Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems

Andi Drebes[1], Antoniu Pop[2,3], Karine Heydemann[1], Albert Cohen[2], and Nathalie Drach-Temam[1]

[1] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, Laboratoire d'Informatique de Paris 6 (LIP6)
F-75005 Paris, France
[2] École Normale Supérieure, Département d'Informatique
45, Rue d'Ulm, 75005 Paris, France
[3] The University of Manchester, School of Computer Science
Oxford Road, Manchester M13 9PL, United Kingdom

Abstract. We present Aftermath, an open source graphical tool designed to assist in the performance debugging process of task-parallel programs by visualizing, filtering and analyzing execution traces interactively. To efficiently exploit increasingly complex and concurrent hardware architectures, both the application and the run-time system that manages task execution must be highly optimized. However, detecting performance anomalies is challenging as bottlenecks can arise directly from the application, the run-time or interaction with the hardware. In Aftermath, key metrics and indicators, such as task duration, state information, hardware performance counter values and data exchanges can be visualized jointly, aggregated and related to the machine's topology. The tool supports traces of up to several gigabytes, with fast and intuitive navigation and on-line generation of new derived metrics. As it has proven invaluable to optimize both OpenStream's run-time and applications, we illustrate Aftermath on genuine cases encountered in the OpenStream project.

Keywords: Performance debugging, trace visualization, task parallelism

1 Introduction

With widely available, inexpensive multi-core processors, the main bottleneck has become parallel software rather than hardware. To help programmers express parallelism, a recent trend in parallel programming languages is to rely on fine-grained tasks whose execution is managed by a run-time system [1–5]. However, such abstractions only help to express the available parallelism but cannot help in detecting and correcting performance issues that arise from the complex interactions with the hardware.

Thus, a major challenge remains: to efficiently exploit the resources of increasingly complex parallel hardware architectures. This entails: (1) ensuring

that enough parallelism is available during execution to exploit all processors; and (2) efficiently managing accesses to shared objects and optimizing data transfers to avoid contention and high-latency memory accesses in NUMA systems.

While some problems can only be addressed in the implementation of the application, many others may be generalized and handled in the run-time system. Performance bottlenecks can therefore occur in any of the two components or their interaction with the operating system or the hardware. The first step towards a solution is thus to characterize and locate the problem, but relying only global statistics or prototyping solutions for different causes until the problem disappears is often impractical due to system complexity. A visual representation of events, system entities and their relationships can provide the insight necessary for an accurate analysis, sorting causes and effects, and tracking application-specific anomalies from inefficiencies in the run-time system's heuristics. The information about these events can be collected and recorded into a trace file which can then be used by a tool for off-line analysis and visualization.

We present Aftermath, a tool for interactive off-line visualization, filtering and analysis of execution traces. Different key metrics and indicators can be displayed jointly, which accelerates the discovery of significant correlations. For more complex relationships, Aftermath offers powerful filtering mechanisms and is able to relate information to the machine's topology. A responsive graphical user interface gives quick access to all of these features, allowing to explore traces rapidly and to control the degree of detail that is needed at each step of the analysis. Aftermath has been designed for task-parallel languages in general, and to analyze the performance of dependent task programs in particular.

In this paper we discuss the features and implementation of Aftermath in the context of a state-of-the-art task-parallel language, OpenStream [2], a dataflow, stream programming extension of OpenMP. Using performance anomalies encountered in the OpenStream project, we illustrate how Aftermath helps programmers debug the performance of applications and run-time systems.

The remainder of this document is organized as follows. In Section 2, we define our requirements for performance analysis and trace visualization. Section 3 presents the interface of Aftermath and its main functionality. We then illustrate, in Section 4, how Aftermath has been used to detect performance anomalies located in user applications. An example for analysis of run-time systems is given in Section 5. Existing tools for trace visualization are summarized in section 6.

