Tools for Exascale Computing: Challenges and Strategies

Report of the 2011 ASCR Exascale Tools Workshop held October 13-14, Annapolis, Md.

U.S. DEPARTMENT OF ENERGY
Office of Science

Sponsored by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research

ASCR Tools Challenges for Exascale Computing

October 13 – 14, 2011
Annapolis, MD

Workshop Report Committee
John Daly (Department of Defense)
Jeff Hollingsworth (University of Maryland)
Paul Hovland (Argonne National Laboratory)
Curt Janssen (Sandia National Laboratory)
Osni Marques (Lawrence Berkeley National Laboratory)
John Mellor-Crummey (Rice University)
Barton Miller (University of Wisconsin)
Dan Quinlan (Lawrence Livermore National Laboratory)
Philip Roth (Oak Ridge National Laboratory)
Martin Schulz (Lawrence Livermore National Laboratory)
David Skinner (Lawrence Berkeley National Laboratory)
Jeffrey Vetter (Oak Ridge National Laboratory and Georgia Institute of Technology)

Report Editor
Sonia R. Sachs

This report summarizes the results of a workshop on software tools for exascale machines. The goal of the report is to highlight the challenges in providing scalable tool support on exascale class machines and to identify key research and development needs as well as opportunities to meet these challenges. In this context we define tool support very broadly as software that helps programmers to understand, optimize and fix their codes as well as software that facilitates interaction between application, run-time, and hardware. This includes tools for performance analysis, static and run-time optimization, debugging, correctness verification and program transformation.

The workshop participants considered a range of approaches, from evolutionary ones based on existing tools to revolutionary approaches needed to cope with new challenges that will arise with the emergence or exascale systems.

Tools challenges for exascale systems include coping with the extreme scale of concurrency; gaining insight into the behavior of dynamic adaptive parallel applications and runtime systems; constructing tools that are resilient to failure; understanding data movement and power consumption; understanding of utilization of shared resources, including deep memory hierarchies, network, memory bandwidth, and I/O; and coping with multi-level parallelism and heterogeneous processor cores. Tools must support measurement, attribution, analysis, and presentation of performance and correctness properties. Each of these issues is challenging in the face of the complexity of exascale systems.

Contents

Executive Summary

This report documents the 2011 Exascale Tools Workshop held October 13-14 in Annapolis, MD. It is authored by the workshop organizing committee with input from the many lab, academic, and private sector attendees listed on the workshop website:

http://science.energy.gov/ascr/research/computer-science/exascale-tools-workshop/

It is the goal of this report to provide clear articulation and prioritization of challenges (both known and uncertain), prioritized list of responses, road-map with time-lines for implementing those responses, and a rough idea about the order of magnitude of costs involved. This report details the presentations at the workshop, the breakout groups work tasks, as well as homework tasks that were identified and completed after the workshop.

The workshop identified general challenges and solutions for the different categories of tools, as well as the specific challenges and solutions for each category. The workshop brought in cross-cutting topics, such as power, resilience, and criteria for measuring tools impact on applications. This is not the first nor the last forward-looking HPC tools workshop. Exascale is new to the discussion and brings with it more than just changes in *scale* for tools, but also significant changes in *scope* as well. The tool spaces discussed were performance as well as correctness and/or debugging. In both areas there are new topics which exascale brings with it. The targets for performance at exascale are more broad than simple time to solution and include power and thermal concerns. The nature of error detection and the context in which debugging is done are likewise expanded from something done during a side session with a tool, to the broader and more persistent context of production execution. In general we expect a need for more watchdogs for both performance and correctness at exascale that are "built-in" the exascale runtime compared to petascale. This is not to say that a one-size-fits-all approach is merited. A spectrum of tools are needed and both the tool in a session as well as tools built into the runtime approaches will complement one another.

In order to describe a path forward for exascale tools that serves both contexts, discussion was afforded to the topic of APIs for information sharing between tools and also with the application itself for auto-tuning. In keeping with the theme of broadened scope for tools, componentization of APIs to modularly organize the topics that a tool addresses are discussed. These APIs must also keep pace with the explosion of concurrency and scale expected at exascale. This last concern makes attention to performance overhead, asynchronous analysis capabilities, and fault tolerance key concerns.

Lastly, the path forward also includes a discussion of roles for HPC researchers that benefits from what the vendor ecosystem can provide in the tools space.

1 Introduction

Exascale class machines will exhibit a new level of complexity: they will feature an unprecedented number of cores and threads, exhibit highly dynamic behavior, will most likely be heterogeneous and deeply hierarchical, and offer a range of new hardware techniques (such as speculative threading, transactional memory, programmable prefetching, and programmable accelerators) that all have to be utilized for an application to realize the full potential of the machine. Additionally, users will be faced with less memory per core, fixed total power budgets, and sharply reduced MTBFs. At the same time, it is expected that the complexity of applications will rise sharply for exascale systems, both to implement new science possible at exascale and to exploit the new hardware features necessary to achieve exascale performance.

Looking back four years, to set the stage for the deployment and productive use of current computer platforms, the Workshop on Software Development Tools for Petascale Computing, held in Washington DC on August 1-2, 2007, identified and prioritized research opportunities in the area of software development tools for high performance computing. These opportunities have been documented in the ensuing report [51], which describes the basic requirements for tools that can help application developers deal with hardware complexity and scalability challenges at petascale. For performance tools, the report discusses needs for online measurement and adaptivity to better address heterogeneous and hierarchical architectures, and stresses the need for hardware and system software to make the necessary performance information available to tools to improve the accuracy of the performance analyses. For correctness tools, the report identifies scalability as essential, and pointed out that that application developers request lightweight, easy-to-use tools for diagnosing errors. Concurrently, it likens performance and correctness tools to efficient and flexible scalable infrastructures for communication, data management, binary manipulation of executables, batch schedulers and operating systems etc.

One might realize that the needs previously associated to petascale tools are in fact persistent. However, the exascale landscape poses many more formidable challenges, and as it has been pointed out "exascale is hostile for tools". In essence, applications and tools will face similar issues in exascale (e.g., new programming models and growing complexity) and will need to evolve concomitantly.

While several tool sets have been successfully deployed on petascale machines, in most cases this support is rather limited. Scaling is often achieved by applying brute force and tools are restricted to single programming paradigms. Furthermore, current generation tools mostly focus on the data collection combined with post mortem analysis and visualization and have only limited support for online or in situ analysis and evocation of response.

To overcome these limitations and provide the users with the necessary tool support to reach exascale performance, we need a new generation of tools that help users address the bottlenecks of exascale machines, that work seamlessly with the (set of) programming models on the target machines, that scale with the machine, that provide the necessary automatic analysis capabilities, and that are flexible and modular enough to overcome the complexities and changing demands of the exascale architectures.

