtime and Removal Overhead

To evaluate the overhead of our approach we study two critical properties of the prototype implementation: first, the general runtime overhead introduced by storing the sample and event data in the new format and, second, the delay introduced by the removal operation.

To determine the runtime overhead introduced by the prototype we applied the trace replay described above for all target applications. We compare our OTFX prototype to the state-of-the-art tracing library OTF2, Score-P's standard tracing library. For the overhead measurements both tracing libraries were modified to use up to 2 GiB of memory to keep trace data in main memory. This allows eliminating all effects

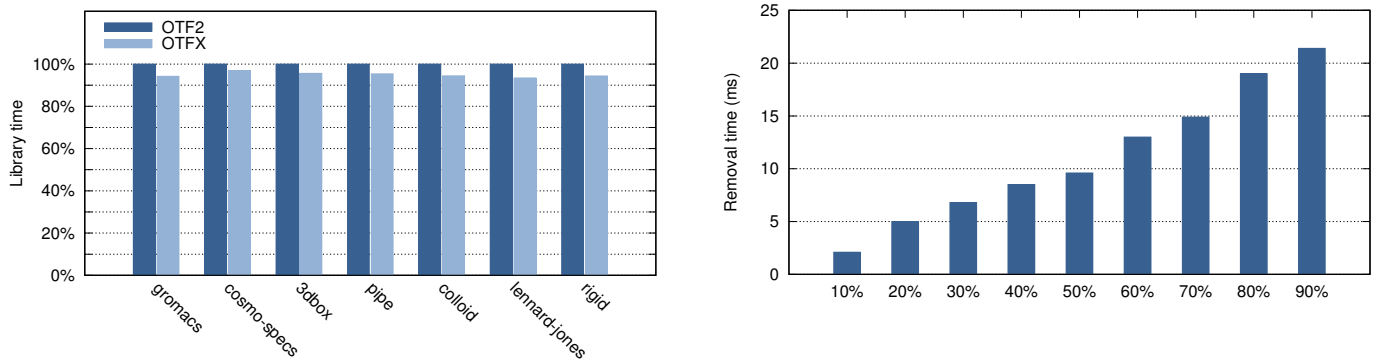


Fig. 5. Overhead OTFX: runtime overhead of OTFX in comparison to OTF2 (left) and overhead of removal operation for a 100 MB memory buffer (right).

of file interaction in OTF2 and the sample removal or any other optimizations in OTFX and evaluating only the event writing and internal data management for both. Next to the benefits described above, the trace replay enables to measure only the overhead caused by the tracing library itself without the overhead introduced by a monitor like Score-P.

Figure 5(left) shows the relative runtime of OTF2 in comparison to our OTFX prototype for the target applications. Since our approach to organize and store sample data according to the above described λ-distribution utilizes features already in place within OTFX, they do not introduce additional overhead to OTFX. The main additional step affecting the storage of sampling data is the computation of the λ-level, which can be achieved within 2-2.5 clock cycles with the above described multiply and lookup method (see Figure 2), i.e. adds only negligible extra time to the general overhead of handling and storing data within the library.

The measurements proved our assumption; within measurement accuracy there have been no differences to the original runtime of OTFX. In comparison to OTF2, our prototype was on average slightly faster than OTF2. However, this can be attributed to other previous optimizations within OTFX and is not a result of our new approach. In general, the library times of OTFX account on average for 7.8 % of the overall measurement runtime. This demonstrates that our prototype suffices our requirement to not introduce additional overhead in the measurement process.

Next to the general runtime overhead, the delay caused by the removal of already stored samples is critical for assessing the feasibility of our approach. Keeping in mind that one of the main ideas is to avoid the bias caused by the delay of memory buffer flushes, the delay of the removal operation should be as small as possible and drastically smaller than a memory buffer flush.

As stated above, the complexity of the removal operation primarily depends on the number of memory bins that are revoked in the hierarchical memory buffer. For that, we use synthetic benchmark to be able to manage and exactly steer the grade of the removal operation. The synthetic benchmark writes 100 MB of data and then a removal is triggered revoking a pre-defined amount of memory bins; in this case from 10 % to 90 % of the distributed memory bins. Figure 5(right) shows the overhead of the removal operation for different grades of the removal (from 10 % to 90 %) for a 100 MB memory buffer. The time spent in the removal operation and, thus, its overhead increases linearly with the percentage of revoked memory bins. For the above described distribution of samples according to the λ-function, a single removal operation revokes a maximum of 50 % of the memory bins equal to half the stored samples. Depending on the number of other stored events, e.g. MPI data, the actual number of revoked memory bins can be smaller. Consequently, the maximum overhead of the removal operation is about 10 ms for a 100 MB memory buffer. Thus, the removal operation creates a minor but noticeable interruption of the application, however, this overhead is drastically smaller than a memory buffer flush which accounts on average for 500-600 ms; not including varying file system response times due to system load or additional delays for file creation at the first memory buffer flush.