2 Requirements for performance analysis

We identified two key scenarios, frequently occurring in the performance debugging process. In the first case, the programmer suspects that there is a performance anomaly or is looking for optimization opportunities, but has not identified any specific issues. Browsing through an execution trace, which we refer to as *trace exploration*, can help building up a hypothesis by identifying sub-optimal program behavior. In the second case, the programmer has already developed

one or more hypotheses and tries to confirm or to refute them. Performance debugging is often an iterative refinement process, alternating between these two situations. An application for trace visualization and analysis should therefore provide data selection and visualization tools that fit both situations.

2.1 Trace visualization

Trace files generally contain two types of information: (1) static information about the execution context, e.g., the machine topology, the number of worker threads, the different tasks or work functions; and (2) dynamic information on execution events, e.g, worker state transitions, communication events and hardware performance counter values. For efficient analysis, basic topological, temporal and relational aspects need to be represented adequately at the same time. The user should be able to:

- Distinguish the activity of different processors and worker threads.
- Observe activity over time and the evolution of metrics.
- Precisely identify the types of events.
- Determine involved entities, e.g., source and destination of data exchanges.

The graphical representation should provide adequate support to make apparent any strong correlations between events. For example, if a performance issue only occurs on specific processors, in specific intervals or after specific events, this behavior should be directly identifiable on the visual representation.

The interactive exploration of traces is an essential aspect that provides a quick overview on the trace data and helps develop a working hypothesis. Navigation along the different dimensions, e.g., changing the interval to be displayed, should therefore be intuitive to the user. With trace files of up to several gigabytes, rendering needs to be sufficiently fast during trace exploration.

2.2 Control over the amount of detail

For exploration of specific aspects or in order to reduce the amount of data to be visualized, it must be possible to filter the information from the trace, such that only relevant information is displayed. The result should be visible immediately when the filter is applied. Filters are also an important tool when testing hypotheses. In order to check if an assumption is correct, the user needs to filter out all situations for which the premise of the hypothesis does not hold. As conditions can be complex, it should be possible to combine filters easily.

However, even with powerful filtering schemes, visual feedback is not always sufficiently precise for a distinct conclusion. In such cases it may be necessary to statistically correlate events, which means that it should be possible to aggregate trace data and display statistical information on event distributions, either presented in separate views or along with the information that was quantified. The latter case might enable the user to draw conclusions on relationships between existing and newly aggregated aspects. If none of the basic statistical counters

alone can provide enough information about a relationship, it is essential to be able to combine them. The user should be guided through this process by a user interface that allows to precisely select which information should be derived and how it should be displayed.

Finally, it must also be possible to obtain detailed information about specific events. This can help detecting outliers or to develop generalized rules from particular situations. For example, the user could select a few corner cases for task duration one after another and then try to figure out the generalized conditions for fast or slow task execution.

3 Aftermath

Aftermath[4] has specifically been designed to meet the requirements outlined in the previous Section, allowing fast, interactive, visual exploration and analysis of traces generated by fine-grained task-parallel applications and their run-time systems, executing on modern many-core architectures. In this section, we give an overview on Aftermath's design and its features, we present the layout of its graphical user interface, the trace format required and we explain how Aftermath can exploit information from application symbol tables and trace annotations.

3.1 Organization of the main user interface

Figure 1 shows Aftermath's main window, composed of five different parts:

1. A *timeline component* placed in the center shows the activity of each of the processors over time (e.g. the different states of the worker threads associated to the processors, evolution of performance counter data collected using the PAPI library [6] and specific discrete events, such as task creations, and communication between workers).
2. A set of *filters* for various basic properties at the left side allows to control what is shown by the timeline component (e.g. only tasks of a specific type, tasks whose execution time is in a certain range, tasks that write to certain NUMA nodes, etc.).
3. The right side contains a group of aggregating, *statistical views* that help quantifying basic information for a user-defined interval from the timeline view (e.g. task duration in a histogram, average parallelism and a communication matrix).
4. The bottom part is reserved for *detailed textual information* about a selected state and the task execution associated to it (e.g. the task and state type, the duration and data-flow-specific information about the producers of the task's input data as well as the consumers of its output data).
5. A set of *generators*, accessible from the menu bar at the top, allows to derive metrics from high-level events or to combine existing statistical counters (e.g. average task duration, number of bytes exchanged between specific