To address the challenges posed by exascale systems and to meet the high-level requirements outlined above, significant research and development in the area of tools and tool support is needed. These efforts will need to focus both (1) on developing new tools capabilities that users will need to scale their applications to exascale and (2) on novel infrastructure that make implementing such tools feasible. Further, (3) research and development of software tools has a significant overlap with all other areas of exascale software and hardware design; these must be addressed as part of Co-Design efforts. Finally, (4) after tools have been developed we need concrete support models

that ensure the availability of the tools in the long run and across multiple hardware generations. In the following sections we will highlight issues and research opportunities necessary to cover these efforts.

User Expectations on the Usability of Exascale Tools Integrated simulation codes are large, often complex, and sometimes pushing limits of everything including language features. Tools deployed must be robust enough to handle these codes. At the same time these tools must be highly usable if they are to see adoption by user communities and deliver impact in increasing the performance of a large share of the exascale application space.

Usability is not a single target as there are many stakeholders and scopes within which tools are used. Tools used by end users must include options for high-level lightweight diagnostics that give an indication as to whether a performance loss or incorrect result has occurred as well as deep-dive tools that enable the power user or performance engineer to fully dissect the problem to be addressed. This provides a challenge to exascale tool design, development, and deployment since one or two full-featured, highly responsive tools will not exist. Instead there will be a suite of more special purpose tools. An important part of tool deployment will include establishing of a usage model for tools at exascale to document expectations, limitations, and a guide to their applicability for problem solving.

The goal of usability is to enhance the impact that tools have on exascale workloads by catalyzing their widespread use in broader HPC communities. It's not expected that all tools should serve all exascale stakeholders, but it is expected that all stakeholders have some tool that is accessible and usable for them to examine performance, debugging, and correctness questions.

User Requirements for Tools A key and new aspect of exascale is the scope of what users will require from tools. Instead of one or two simple measures of performance, exascale computing will see a broadening of scope as to what tools report and what they address. Measures such as floating point rate and communication time will be augmented with power consumption, data movement, network contention, and reliability concerns. In increasingly complex exascale architectures users will require expanded scope and increased attribution of events and metrics back to their causes. Exascale machines are expected to look significantly different from previous machine generations. They will feature significantly larger core counts, less memory per core, new hardware features like programmable prefetching or speculative threading, software controlled accelerators, etc. Users will expect tools to help them cope with these new features and the challenges they introduce. Additionally, while HPC tools have traditionally focused on debugging and computational speed, exascale tools must expand to support the measurement and analysis of other metrics of interest such as memory utilization, temperature, reliability, and power consumption. Moreover, the depth of the analysis provided must deepen if users are to be able to address and correct the issues identified by exascale tools.

Exascale Tool Development Strategy To meet the usability expectations and requirements stated above in the face of increasing machine and application complexity, the community developing tools may rely on a strategy of layered interoperability. Since different tools often share needs (e.g. code browsing, or binary analysis), support for tool components and sharing of tool infrastructure will be critical to develop the tools required for exascale in a timely fashion. Tool components providing these functions that are shared between different types of tools will both lower the development costs and deliver improved usability to users who will see common landmarks as they move between tools. One such example of hierarchical interoperability in the existing tools space is the

Performance API (PAPI) which is both a shared resource to tool developers as well as a common reference frame for users as to the names and meaning of performance events and measures. Given then expanding scope of what the goals of performance optimization are, minimizing not just wall time but factors such as power or data movement, we see a need for similar expansion in scope for such shared APIs.

Summary of Activity at the Workshop This section summarizes discussions conducted at the DOE Workshop on Exascale Tools held during October 13-14, 2011 in Annapolis, MD. The goals for the workshop were as follows:

1. Define objective criteria for assessing tools for exascale application development (debugging, correctness, and performance) along a wide spectrum of challenges, namely power, locality, resilience, etc.

2. Identify tools requirements and interfaces for hardware and system stack, particularly compilers and run-time systems and metrics for success. Identify integrated systems solutions rather than ad-hoc tools solutions.

3. Prioritize challenges for exascale tools, both generally and specifically to each tool category.

4. Prioritize options addressing the identified challenges.

5. Lay out a roadmap, with options, timeline, and rough cost estimates for needed exascale research in the different tools category.

To that end, the workshop agenda included 8 presentations by leading experts in HPC tools from DoE laboratories and academia, and three working sessions on (i) Performance Analysis Tools, (ii) Debugging and Correctness Tools, and (iii) Cross-cutting Issues. A brief summary of the workshop presentations and working sessions is included below, with a focus on past failures and successes as well as recommendations for future directions. The remainder of the report dives deeper into the technical details of these topics and also discusses their strategic implications in a broader context.

1.1 Background and User's Perspective

Some background to this workshop comes from NNSA and Office of Science working groups and related previous proposals. Additionally the landscape of what will be needed for exascale tools is considerably informed by extrapolation of current petascale user perspectives onto likely exascale target architectures. These discussions set the stage for the topical working sessions on performance analysis, debugging and correctness, and finally a working session on cross-cutting issues.

The ASC working group on tools formed in June 2010 and has met three times since to define and explore the needs across laboratories for the ASC community. The ASC perspective focuses on a full spectrum of tools including performance, correctness, and debugging. Runtime and compilers are not in scope for the ASC working group and left to vendors. The overall theme of the ASC working group at the application level echoes a need for a broadening context of monitoring (chip events, memory, power, locality, etc.) as well as a static analysis tools for evaluating code. At the system and file-system level a move toward holistic system-wide monitoring as opposed to ad-hoc specific tests is recommended. Bridging these two topics is a proposed strategy for hardware and software information exchange and leveraging through shared APIs. Those APIs expose information to the application for introspection and to the stakeholders in other ASC working group topics such

as scheduling and resource monitoring. The need for testing, integrating, and maintaining tools is also highlighted.

Background for the workshop was also provided from the defined but unfunded Exascale Software Center (ESC) activities, which had previously proposed a plan for exascale software including tools. Looking at the tools sections of ESC one finds a common call for increased breadth of what tools examine, which is in line with the ASC recommendations. Additionally, there are three unique areas of emphasis. First is a focus on scalable sampling which is detailed as a tractable means to deal with exascale concurrencies. Second is an explanation of the importance of attribution of both performance and correctness measurements back to root causes in the code or hardware. Third is the possibility of co-design as a means to improve hardware support for the needs of exascale tools. There are additional common threads between ASC and ESC such as managing exascale data volumes, analytics to provide insight, and software interoperability.

As exascale plans unfold, much remains to be determined about the precise nature of exascale architectures and programming models. A great deal, however, can be learned from the perspectives of the scientists who currently use the petascale computing resources at NERSC, OLCF, and ALCF as tools to drive scientific discovery. To embrace and benefit from those perspectives, the workshop used information from these facilities about current usage and needs in HPC tools as background material. Many of the science teams using these facilities at large scale now are likely to be to early adopters of exascale computing.