B. *Feasibility and Use Cases*

In this section we describe different measurement scenarios in which a user can benefit from our new approach and also its limits in regard to measurement runtime, sampling frequency and application behavior. In this respect, we define a model that considers (a) the initial sampling frequency f_s, (b) the measurement duration t_m, (c) the amount of collected additional information per sample d_s, and (d) the rate r_e of other events, e.g. MPI events.

To define different scenarios we use an initial sampling frequency f_s of 10 kHz, i.e. one sample every 0.1 ms. While tools like Score-P or Extrae use 10 and 50 ms, respectively, as their default sampling interval, much lower frequencies can be used without introducing significant overhead [4]. We consider 0.1 ms a good trade-off between accuracy and overhead. For the amount of additional information per sample we consider two options: a smaller one with two hardware performance counters, e.g. instructions and cycles to compute the rate of instruction per cycle (IPC) that defines the compute intensity, and large one with eight counters as used as default by Extrae. Based on the current implementation in OTF2 (without runtime compression) this results in $d_s \in \{48\,B, 102\,B\}$ containing the timing, the sample record and two or eight counter records [26]. For the amount and frequency of MPI events we use again two options: a low rate of 1 KB/s and

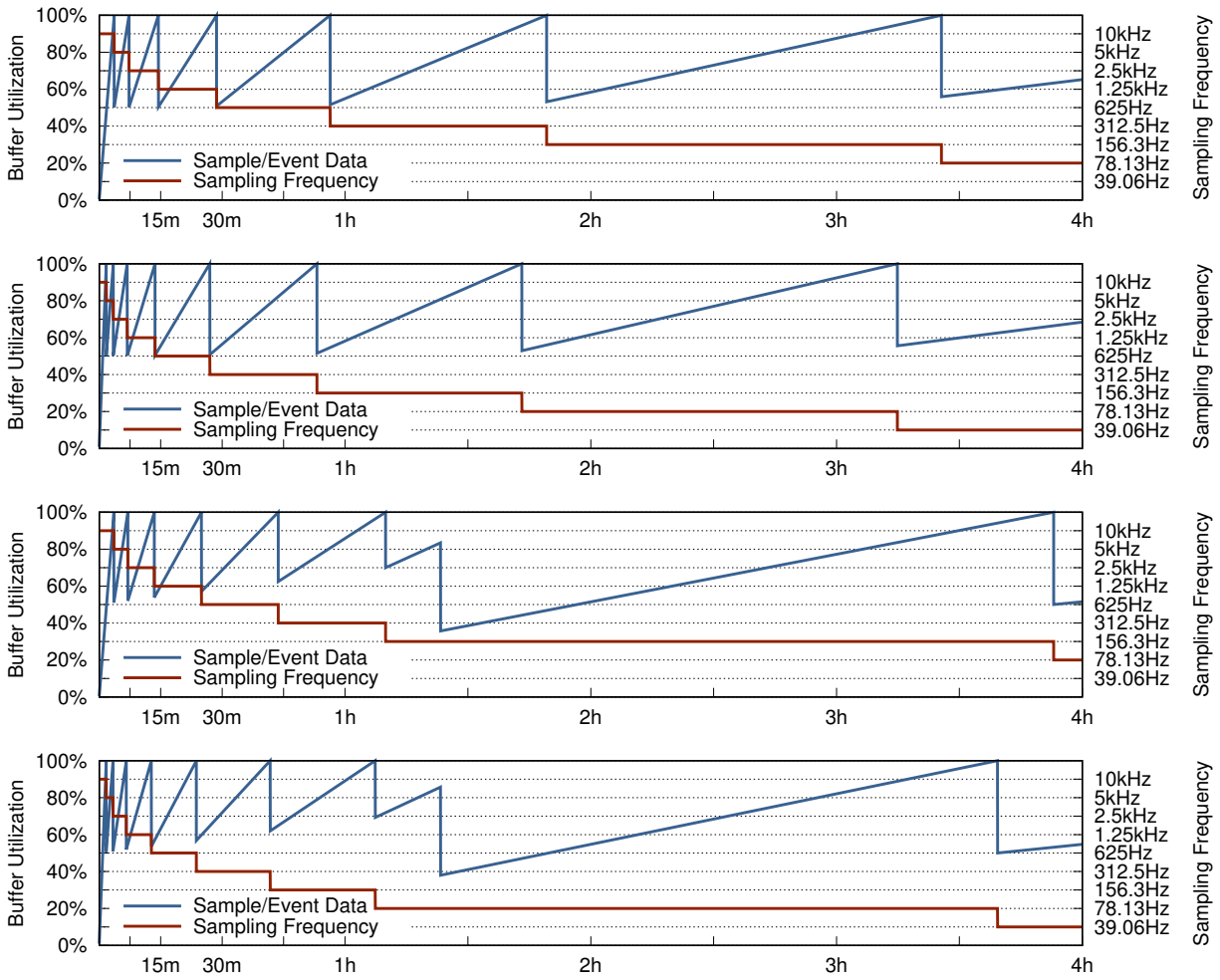


Fig. 6. Model of buffer utilization and sampling frequency for the Scenarios A to D (top to bottom).

higher communication density of 10 KB/s, which are oriented on the behavior of Gromacs and Cosmo-Specs+FD4. Thus, we use $r_e \in \{1\,kB/s, 10\,kB/s\}$. Furthermore, we consider t_m to be variable, i.e. by varying the problem size of a given application. The runtime t_m is used to characterize the length of the measurement. We set a maximum runtime of four hours, which represents most types of applications; at least in a scenario where they are analyzed. In addition, in the following scenarios the hierarchical memory buffer has a capacity of 100 MB and keeps the MPI events until they reach 50 % of the total memory buffer; after that, they are completely removed. This is the current default in OTFX since most analyzers fail on partial communication data.

Based on this, Figure 6 shows the buffer utilization and the automatically adapted sampling frequency over time for the following four scenarios based on the above described model parameters:

- Scenario A: $f_s = 10\,kHz, d_s = 48\,B, r_e = 1\,kB/s$