[4] Available under a GNU GPL license at http://www.openstream.info

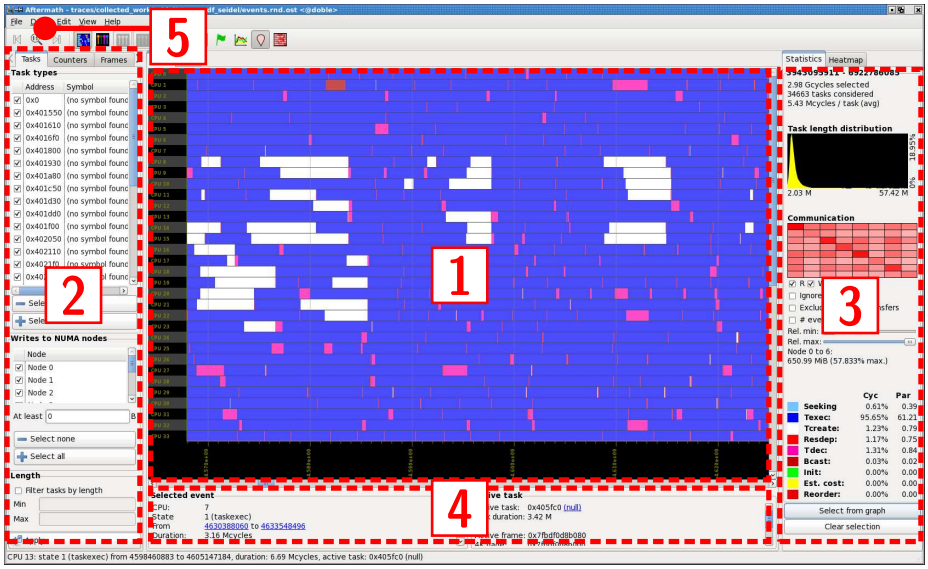


Fig. 1: Aftermath's main window: timeline (1), filters (2), statistics (3), information on selected tasks / events (4) and menu bar for derived metrics (5).

NUMA nodes, ratio of two hardware performance counters, etc.). Selecting the appropriate menu entry opens the corresponding dialog that guides the user through the creation of a derived metric.

Aftermath allows arbitrary zooming and scrolling along the timeline through an intuitive interface. Filters directly affect the information shown in the timeline and the statistical views for the selected portion of the trace in order to provide immediate visual feedback. Rendering was optimized carefully, such that no delays interrupt the user's work-flow. During development of Aftermath, we found that complete traversal even of multi-gigabyte traces only represents a small fraction of the rendering time. Displaying only information that is visible at the selected zoom level reduced the overall delay sufficiently. For example, instead of rendering all the state changes in the timeline, only states that represent a relevant part of the interval defined by a pixel on the screen are shown. For a set of communication events whose communication lines overlap, only one line is drawn. The resulting rendering operations are carried out by the Cairo graphics library [7]. For standard user interface components we have used GTK+ [8].

3.2 Trace format

Aftermath currently only supports its own, native trace format optimized for the OpenStream run-time and OpenStream applications. In addition to generic trace events of task-parallel applications the format also defines several event types for data-flow analysis and OpenStream-specific data structures. As traces can

contain hundreds of thousands of events, trace data is stored in a binary format in order to reduce its size and to avoid long parsing delays when a trace is opened. Further reduction of the file size could be achieved by trace compression.