NERSC has a 4000+ user base, which spans a wide range of science applications and requirements. Through it's annual user survey, trouble ticket system, and user software environment, NERSC tracks when, how, and to a lesser degree why users reach for tools. The overall synopsis is unsurprising in that users lack inherent interest in tools but do reach for them when there is a performance or correctness issue which impedes their research. NERSC records invocations of performance tools and debuggers and finds the usage of performance tools to be roughly twice the amount of debuggers, a rising interest in filesystem and I/O tools for performance measurement, and significant usage of lightweight monitoring tools that provide ongoing regular assessments of performance from production workflows. Conversely, users express dissatisfaction with tools which are cumbersome, have a steep learning curve, or impede their workflow from progressing. On the exascale horizon, NERSC users will be well served by always-on lightweight tool layers that provide dashboard level diagnostics that pro-actively indicate where application performance or correctness issues may exist. From that point a variety of deeper-dive tools should be available electively as needed.

OLCF reports a small but increasing fraction of users embracing performance and debugging tools with an indication of interest in portable tools across multiple platforms and compilers. In some sense this tracks not the overall measure of demand, but the fraction of users for which tools are required to make forward progress as opposed to ongoing regular use of tools. A well defined support model for both fixing tool defects and training in tool usage is seen to be important to the future strategy for tools. As OLCF is heavily invested in GPU architectures and a significant effort ongoing is to bridge the gap between existing tools and heterogeneous architectures for both performance analysis and debugging. Those efforts combined DOE, TU Dresden, and vendor for R&D efforts to deliver an interoperable tool chain that can operate at full system scales.

ALCF reported on both an inventory of tools that are either currently deployed or in process of being deployed on IBM BlueGene architectures as well as forecasts for exascale tool use. A wide spectrum of tools from the standard unix tools to BlueGene specific vendor tools are in place on the BlueGene/P system and in planning or development for BlueGene/Q. These tools come from vendors, academia, and including tools developed in-house, but the focus is on open source tools. The ALCF BlueGene experience points towards the need for a robust DOE tools program going

forward to exascale as even at petascale vendor-only strategies have not succeeded. In particular the vendor's ability to sustain tool development, support, and maintenance are difficult to maintain after the initial R&D for the machine has been completed. Looking toward exascale, ALCF users put emphasis on better hardware performance counters, tool support for architectural features such as SMT and transactional memory, as well as thread profiling and correctness checking. Issues of new or renewed emphasis for users also include memory usage and layout, network congestion, OS level information on scheduling and memory usage, and finally the recurrent topic of bringing power usage into the tools space.

The last presentation from the background and introduction to the workshop presented the NNSA user perspective, which reinforced that many petascale issues remain to be solved as we begin to examine exascale issues. This perspective emphasized the role of tools in production computing and the need for tools to add to rather than detract from the science productivity of users. In petascale and looking toward exascale, disruptive changes in HPC environments impede the utility of computing resources as scientists must focus on porting and learning new tools. Thus, there is a balance between what new programming models and tools offer and the new science that could be explored if continuity in the HPC space can be provided. NNSA's large integrated codes are complex and hard to change. Any exascale directions that embrace radical changes in tools or environments should recognize the negative impact on science output that may be incurred. Looking at the exascale horizon NNSA users and staff see a need to provide a shared infrastructure for tools, improved hardware and software APIs to wrap, monitor and introspect applications, and a growing need to examine application issues both post-mortem and in-situ.

1.2 Characteristics of Exascale Systems Affecting Tools

Power Power consumption will be a significant constraint for exascale systems; thus, it will be important for exascale applications to make efficient use of power. To keep power consumption and chip temperatures within an allowable range, current microprocessors employ thermal throttling, which involves adjusting clock frequency to one or more of its cores. On exascale systems, it is expected that software will also be able to adjust voltages and frequencies as well to tailor power consumption for code regions.

Failure Due to the enormous number of electronic components, exascale systems will fail more frequently than systems today. This has two implications for tools. First, tools will need to provide applications with mechanisms for monitoring failure so it can be managed appropriately. Second, tools will need built-in fault tolerance to survive.

Extreme scale Exascale systems will contain millions of cores and billions of threads [42]. As a result, performance tools for exascale systems will need both mechanisms and policies for coping with scale during measurement, analysis, and presentation.

Multi-level parallelism Parallel applications will include multiple levels of parallelism such as task parallelism between parts of a coupled application, process parallelism (e.g., multiple MPI processes), thread-level parallelism within a process, as well as parallelism within a core, including hardware multithreading, instruction level parallelism, pipelining, and short vector parallelism.

Dynamically varying hardware performance Today's processors already employ frequency and voltage scaling of cores to control power consumption and keep processor chips within their

target thermal design envelope. When such scaling is applied, both the energy efficiency and computational power of cores changes. In exascale-class systems, other factors will cause non-uniformities, including process variations that may cause some cores or whole chips to perform differently, as well as hardware recovery of errors that may either be transient or persistent.

Heterogeneous cores Processors in exascale systems are expected to contain multiple heterogeneous cores. To keep power consumption low, most cores will be low-power lightweight cores optimized for throughput; a smaller number of latency optimized cores will be available to execute less parallelizable work. Performance tools will need to assess how well computations are utilizing each kind of cores as well as the overall utilization of the system.

Complex memory subsystems Data movement is much more costly than communication. Exascale systems are expected to make use of deep memory hierarchies to avoid the latency and power costs of off-chip accesses. Multi-socket nodes will contain NUMA memory domains.

Hardware support for resource management For efficiency and performance, hardware will play an increasingly central role for managing shared resources. Specifically, hardware will play an integral role in managing thread-level parallelism and communication flow control.

Asynchrony To tolerate latency and overlap I/O and communication with computation, exascale applications will need to make extensive use of asynchronous I/O and communication. Furthermore, computation on different kinds of cores may be decoupled as well. Today, for performance computations on GPU accelerators are most commonly launched using an asynchronous interface.

Adaptive software As described previously, hardware performance will have dynamic performance characteristics. To cope with such performance variability, hardware and software systems will need to adaptively schedule computation among processor cores.

Hybrid programming models Programs on exascale systems will likely include legacy components in addition to components based on emerging models. Measurement and analysis of hybrid programs will undoubtedly be harder if the programming models used for its components are dissimilar.

1.3 Performance Analysis Tools

The workshop discussion on performance analysis tools opened with a discussion about what the objectives and units of performance are. In some sense, science per unit time has always been the intended goal of HPC computing performance. Over many generations that has been reduced to the more measurable quantities of wall clock times, FLOP rates, computational intensity, etc. What is unique about exascale is how these simplistic measures of performance are being met with constraints on power consumption and data movement. Accordingly we must re-calibrate our intended goals in attaining performance to accommodate and inform this broader sense of what performance is all about. To adjust to these new constraints, the scope of what is monitored must broaden to include power and thermal data at chip, node, and possibly facilities levels.

There must also be a re-calibration of performance expectations through performance assertions that describe the fractional utilization of the rate limiting resource stands in the way of performance. It may be ok to run half as fast if one understands that a shared resource is being contended for.

Another balanced way to set performance expectations is by relative comparisons. Differential profiling, running the same application at differing scales or workload levels, can both serve to describe performance in ways that no single number can. At exascale it may be important to think about performance in terms of distributions and expectations rather than as simple static metrics.