- Scenario B: $f_s = 10\,kHz, d_s = 102\,B, r_e = 1\,kB/s$
- Scenario C: $f_s = 10\,kHz, d_s = 48\,B, r_e = 10\,kB/s$
- Scenario D: $f_s = 10\,kHz, d_s = 102\,B, r_e = 10\,kB/s$

In all four scenarios the initial sampling frequency drops quite quickly in the first minutes and then is halved with increasing intervals. In the first two scenarios the MPI events can be kept for the entire measurement duration; in the second two scenarios the communication data reaches the 50 % limit at about 1:23h is completely removed from the buffer, i.e. from this point the measurement only keeps the sampling data. In the first and third scenario (with two counters/sample) the sampling frequency is halved seven times during the measurement and reaches 78.125 Hz at the end of the measurement; which is in the order of the 100 Hz default frequency of Score-P. In the second and fourth scenario (with eight counters/sample) the sampling frequency drops almost twice as fast and reaches 39.0625 Hz at the end, which is still about twice the default frequency of Extrae (20 Hz). From the given scenarios it can be inferred that, first, the default sampling frequency of the tools is for the majority of cases too low, which results in a very coarse sampling and reduced level of detail. Second, while this model can also be used to estimate a suitable sampling frequency from the beginning, finding an appropriate sampling frequency is complex, especially for short runtimes. The advantage of the automatic frequency adaption is that it completely frees the user from any estimation or guessing and always selects a suitable sampling frequency.

VII. Conclusion

In this paper, we present a novel approach to automatically adapt the sampling frequency during runtime to the given amount of buffer space. It is applicable for sampling-based performance monitors and hybrid sample/event-based monitors combining a detailed event-based recording of MPI communication and a coarse recording of the remaining application with periodic sampling. With the new approach the entire measurement can be kept within a single memory buffer, which avoids disruptive intermediate memory buffer flushes, excessive data volumes, and measurement delays due to slow file system interaction. It allows recording in a very high sampling frequency and whenever the recording memory buffer is exhausted, half the stored samples (every second sample) are discarded and the sampling frequency is halved, as well. Moreover, our approach releases the user to estimate or guess an appropriate sampling frequency on its own by adjusting automatically to a suitable sampling frequency.