Trace files are organized as streams of data structures, which can either contain events (i.e., state changes, hardware performance counter values, communication events or discrete events, such as the creation of a task or beginning and end of task execution), topological information about the machine (e.g., which CPUs are associated to the system's NUMA nodes), descriptions of hardware performance counters or information about the location of OpenStream-specific data-flow buffers. Structures can appear in any order, e.g. the trace might contain events of different processors in an interleaved fashion, as long as total order for events is preserved for each processor. This reduces overhead when the trace data is collected and dumped to a file as no time-consuming sorting is necessary.

The format was also designed to contain only few redundancies. Information not directly available in the trace file, but needed for rendering or generation of basic statistics is derived and added to the internal representation when the trace is loaded into main memory.

3.3 Symbol tables and annotations

Further information on tasks can be obtained by loading the symbol table of the binary file of the executed application. Aftermath is able to extract the symbolic names of functions using the standard command-line tools of the GNU/Linux system. When a task is selected in the timeline, its symbol name is looked up and shown in the detailed textual view. A click on the name starts an editor with the corresponding source file and line.

Information that cannot be derived automatically can be provided by user-defined annotations. A double-click on the timeline opens a dialog that lets the user enter an arbitrary text and choose a color for the new annotation. Annotations can be saved independently from the trace file and loaded for further analysis at a later point in time. This is especially useful when analyzing complex traces over a longer period of time.

4 Debugging application performance

The following cases, which we have encountered during development of benchmarks for the OpenStream project, illustrate how Aftermath can be used for performance debugging of task-parallel applications. The first example emphasizes a design problem of an implementation of the *seidel* benchmark, leading to inefficient memory accesses. The two subsequent examples are performance issues solved with Aftermath while implementing the *kmeans* application.

4.1 Seidel: detecting contention on memory controllers

The *seidel* benchmark simulates heat transfer on a surface by performing a 5-point stencil operation on the elements of a 2-dimensional matrix. For parallel

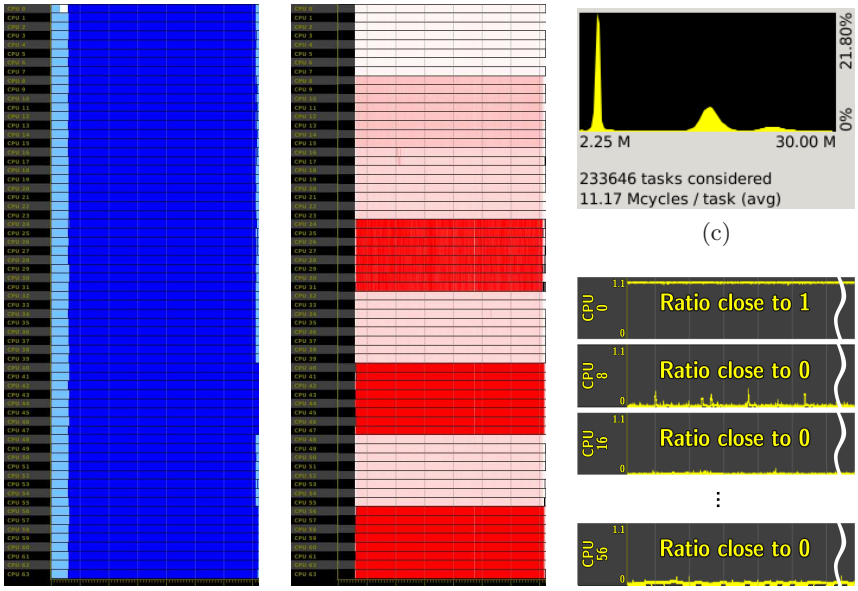


Fig. 2: High-latency memory accesses of *seidel* using a shared matrix: (a) All of the 64 workers are in task execution state (dark blue) for almost the whole execution (b) Heatmap mode indicating a relationship processor → task duration (c) Task duration distribution (d) Ratio of local accesses to the total number of accesses for CPUs 0, 8, 16, and 56.

execution, a globally shared matrix is divided into equally-sized blocks that are each processed by a different task.