Moving on from the topical discussion of the nature of performance, the group identified the increasing need to provide not just metrics but attribution and root causes indications about those metrics. A discussion on what the right semantics and context to provide such attribution pointed to the need to place performance metrics and analysis in a broader context of the system-wide data, allowing for inter-job effects, and to easier and more accurately annotate within a job the phases of computation and boundaries between coupled computations in, e.g., mutli-physics or MPMD codes. The effects of inter-job resource contention are already widely felt in subsystems like filesystems and interconnects. At exascale the need to take a facility level view of performance analysis will become vital. The semantics for root-cause attribution are also potentially impacted by the adoption of domain specific languages (DSL's), which both raise the level of abstraction in attribution, but also provide a smaller possibly more well known set of points to direct instrumentation for attribution.

Having determined from the previous discussions that the scope and nature of performance analysis at exascale will be markedly different, the group turned to the practical discussion of how and whether the needed monitoring can be achieved. The interfaces in hardware and software that will be required must meet certain design goals. First, both static and dynamic analysis will be required. Second, the interfaces must recognize that there will potentially be multiple consumers of measured events. To the latter point it becomes important that the APIs by which tools gather their data, and by which applications potentially introspect on their performance, be systematically defined and allow multiple contexts for monitoring. An API that spans multiple performance contexts (chip counters, thermal, power, interconnect, etc.) may be provided using uniform semantics and a components approach to define the contexts. One significant challenge may present itself in events which essentially can not be observed. For instance, if the clock rate is throttled in hardware based on thermal conditions, tools may have, at best, secondary means to gather what has happened and why.

Exascale tools that monitor and report on data from these APIs will need to span multiple use cases. The spectrum of shrink-wrapped easy to use tools to experts-only tools will be needed to maintain performance at exascale. Making performance tools a first-class element in exascale software was discussed. The need to ask performance questions after a run has started, to auto-tune an application, or to dump a detailed snapshot of an application's performance state and allow if to continue running normally are all modes that should be accommodated. Allowing users to insert performance assertions into their code that, if not met, take action on their behalf will be best accomplished with some inter-operation with compiler and runtime engineering.

Techniques described later in this report, such as sampling and profiling, are a crucial means to achieving the scalability required for exascale. Data volumes which scale with concurrency must be mitigated if performance tools are to find adoption outside the most heroic cases where large quotas and extreme slowdowns are tolerated. A hierarchical approach for exascale seems fitting whereby easy performance analysis is made easy and in-depth tools are available to dissect performance challenges in detail. Likewise a combination of in-situ and off-line analysis of performance data will be needed. Where possible, finding categories or clusters of performance states may assist in the data challenge. Likewise the performance attribution discussed above can aid in this area by informing which pieces of performance data are actionable and worth saving versus the ones which may be ignored.

Lastly, we discussed the topic of machine learning research and its potentially untapped utility in performance analysis tools. Exascale will broaden the number of performance data gathered

both as concurrency goes up and as the jobs running in an exascale machine are imagined to be large in number and interacting. Additionally the potential for auto-tuning at exascale will be greater than ever. These two avenues open the door for a variety of interesting exascale research topics. Broadly, we may classify them as machine learning between jobs, i.e., understanding at the workload and inter-job contention level the dynamics and expectations of performance at a system level, and machine learning within jobs or applications where the application seeks out patterns or structure in how it can best adapt to meet performance goals. Both directions are simultaneously achievable and indeed synergistic. An operational exascale machine will itself be a complex system worthy of study and research. Machine learning likely has important roles in that regard. Finally, it was noted that the first topic on the expanding nature of performance measurement must be noted when the objectives of machine learning are put in place.

Recurrent exascale themes in this discussion were the broadening of scope of performance measurement topics, the broadening of scope of analysis from the application to the workload, challenges due to concurrency and data issues, the need for attribution of root causes, as well as an acknowledgment of the need for a spectrum of tools for a wide range of use cases.

1.4 Debugging and Correctness Tools

The Debugging and Correctness Tools session was organized around a series of questions posed initially to the panelists and then opened to all attendees for comment.

The first topic considered was the impact of streaming processors (especially GPUs) on debugging. Part of the tools challenge for streaming processors is that the programming model is still evolving rapidly. Most attendees felt that the programming model would eventually converge into something higher level that would permit programmers to abstract away many of the details of streaming processors (and certainly the details of specific vendors processors). From a standpoint of debugging, additional complexity due to streaming processors arises from the larger number of threads, the separated addresses spaces, and the difficulty in fault and breakpoint containment. Memory debugging of unique hardware features such as scratchpad memory is critical. It is important that, when developing new programming models such as domain specific languages (DSL), debugging is considered from the start. Likewise, additional extensions to binary file formats such as DWARF might be needed to handle debugging DSLs.

One potential new challenge for exascale debugging will be the power limits. Since debuggers generally require additional threads and code to run to support them, it is possible that the debugger could put the entire application over its power budget. Ways to deal with this could include dedicating resources for debugging (i.e., allocating extra nodes if the debugger is used). It was also felt that for debugging, slowing down nodes (to bring the application back within power limits) would be acceptable.

Fault tolerance, especially when the user is expected to supply some of the detection or recovery code, will further complicate debugging for exascale. Debugging environments will need to support injecting faults (and multi-faults too) into an application to allow testing of these mechanisms.

Debugging at full scale will still be required to find bugs. Thus, online tools that are able to find equivalence classes or groups will be critical. Sampling will play an increasing role, but effectively using sampling for anomaly detection is hard.

Some additional hardware support may be required for debugging. Examples might include global timestamps or improved watchpoint registers.

Program analysis techniques (static analysis) could help to identify problems at smaller scale. For example, using symbolic techniques to reduce problems to examples that require fewer threads. However, it will be important for tools to require constant or logarithmic time proportional to

the number of threads (given the large number of threads expected). In addition, improved type systems may improve both the correctness of programs and the quality of information from tools. The growth in popularity of scripting languages and some styles of domain specific languages has made the available type information worse, not better, recently. There is a need to either enhance programming models to support richer typing, or develop new type inferencing systems to discover type information that is not explicitly defined.

1.5 Cross-cutting and Tool Infrastructure Issues

To join the distinct tools topics into a coherent program, the workshop discussed the cross-cutting issues that relate to both performance, correctness, and debugging tools. This also included an extensive coverage of tool infrastructure requirements and challenges. All or most of the tool areas discussed will need to interface with both applications, runtimes, and the underlying architecture. Likewise exascale tools, and exascale software more generally, will need to be engineered and tested in ways that make them reliable resources for the computing community.

The cross-cutting topic of interoperable APIs for tools is wide ranging but tractable. Most tools will need some functionality to connect and detach from applications, to scalably transfer and aggregate data from many threads/tasks, and provide analytics and visualization. Many such APIs exist but they are not integrated or streamlined for use at exascale. We need to redesign them to create the basis for a common shared tool component infrastructure that maintains a set of independent and scalable components and uses them to enable quick assembly of components into new and scenario specific tools.