We evaluate our prototype implementation OTFX on the basis of seven application traces from different scientific domains. In comparison to the state-of-the-art tracing library OTF2, our prototype introduces on average 5.1 % less overhead, while the maximum overhead of the removal operation to adapt the sampling frequency is about 10 ms for a 100 MB memory buffer and, thus, drastically smaller than a memory buffer flush which accounts on average for 500-600 ms. Furthermore, we describe a model for our approach in typical usage scenarios that highlights the automatic adaption of the sampling frequency and reveals that the default sampling frequency of common monitors is too low for many use cases resulting in a very coarse sampling and reduced level of detail, while our approach always selects a suitable sampling frequency.

Acknowledgments

This work is supported by the Spanish Ministry of Economy and Competitiveness under contract TIN2015-65316-P.

References

[1] Top500, "Top 500 supercomputer sites," Nov 2016, http://www.top500.org.

[2] A. Knüpfer, C. Rössel, D. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Springer Berlin Heidelberg, 2012, pp. 79–91.

[3] "Extrae instrumentation package," http://tools.bsc.es/extrae.

[4] T. Ilsche, J. Schuchart, R. Schöne, and D. Hackenberg, "Combining Instrumentation and Sampling for Trace-Based Application Performance Analysis," in *Tools for High Performance Computing 2014: Proceedings of the 8th International Workshop on Parallel Tools for High Performance Computing*. Springer International Publishing, 2015, pp. 123–136.

[5] V. M. Weaver, "Linux perf_event Features and Overhead," in *Proceedings of the 2013 FastPath Workshop*, 2013.

[6] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries," in *Applications, Tools and Techniques on the Road to Exascale Computing*, ser. Advances in Parallel Computing, vol. 22, 2012, pp. 481–490.

[7] Virtual Institute – High Productivity Supercomputing (VI-HPS), "Score-P and OTF2 website and download page," May 2015, http://www.vi-hps.org/projects/score-p.

[8] M. Wagner and W. E. Nagel, "Strategies for Real-Time Event Reduction," in *Euro-Par 2012: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7640, pp. 429–438.

[9] 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 *Parallel Processing (ICPP), 2013 42nd International Conference on*, 2013, pp. 970–976.

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

[11] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca Performance Toolset Architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[12] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *International Journal on High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[13] J. Stolle, M. Wagner, J. Doleschal, F. Schmitt, and H. Brunst, "Adaptive Runtime Filtering: Reducing Trace Size and Bias in Event-Based Performance Analysis," in *18th International Conference on Computational Science and Engineering*, 2015, pp. 262–269.

[14] J. Mußler, D. Lorenz, and F. Wolf, "Reducing the Overhead of Direct Application Instrumentation Using Prior Static Analysis," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 65–76.

[15] M. Wagner, A. Knüpfer, and W. E. Nagel, "OTFX: An In-memory Event Tracing Extension to the Open Trace Format 2," in *Algorithms and Architectures for Parallel Processing*. Springer International Publishing, 2016, pp. 3–17.

[16] M. Wagner, J. Doleschal, A. Knüpfer, and W. E. Nagel, "Selective Runtime Monitoring: Non-intrusive Elimination of High-frequency Functions," in *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*, 2014, pp. 295–302.

[17] A. Knüpfer and W. E. Nagel, "Compressible Memory Data Structures for Event-based Trace Analysis," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 359–368, 2006.

[18] H. Servat, G. Llort, J. Gimneza, and J. Labarta, "Detailed Performance Analysis Using Coarse Grain Sampling," in *Euro-Par 2009 - Parallel Processing Workshops*, 2010, pp. 185–198.

[19] G. Llort, J. Gonzalez, H. Servat, J. Gimenez, and J. Labarta, "On-line Detection of Large-scale Parallel Application's Structure," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1–10.

[20] C. E. Leiserson, H. Prokop, and K. H. Randall, "Using de Bruijn Sequences to Index a 1 in a Computer Word," 1998.

[21] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation," *Journal of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, 2008.

[22] M. Lieber, V. Grützun, R. Wolke, M. S. Müller, and W. E. Nagel, "Highly Scalable Dynamic Load Balancing in the Atmospheric Modeling System COSMO-SPECS+FD4," in *Applied Parallel and Scientific Computing*, ser. LNCS, vol. 7133. Springer, 2012, pp. 131–141.

[23] Argonne National Laboratories, "Nek5000 website," May 2015, http://nek5000.mcs.anl.gov.

[24] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.

[25] Sandia National Laboratories, "Lammps website," May 2015, http://lammps.sandia.gov.

[26] "Open Trace Format 2 User Manual," http://www.vi-hps.org/projects/score-p.