The trace in Figure 2a shows that the execution starts with the sequential generation of random data, during which processors are idle (light blue). It is followed by a parallel phase carrying out the actual computations, where all processors remain in task execution state until program termination (dark blue), indicating that there is sufficient parallelism available.

Most tasks should have approximately the same duration because the amount of work is the same, except for a few blocks at the borders of the matrix. However, when selecting the parallel phase to display statistical data about task duration, a performance anomaly becomes apparent. The task duration histogram shows an abnormal distribution: several peaks appear denoting groups of tasks with different execution durations as shown in Figure 2c.

The *heatmap view* of Figure 2b, in which tasks on the timeline are shown with a different intensity of red according to their execution time, suggests that the task duration directly depends on the processor executing the task. The shortest tasks are executed by processors 0 to 8, which are located next to the memory controller of NUMA node 0. Tasks from processors associated to nodes

3, 5 and 7, which are at a distance of two hops from node 0, have the highest duration. The shared matrix is thus located on node 0, which causes memory accesses to be the bottleneck in this application.

The trace file also contains hardware performance counter data for the number of requests to local and remote memory controllers. On the test platform, these are northbridge-wide counters [9] that aggregate accesses of 8 CPUs sharing a memory controller. In the trace file, they have been associated to the CPUs with the smallest identifiers, i.e. CPUs 0, 8, 16, 24, 32, 40, 48 and 56. Aftermath is able to combine these two counters for remote and local accesses to a derived metric representing the ratio between local accesses and the total number of accesses. Figure 2d shows the evolution of this metric extracted from the timeline view, with a vertical plotting range clipped to $[0, 1.1]$. For CPU 0, the first memory controller, the value is close to 1 indicating a high fraction of local accesses. For the other CPUs the value is close to 0. Thus, most of the memory accesses are remote, targeting node 0, which finally explains the abnormal distribution of task durations.

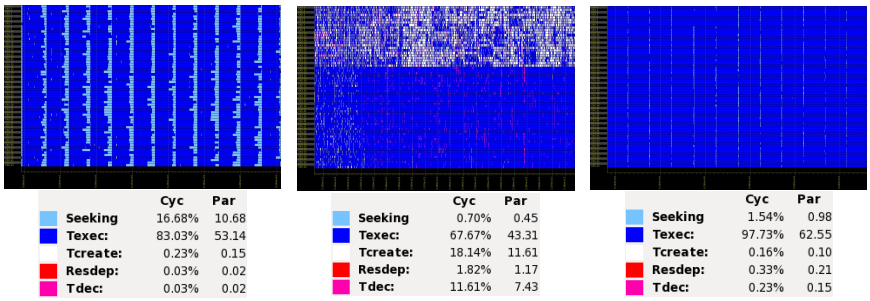
Interleaved allocation of the matrix data is one possible solution to this problem: distributing memory accesses across NUMA nodes reduces the contention on a specific memory controller. For this example, it speeds up the execution more than fourfold.

4.2 K-means clustering: block size

The *kmeans* application is a data-mining benchmark that partitions a set of n multidimensional points into k clusters using the K-means clustering algorithm. The block size passed as a parameter to the application must be chosen carefully as it determines the number of tasks and the amount of work per task. For huge block sizes, the available parallelism is low; tiny block sizes generate significant task management overhead. Choosing the block size which yields minimal execution time is not enough as the interaction between the application and the run-time might not be optimal. We need to analyze the execution trace to better understand how the application and run-time interact.

Figure 3 shows traces for three different values for the block size. With blocks of 160,000 points, as shown in Figure 3a, iterations of the algorithm become clearly distinguishable as alternating vertical "stripes" of tasks in either *execution* states (dark blue) or *seeking* states (light blue), where workers seek for new tasks to execute. At the beginning of an iteration, all worker threads execute tasks, but towards the end, most of the processors run out of work. At the origin of this problem are small differences in timing, which let some faster workers steal tasks from slower ones, hence creating an imbalance. On average, workers are in task execution state for only 83% of the time and seeking in the remainder.