Software engineering topics in this space are numerous and already under-served at petascale. There is a need in the HPC software community to raise the bar when it comes to design, build/test, packaging, and release of software tools. Models where tool developers do custom installations on various HPC resources are common and undercut the notion of HPC tools as robust and reliable software products. Exposing build/test data to the broader community, providing attention to making software packages such as RPMs that are easily deployable, and improved documentation will all increase the positive impacts of exascale tools.

2 Performance Analysis Tools

Harnessing the potential of exascale platforms will be a daunting task because of their unprecedented complexity and scale. Hardware and software support that helps users identify performance bottlenecks, understand their causes, and identify how they might be ameliorated will be crucial for using these machines effectively.

It is critical to note that HPC performance tools do not stand alone. Rather, they exist as part of a HPC ecosystem that includes both hardware and software; these tools are critically dependent on this ecosystem for functionality and features. While many other parts of the software stack leverage abstraction to reduce complexity, performance tools don't have this luxury; they must interact with all system components to gather information about resource consumption, inefficiencies, and opportunities for improvement. Specifically, performance tools require access to state maintained by the hardware, operating system, programming model runtime systems, compilers, and applications. For each significant hardware and software component they must have interfaces to gather and analyze information, often at an extremely detailed level. All of the hardware and software components must be designed with appropriate interfaces for tools. Lack of support from any hardware or software component in the HPC ecosystem can prevent effective tools from being developed and deployed.

Traditionally, the role of performance tools on parallel systems has been to support measurement, analysis, attribution, and presentation of application performance measurements for post-mortem inspection. However, to get the most out of exascale hardware platforms, applications will also need to monitor, analyze, and respond to emerging performance and reliability issues. Specifically, applications will need to be aware of their own performance characteristics, such as declining spatial and temporal locality as data objects and their relationships evolve during execution, as well as the mapping of computation, communication and I/O onto physical resources. In addition, applications will need to monitor and react to dynamic performance, adaptive decisions by the runtime, and failure characteristics of the hardware on which they execute.

Section 2.1 outlines some key research challenges for performance tools for exascale systems. Section 2.2 describes support that tools need from other parts of the ecosystem. Finally, Section 2.3 briefy summarizes the state of the art in performance tools.

2.1 Research Challenges for Performance Tools on Exascale Platforms

The need for performance analysis and tuning of software on exascale systems will span the entire software stack. Traditionally, tools have focused on applications alone. In addition to traditional tools for post-mortem analysis of application performance, performance tools for exascale software must also provide interfaces for on-line performance measurement and analysis, which will be needed to guide adaptive applications. The requirement for on-line analysis applies to many of the problems that performance tools must address.

2.1.1 Minimizing Power Consumption

To date, power consumption has been managed by hardware alone. To tackle the problem of power consumption more effectively, it will be important for exascale systems to make observation of power consumption accessible to software as well.

There are two ways that software can assist with reducing power consumption. First, performance tools will need to be able to measure and attribute power consumption to application and library source code. Such tool capabilities will enable users to identify power inefficiencies that can be improved by adjusting data structures and/or executable code associated with the inefficiency.

Second, as applications execute, they will need to be able to measure power consumed by particular code regions. For inefficient code regions that lack a way to improve efficiency by transforming code or data, it may be beneficial for an application to lower the clock rate for a code region so that processor cores aren't consuming as much power while waiting for long latency memory operations to complete. In other cases, a program may be able to improve its power efficiency at runtime by reorganizing the data it is manipulating to improve spatial and temporal locality of data accesses. For instances, reordering the nodes and edges in a graph can improve the efficiency of repeated traversals.

Constructing power-aware tools and applications will require hardware capabilities for measuring power consumption. Further study is necessary to understand what sort of power measurements will be most useful. For instance, one could measure power for processors, memories, and interconnects within and across nodes. Without a finer granularity of measurements, e.g., at the core level, it will be hard to make power feedback actionable during post-mortem analysis or online.

At the same time we must also consider the power consumption of the tools themselves. Their power budget will add to the overall power consumption and when applications operate close to the maximal available peak power, tools must be careful to not push the application beyond those limits or to force executions at smaller scale where the results may be not appropriate or helpful.

2.1.2 Coping with Failure

Performance tools for exascale systems will need to deal with failure in three distinctly different ways. First, measurement, analysis, and presentation mechanisms themselves must be resilient or tolerant of failures. While existing tools can tolerate some kinds failures, e.g. missing data for some threads or processors, more pervasive fault tolerance within tools will be needed. Second, performance tools will need to measure the frequency and cost of failures and recovery actions by applications. Third, tools must automatically adjust to changing system and application configurations caused by a recovery process and must be able to continue meaningful and consist measurements after an application restart. The latter requires a significantly deeper integration into the system stack and into fault tolerance techniques than currently possible.

2.1.3 Managing Performance Data at Scale

One of the core challenges for performance tools at exascale will be the scalable collection and analysis of performance data. With millions of cores and billions of threads, measuring the performance of exascale applications will involve collecting a flood of data to ensure that measurements capture the range of behaviors. The scale of executions will make capturing comprehensive traces for long-running executions impractical. Designing performance analysis measurement and analysis strategies that deliver insight without exhaustive data will be important.

Different data organizations are appropriate for measurement, analysis and presentation of performance data. For instance, thread-centric measurements are the easiest to record because each thread can record its own measurements independently. However, resource-centric, time-centric, code-centric, and data-centric perspectives are more appropriate for analyzing certain kinds of performance problems. Because of the large data volumes involved, recording, analyzing, and transforming measurement data must necessarily employ scalable parallelism. While it will often be useful to distill performance data into compact summaries, e.g., a profile, to obtain a high-level perspective of application performance, understanding performance problems that vary over space and time will necessarily require significant amounts of data. Storing large volumes of performance data in forms suited to different kinds of analyses and presentations will require careful design of

multiple persistent representations. While a dense representation may simplify the task of locating a particular piece of data, a sparse representation will often be dramatically more compact. The design of appropriate persistent representations for exascale performance data will need to consider space, access patterns, and the time complexity of reconstructing desired statistics or renderings.

2.1.4 Assessing Data Movement

On exascale systems, data movement and power consumption will be closely related. Current projections are that in exascale systems reading or writing a value from DRAM will require a factor of 800 more power than a double precision floating point operation [12]. Even just moving a double precision value across a chip will cost over a factor of ten more than computing with it [12]. As a result, keeping power consumption low will require minimizing data movement.

Exascale systems are expected to employ multi-level memory hierarchies to keep data on chip and close to the processor cores manipulating it. While one can monitor a thread's data access patterns with instrumentation, without using a simulator understanding the interplay between threads, data at various levels in the memory hierarchy, and policies for scheduling computation will require hardware support. Only with hardware monitoring will it be possible to identify the level of the memory hierarchy where the data is resident as well as the latency and power associated with moving it to the requesting thread. It will be important to measure data movement up and down through the memory hierarchy as well as across and between chips within a node.

Performance tools for exascale systems will need to leverage hardware monitoring capabilities to identify data objects that are the subjects of inefficient access patterns, identify code regions that access data objects inefficiently, quantify the costs of these inefficiencies, and provide guidance as to how the program might be improved.

2.1.5 Resolving Network Performance Bottlenecks

The bisection bandwidth of HPC systems is not growing linearly with their computational power. For that reason, we expect that tools that help identify, explain, and resolve performance problems due to network congestion will be important on exascale platforms. Recent experience has shown that dramatic improvements in application performance can result by adjusting how applications use communication networks in tightly-coupled HPC systems. Examples include using a better embedding of an application's logical communication topology into the physical topology of a machine [21] or by spreading communication over time [33]. As a result, performance tools for exascale systems will need to address measurement, analysis, attribution, diagnosis, and presentation of performance problems on communication networks. I/O networks will need similar support for assessing how and why performance problems arise.

2.1.6 Assessing the Impact of Asynchronous Operations

On today's systems, asynchronous (non-blocking) operations are used for interprocess communication, I/O, and scheduling work on GPGPUs. On exascale systems, asynchronous operations will be ubiquitous for efficiency. In the presence of asynchrony, the causes of performance problems may become far removed from the symptoms. Performance tools must measure asynchronous operations and assess their impact on system performance.

Some important questions performance tools will need to answer include assessing whether the presence of asynchronous operations causing a performance benefit rather than a loss. In today's communication networks, asynchronous operations are treated as "unexpected messages" and often produce more trouble than benefit. For exascale systems, it will be critical to adjust mechanisms

and tune policies so that applications can benefit from asynchrony. Tools will need to derive answers to some of the following questions:

- Are asynchronous operations being used effectively to overlap different kinds of activities?
- Is one kind of activity stalled waiting for another?
- Can we bound how much improvement might be achieved by better exploiting asynchrony?

2.1.7 Tuning Adaptive Resource Management Policies

For reasons described earlier, exascale platforms will require adaptive algorithms for mapping and scheduling communication, computation, and I/O. Tool support for gaining insight into the behavior resulting from such algorithms and identifying opportunities for improving their performance will be a significant new challenge for performance tools. Any significant intrusion of the tools into the normal operation of such software is likely to substantially alter the behavior of the system under measurement. Measurement, analysis, and presentation techniques that provide insight into the behavior of such algorithms and assess how well they are managing the resources under their control will be needed.

Monitoring the shared resources (shared cache, bandwidth and pipe line usage for SMT) Per-thread information is not enough to reflect the shared resouce contention. System-wide profiling is necessary. A naive analysis method to blame the costs to causes is as follows: (1) Use a separate thread called monitoring thread to sample (EBS or proxy based sampling) the shared events. (2) In the interrupt handler in monitoring thread, send signals to other threads which read the event consumption in per-thread mode. Attribute the system-wide consumption to the top two threads with the largest events in per-thread mode.

For data presentation, we can create a new view to show the shared event consumption (system wide) according to the time line. Together with the trace view, we can know the time interval with high shared event consumption is corresponding to the concurrent execution of functions in each thread. Note that the contention should be related to per-thread performance and time interval (actually the schedule of the threads).

2.1.8 Diagnosing Root Causes of Performance Problems

In many cases, when measuring the performance of parallel programs it is straightforward to pinpoint symptoms of inefficiency. For instance, idle worker threads in a multithreaded program indicate that there is insufficient parallelism to keep all workers busy. Similarly, a thread spin-waiting for a lock in a multithreaded program is a symptom of lock contention. However, identifying the root causes of such performance problems is a different matter. Recently, it has been shown that the root cause of each of these problems can be pinpointed and quantified with a *problem-focused* measurement strategy designed to transfer blame from threads suffering the symptoms of inefficiency to the threads causing the inefficiency [44, 47]. Being able to quantify losses associated with such abstract performance problems and attribute them to source code is important if one wants to improve the performance of complex parallel programs.

On exascale systems, the root causes of application performance problems will be difficult to identify using traditional measurement strategies because of system complexity. This will be especially true within nodes because of their massive threaded parallelism, heterogeneous cores, dynamically varying hardware performance, and adaptive software. Developing problem-focused

measurement strategies along with analysis capabilities tailored to interpret the resulting measurement data will be important for diagnosing root causes of performance problems in the face of this complexity.

2.1.9 Analyzing Contention for Shared Resources

A particularly difficult performance analysis challenge is understanding contention for shared resources. This problem arises with cores shared between multiple hardware threads, cache shared between multiple threads and/or cores, as well as with memory and communication bandwidth shared by all threads on a chip. In today's systems, it can be hard to understand the performance impact of having threads compete for shared resources of various kinds. With exascale systems, the enormous number of threads involved will make it critical to develop techniques for measuring and quantifying the performance impact of contention. Of course, for such measurements and analysis to be actionable, tools must attribute the costs of contention for shared resources to code.

To improve application performance on exascale systems, it will be important to determine whether or not an application generates significant contention for a particular shared resources such as the interconnect between nodes. Does such contention occur in bursts or is it continuous? Is the load placed on a shared resource by various threads on a node uniform or non-uniform? The answers to these question will frame the approaches that may be useful for ameliorating contention to improve performance.

2.1.10 Coping with Hybrid Architectures

Exascale architectures are likely to contain a heterogeneous mix of both throughput-optimized and latency-optimized cores. One must measure and assess performance carefully depending upon the application developer's objective. One objective might be to minimize power consumption, which would likely require using the most appropriate type of core for the workload at hand. Another might be to minimize execution time. A third might be to balance both power consumption and execution time.

Due to the fact that latency-optimized and throughput-optimized cores are rather different, the performance of each will need to be measured separately. A dependence on work running on one may require the other to stall. Thus, analyzing the performance of heterogeneous will consist of analyzing the performance of each kind of cores separately as well as analyzing the relationship between them. Since measurement support for latency-tolerant cores is in a fledgling state, the broader problem of assessing the performance of heterogeneous nodes has not received much attention to date.

2.1.11 Coping with Massive Threading

At exascale, the use of threading will no longer be optional for applications. Without threads it will not be possible to reach the concurrency levels needed for exascale while staying within the confines of the limited node memory. Further, new architecture paradigms, like GPGPUs, explicitly rely on threading. However, tool support for threading is currently still weak and new approaches will be needed to provide the user with performance analysis of threaded programming models. Analysis of threaded code will include understanding overheads associated with threading (e.g., start up costs for threaded regions) and assessing whether applications are exploiting threaded performance effectively.

2.1.12 Data Mining and Presentation

Rather than simply presenting performance measurement data to application developers for them to explore, the complexity and scale of exascale executions will require tools to do more to direct attention to problems and phenomena of interest. Presentation tools must be scalable to cope with the volume of measurement data from exascale platforms. Useful techniques for analyzing executions of enormous scale will include adaptively and selectively recording of performance data, *in situ* or online analysis as well as data compression.

Regardless of how data is collected and analyzed, it will be important to present performance analysis results in a scalable actionable form. Different views of performance data are useful for tackling different kinds of problems. Code-centric views enable one to pinpoint performance bottlenecks associated with program regions. Data-centric views enable one to better understand the costs associated with particular program data structures. Time-centric views help one better understand how system state, application data, and process activity evolves over time. In addition, it will be worthwhile to explore alternative approaches for presenting performance data, e.g., mapping it to the physical domain familiar to application developers or to the physical topology of the target platform.