For a block size of 625 elements, shown in Figure 3b, more parallelism is available as shown by the low amount of time spent seeking. However, the overhead of managing a large number of short-lived tasks reduces the overall proportion of time spent executing and, therefore, reduces execution performance. This overhead is apparent in the amount of time spent in states *tcreate* (task creation),



(a) $S_b = 160$k (b) $S_b = 625$ (c) $S_b = 10$k

Fig. 3: Worker thread activity in *kmeans* for different values of the block size S_B

tdec (synchronization counter management) and *resdep* (dependence resolving). In addition, processors 0 to 16 create significantly more tasks than the others, as can be seen at the top of the figure.

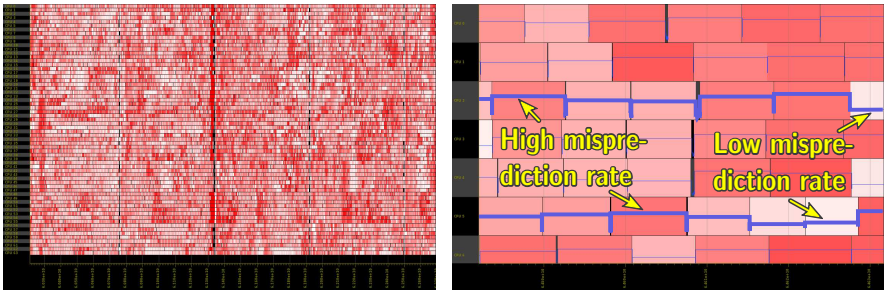
Figure 3c shows the timeline for a block size of 10,000 elements: a low relative task management overhead and sufficient parallelism yield the best performance.

4.3 K-means clustering: branch mispredictions

After determining the appropriate granularity, Aftermath can be used for further performance analysis. Figure 4a shows Aftermath's timeline in heatmap mode indicating the average task duration for each worker over time. To focus the analysis on the computation, all auxiliary tasks have been filtered out. Although all K-means computational tasks have similar workloads, their execution time is not uniform as shown by the shades of red on the heatmap. Contrary to the *seidel* example in Section 4.1, there is no clear and simple relationship between task duration and topology: each processor executes slow and fast tasks and no clear patterns are distinguishable.

Selecting a slow task from the heatmap and clicking on the task name in the lower part of the main window, opens an editor with the task's source code. The innermost loop of the task contains a conditional update of the cluster associated to a point. This results in frequently changing execution paths, which could significantly impact performance if the branch predictor is unable to track the pattern.

Aftermath can display hardware performance counters from a trace file, either directly or after having calculated the (discrete) derivative for each sampling interval. The latter option has been used to generate the branch misprediction count of Figure 4b. As the hardware counters for each processor are monitored right before and right after task execution, the graph interpolates with a constant value corresponding to the average misprediction rate for each task. The combination of the graph and the task duration heatmap immediately reveals a



(a) Global view covering several iterations (b) Zoom with branch misprediction rate

Fig. 4: Heatmap view showing the task duration of *kmeans*

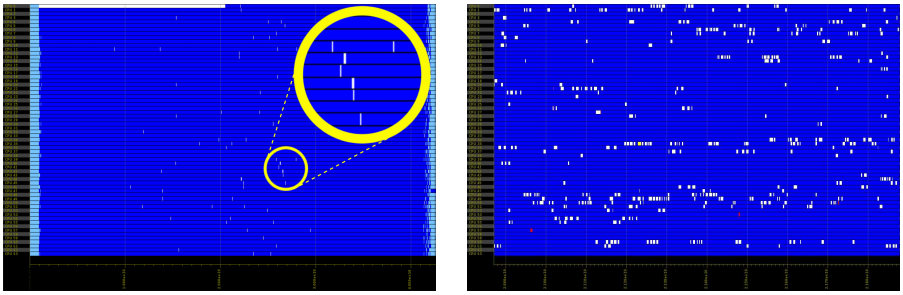
correlation: slow tasks (darker shade of red) have a higher branch misprediction rate than faster tasks (lighter shade).