2.2 Cross-cutting Issues

Since performance tools must interact with all of the hardware and software components in parallel systems, it is critical that each of the key components in ecosystem provide the necessary capabilities and interfaces to support tools. Here we outline requirements for tools starting from the hardware up.

2.2.1 Hardware Support for Performance Monitoring and Adaptation

Hardware support for performance monitoring of exascale systems is a topic that merits co-design with tools. The design of new hardware technologies must consider tool support required to understand correctness and efficiency. Below are a set of things that will require hardware performance monitoring for tools to have adequate insight into application performance and efficiency and to attribute any performance measurements back to the right locations and data structures within the application.

Power Power will be one of the defining problems on exascale systems. While the hardware will need to monitor power and apply thermal throttling when necessary to protect its integrity, various layers in the software stack will need to measure power consumption to (a) attribute power consumption to program code and data for post-mortem analysis, and (b) guide on-line adaptation where program actions (e.g., reordering data) can reduce power consumption. While power monitoring support at the board or rack level could be used to guide system-wide adaptation; finer grain monitoring of or within chips will be necessary to guide application-based adaptation.

Failure Hardware support for monitoring failures in any subsystem will be indispensable. Failure identification will be needed in memories (e.g., ECC), communication (e.g., checksums), and computation. Ideally, failures should be able to trigger interrupts so that they can be associated with code and data affected.

Processors Hardware performance monitoring is critical for providing insight into application behavior. Exascale systems will need hardware support for measuring resource consumption, inefficiency, delay, and power consumption. Two important capabilities for monitoring hardware are support for problem determination via cycle accounting and accurate attribution. For the former, performance monitoring hardware should support a set of events that enable one to hierarchically decompose execution cycles into various categories, with an emphasis on understanding underlying causes for stall cycles. For the latter, it must be possible to attribute performance information sufficiently accurately to program code and data to guide optimization. Most processors today fall short on these requirements.

Memory On exascale systems, data movement will cost more than computation. Hardware support for measuring data movement between levels of the memory hierarchy and between sockets will be critical to obtaining insight into an application's behavior to improve performance and reduce energy consumption. Important characteristics to measure will include latency, memory parallelism, bandwidth utilization, as well as measures that will provide insight into locality and contention.

Network Getting the most out of the various networks in exascale machines will require having hardware monitoring that will enable one to identify and distinguish various kinds of inefficiencies, including hot-spot contention, bursty utilization, under-utilization of available channels, and logical to physical mappings that cause congestion.

2.2.2 Operating System Interfaces

Full access to hardware performance monitoring capabilities. Current operating systems don't provide access to and control of all hardware monitoring hardware available in today's microprocessors and networks. In future HPC systems, all appropriate hardware monitoring performance monitoring capabilities must be exposed to software tools.

Interfaces for inquiry and control. Operating systems for exascale machines will need to provide interfaces that support inquiry and control related to logical to physical mappings of data, computation, communication and I/O.

Interval timer functionality. A common weakness of interval timer modes on most operating systems is that one can only perform real time profiling of single-threaded applications; thread-level profiling is only supported using the `ITIMER_PROF` mode, which measures only the time spent by a thread or spent by the kernel on the thread's behalf, *but not real time*. Only the Solaris operating system supports the `ITIMER_REALPROF` profiling mode, which enables one to perform real time profiling of multithreaded programs and provide a detailed accounting of how much time a thread spent in each micro state [30]. Another benefit of `ITIMER_REALPROF` is that it does not deliver interrupts to a thread while it is blocked at a system call. Without this profiling capability, itimer-based profiling of multithreaded applications can fail to properly account for time spent at some blocking system calls. Operating systems for exascale platforms will need the capabilities of `ITIMER_REALPROF` for efficient and accurate time-based profiling of massively multithreaded programs.

Support for *in situ* analysis Operating systems will need to provide sufficient control over process and thread thread placement to facilitate co-locating tool components with application processes for data reduction or in-situ analysis.

2.2.3 Library Interfaces for Measurement and Control

Performance tools need to intercept certain functions to ensure that tools have the opportunity to perform bookkeeping when the function is invoked. Examples of functions that need to be intercepted include those associated with creation and finalization of threads; initialization and finalization of processes; and signal delivery to name just a few. Instrumentation of such functions is used by tools to ensure that necessary tool control operations, such as initialization or finalization of performance data for a thread, can be performed at appropriate times. Tools use one of two approaches to intercept such operations: *interface wrapping* or *binary rewriting*. Library interfaces must be designed to make intercepting interface operations easy.

Interfaces for performance introspection and adaptation To enable system software and applications to adapt their behavior runtime in the face of their changing needs and platform state, many of the layers of the software stack performance introspection, which enable applications and other parts of the software stack to observe characteristics of their own performance. We expect that applications will need to measure and tune the performance of communication, I/O, and runtime systems as an execution proceeds. To support adaptation, it must also be possible to tune the characteristics of different software layers. Such tuning might come in the form of adjusting resource provisioning, or changing management policies. The mechanisms to support such capabilities across all layers of the software stack are not well understood at present. Understanding exactly what is needed will require further investigation.

2.2.4 Compiler and Runtime Support for Performance Attribution

Compilers record several kinds of information necessary for attributing performance information back to an application and runtime libraries. The most important kinds of information for this purpose are addresses of function entry points, a line map for relating machine instructions back to source code lines, and debugging information, which includes details about inlined code. Today, many compilers fail to provide adequate information for mapping performance back to source code. To avoid problems understanding the performance of exascale systems, procurements of exascale systems must specifically require that compilers and runtime systems provide mappings with suitable accuracy. Run-time libraries for execution models will need to provide support for logical unwinding of call stacks so that costs can be attributed to their calling context. Further, the run time system stack must export any adaptivity decisions to allow tools to track changes in the system and to provide the necessary translation for performance data.

2.3 State of the Art

Trace-based tools for measuring parallel performance, e.g., Vampir [28], Falcon [18], Jumpshot [57], MPICL [55], Caubet et al.'s tracer for OpenMP execution [7], the EPILOG tracing mechanisms [52], TAU [26] or Open|SpeedShop [37], are commonly used to provide insight into time-varying aspects of an application's behavior. However, on large-scale systems comprehensive tracing can be costly and produce massive trace files [49]. Tools like Scalasca [54] therefore provide options to automatically analyze traces and deliver only relevant information to the user. As another approach to control the size of traces yet provide insight into time-varying behavior, the HPCToolkit performance tools

introduce a new paradigm for trace analysis of parallel programs based on traces of call stack samples [45].

In contrast to traces, performance measurement approaches that collect summaries based on synchronous monitoring of library calls (*e.g.*, [49, 50]) or profiles based on asynchronous events (*e.g.*, [1, 11]) more readily scale to large systems because they yield compact measurement data for each thread or process; the size of this data is largely independent of execution time. Synchronous monitoring of communication calls, *e.g.*, by mpiP [50] or Photon [49], yields detailed information about communication activity but only coarsely summarizes computation. In contrast, call path profiles collected using asynchronous event triggers can attribute all costs in a parallel execution (*e.g.*, computation, data movement, or waiting) to the full calling contexts in which they are incurred [1, 31]. An advantage of such asynchronous monitoring is that it can provide detailed attribution of costs everywhere rather than just at interface functions as synchronous monitoring does.