It is possible to transform the condition, making the cluster update unconditional, and hoisting the check outside of the time-critical loop. The task duration distribution becomes more uniform, which solves the performance anomaly.

5 Debugging run-time performance

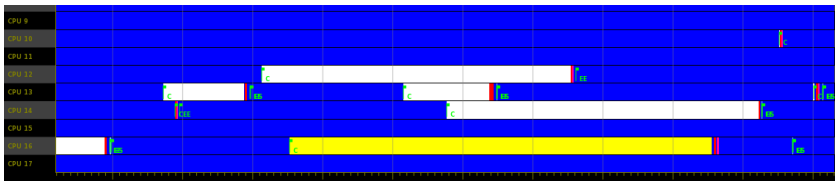
OpenStream relies on a slab allocator, enabling the run-time to locate free data-flow frames rapidly and to avoid expensive operating system calls at each allocation. Evaluation of run-time performance requires joint execution with an application. Hence, run-time debugging often adds up to joint run-time and application debugging. In this section, we show how Aftermath allowed to identify inefficient allocations in the run-time due to a parametrization error that becomes visible during interaction with the run-time of the *seidel* benchmark presented in Section 4.1. In this benchmark, sequential execution is usually optimized through loop tiling, which greatly improves cache behavior. The optimal size of a tile is highly machine dependent. For our test platform a size of 2^{16} elements, also used as the granularity of our parallel implementation, performed best.

We discovered a performance anomaly in the slab allocator when experimenting with an alternative, data-flow implementation of the benchmark: instead of using the original matrix in shared memory, tasks communicate single-assignment data stored in frames of the slab allocator. Figure 5a shows the timeline for this implementation. At a first glance, no performance problem is apparent, only processor 0 creates more tasks than the other processors during the initialization phase. However, a few task creation states rendered as small white bars are visible. Compared to the average task execution time, task creation overhead in this benchmark should be negligible and no such state should be visible in a global view.



(a) Global view on the trace

(b) Zooming reveals slow task creation



(c) Long task creation state (yellow) with short task execution state (blue)

Fig. 5: Long task creation periods in *seidel* due to inadequate parametrization of the slab allocator

It shows that it is essential to zoom into the trace, to reveal the presence at a finer scale of several longer task creation periods, as shown in Figure 5b. Selecting a state causes Aftermath to update the view with detailed textual information about the state and the associated task execution, which lets the user quantify the information of a specific event. For the selected task creation state in Figure 5c, Aftermath reports a duration of about 6 million cycles while the whole task execution takes approximately 9.72 million cycles. A task creation with a hit in the slab allocator usually finishes within a few thousand cycles. Long task creations indicate that the allocation missed the slab cache and that a system call was performed. For a benchmark with only a few different task types like *seidel*, this should only happen at the beginning of the execution when the slab cache needs to be filled. As task buffers are reused, subsequent allocations should be fast. However, scrolling through the trace shows that expensive allocations occur during almost the whole execution time.

Increasing the default maximum buffer size handled by the slab allocator of 512 KiB, which is slightly exceeded by the task buffers in this example, eliminates the largest fraction of slab misses. Most of the expensive system calls and page allocations can thereby be avoided.

6 Related Work

Visualization and analysis of trace files are common techniques, critical for performance analysis and debugging in high performance computing, for which

many tools have been developed. Without going into a complete survey of this field, we have selected four representative tools, *Paraprof*, *Paraver*, *Vampir*, and *ViTE* and we briefly explain why they do not fully meet our requirements for performance analysis in the OpenStream project.

Paraver [10] is a tool for interactive trace analysis, providing powerful filtering mechanisms for different graph types and independent views on trace data. Earlier versions of OpenStream included support for trace files in Paraver's native format. However, the tool's resource model focuses on computation and does not model memory-related resources and task communication patterns, which are essential to the characterization of performance anomalies on many-core architectures.

Paraprof [11], a profile visualization tool of TAU [12], is a retargetable framework for writing trace analysis applications rather than a single tool for a specific type of trace files or performance analysis. It provides a set of extensible components for data sources, data management, analysis and visualization that can be used as a basis for new tools. The overlap between functionality of existing components of Paraprof and those required for data-flow analysis in the OpenStream project is small, such that the implementation cost for an OpenStream-specific tool using Paraprof would have been close to an implementation from scratch.

Vampir [13] is a well-known commercial tool that has been used in high performance computing for almost two decades. It provides a rich user interface for interactive exploration and analysis of huge traces and has a highly elaborated filter interface. Multiple connected views with different granularity from cluster level to function calls are supported. But unlike Aftermath, the tool is optimized for analysis of massively parallel applications based on message passing. Neither NUMA resources nor tasks are modeled explicitly, making fine grained task-based and memory-related analysis impossible.

ViTE [14] is a freely available tool for trace-based analysis of parallel programs focusing on fast rendering. However, the tool lacks support for NUMA topologies and analysis filters.

7 Conclusions

We presented Aftermath, a tool for trace visualization and interactive trace analysis. We illustrated Aftermath's strengths on several examples based on genuine situations encountered through the development of the OpenStream run-time system and benchmarks. From these situations, we derived a list of requirements for performance debugging of task-parallel applications and run-times on many-core systems. Aftermath fulfills all these requirements, and has proven invaluable when simultaneously tacking the sources of performance anomalies in a task-parallel application and its supporting run-time execution environment.

Only a small number of the graphs and metrics covered in this analysis are specific to OpenStream or to data-flow languages. Aftermath can thus already be used for performance debugging of other task-parallel languages and run-times.

The use of a copyleft license for Aftermath's entire source code ensures that it can be extended or modified for any free software project.

It would be interesting to insert Aftermath in a comprehensive optimization framework. For example, one could complement performance analytics and visualization with a predictive model of performance anomalies [15–17].

References

1. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming. PPOPP '95, New York, NY, USA, ACM (1995) 207–216
2. Pop, A., Cohen, A.: Openstream: Expressiveness and data-flow compilation of openmp streaming programs. ACM Trans. Archit. Code Optim. **9**(4) (January 2013) 53:1–53:25
3. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical Task-Based Programming With StarSs. Intl. J. on High Perf. Comp. Arch. **23**(3) (2009) 284–299
4. Intel Corp.: Threading Building Blocks. http://threadingbuildingblocks.org (2009)
5. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 3.1. (2011) http://www.openmp.org/mp-documents/OpenMP3.1.pdf.
6. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with PAPI-C. In Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E., eds.: Tools for High Performance Computing 2009. Springer Berlin Heidelberg (2010) 157–173
7. The Cairo Graphics Team: Cairo graphics. http://www.cairographics.org/
8. The GTK+ Team: The GTK+ project. http://www.gtk.org/
9. AMD: BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors. (2013)
10. Pillet, V., Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. Technical report, In WoTUG-18 (1995)
11. Bell, R., Malony, A.D., Shende, S.: Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In: Euro-Par 2003 Par. Processing. Springer (2003) 17–26
12. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2) (May 2006) 287–311
13. Müller, M.S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E.: Developing scalable applications with vampir, vampirserver and vampirtrace. In: Proc. of ParCo '07. Volume 15 of Advances in Par. Comp., IOS Press (2008) 637–644
14. http://vite.gforge.inria.fr/
15. Parello, D., Temam, O., Cohen, A., Verdun, J.M.: Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In: ACM Supercomputing Conf. (SC), Pittsburgh, Pennsylvania (November 2004)
16. http://www.vectorfabrics.com/products
17. http://software.intel.com/en-us/intel-vtune-amplifier-xe