Tools for measurement and analysis of parallel application performance are often model dependent. Examples include libraries or instrumentation for monitoring MPI communication (*e.g.*, [28, 49, 50, 56]), interfaces for monitoring OpenMP programs (*e.g.*, [7, 25]), or global address space languages (*e.g.*, [43]). In contrast, sampling techniques used by tools, e.g., `gprof` [16] and Open|SpeedShop [37], or call path profiling, such as that used by HPCToolkit [1] and Oracle's Sun Studio [31], are model independent.

Performance tools also differ with respect to their strategy for instrumenting applications. TAU [26], OPARI [25], and VampirTrace [3] among others, add instrumentation to source code during the build process. Model-dependent strategies often use instrumented libraries [7, 22–24, 49]. Other tools analyze unmodified application binaries by using static instrumentation (e.g., SGI's SpeedShop tools [39]), or dynamic instrumentation (e.g., Dyninst [5]) or rely on library preloading, as do SGI's perfex [11], PapiEx [27], Open|SpeedShop [37], and HPCToolkit [1, 15].

Tools for analyzing bottlenecks in parallel programs are typically *problem focused*. Strategies based on instrumentation of communication libraries, such as Photon and mpiP, focus only on communication performance. Vetter [48] describes an assisted learning based system that analyzes MPI traces and automatically classifies communication inefficiencies, based on the duration of primitives such as blocking and nonblocking send and receive. EXPERT [53] also examines communication traces for patterns that correspond to known inefficiencies. In contrast, general toolkits, e.g., HPCToolkit [1], Open|SpeedShop [37], and TAU [26], are *problem-independent*.

Performance analysis tools analyze scalability in different ways. mpiP [50] uses a strategy called rank-based correlation to qualitatively evaluate the scalability of MPI communication primitives. An MPI communication routine is said not to scale if its rank among other MPI calls performed by the application increases significantly when the number of processors increases. Using differential analysis of call path profiles, HPCToolkit can pinpoint and quantify scaling losses in parallel programs regardless of their cause [10, 46].

Memory hierarchy simulators, e.g., CacheGrind [29] or Sigma [13], can provide detailed feedback about memory hierarchy performance; however, such simulators may slow down an execution by two to three orders of magnitude. In contrast, Acumem ThreadSpotter [41] uses sampling techniques to gain insight into memory hierarchy performance at much lower overhead.

Lightweight tools, e.g., IPM [40], provide summary statistics about the amount of communication and computation by process. While such summary information is too coarse to guide performance tuning, it can provide insight into an application's overall performance.

Most performance tools that gather detailed measurements with hardware performance counters present raw measurement data. In contrast, PerfExpert [6], which uses HPCToolkit to gather performance data using hardware counters, attempts to diagnose the nature of node performance

bottlenecks by assessing whether they are due to the memory hierarchy, floating point instructions, integer instructions, or branches.

Basic performance visualization is typically done manually relying on tools like *gnuplot* or *Excel*. Some tool sets, like TAU [26], enable more sophisticated performance visualizations by providing displays for a range of configurable metrics [19]. Further, several tools exist that display timelines for message traces (e.g., Vampir [28], Jumpshot [57], or Paraver [32]) or call path traces [45]. Recent research has focused on using scientific visualization techniques to explore performance problems [38].

In addition to the aforementioned performance tools, several groups have been working on parallel tool infrastructures that support the development of scalable tools. Most notable in this area are Dyninst [4], a library for dynamic binary instrumentation; MRNet, the Multicast-Reduction Network [34], which allows efficient data management for large scale tool environments; P^NMPI [36], an infrastructure to virtualize MPI tools; and the Component Based Tool Framework (CBTF) [8], a generic infrastructure for assembling tools from independent components.

3 Debugging and Correctness Tools

The increased complexity and core counts of exascale systems will diminish the effectiveness of traditional interactive debuggers. To cope with the complexity of exascale executions, application developers will need additional tools that can help them to either automatically or semi-automatically reduce the problem to smaller core counts or to detect the problem itself. Tool support for debugging at exascale can and should range from simple approaches that cluster processes into similar groups to automatic root cause analysis tools that directly point users to the most probable causes for observed behaviors. Further, many solutions will no longer be either static or dynamic, but rather require an integrated approach that is capable of combining static information extracted from an application's source or binary code with dynamically gathered and aggregated data.

3.1 Research Challenges for Correctness Tools on Exascale Platforms

There is a wide range of issues that challenge the development of correctness tools for exascale platforms. These issues stem from the extreme scale of such systems, increased complexities in the architectures, and new programming models and languages.

3.1.1 Scaling Debugging Techniques

While many debugging and correctness problems can be reproduced at smaller scales, where they can be analyzed more easily, there will always be problems that can either not be reproduced at smaller, non-production scales. As a consequence, we will need tools, including debugging and correctness tools, that can operate at the full scale of the machine. To achieve this, we need to provide tools such as MRNet to the tool developers that simplify the construction of tools at scale; the goal is to lower the barrier to entry for innovative and small-project researchers to experiment in this area.

Such tools will be necessary to debug both applications themselves, which will be expected to scale to the size of the whole machine for capability runs, and the system software stack. The latter often cannot be fully tested at the vendor site before delivery, since vendors often don't have access to the largest system configuration, in particular in the context of DOE machines. As a consequence, we not only need system level debugging tools, but we will also need to provide users with tools that can help distinguish between application and system problems.

To scale debugging and correctness approaches to the expected billions of threads, three general approaches are seen as possible paths towards exascale debugging and correctness tools: support for reducing the problem to smaller scales; techniques for full-scale in-situ or online analysis coupled with a high-level presentation of aggregated data; and automatic error detection and root cause analysis systems.

Problem Reduction Techniques

The first step for debugging a problem at large scale is for the user to reduce its scale and try to reproduce the problem with fewer number of nodes. To enable this kind of reduction, debugging tools must support group operations that allow users to manipulate or advance all processes or threads in one group in a single operation. The same is also true for graphical user interfaces, where any information must be reduced so it can be properly displayed. Displays that only list all processes or threads will no longer be feasible. Similarly, values gathered from multiple processes and displayed with a tool must be aggregated and displayed using statistical metrics, but also allow users to closer examine the distribution if it is of interest.

Additionally, tools may be necessary to support such a downscaling, e.g., by increasing the likelihood that problems can be reproduced at small scale. This is particularly important for detecting and fixing non-deterministic problems, such as race conditions, and can include mechanisms to enforce task interleavings or perturbation layers that introduce noise to randomize execution.

Automatic Analysis Techniques

In many cases such groups of interest may also be detectable using automatic techniques. Those can include outlier detection, various forms of clustering, and automatic model generation. In all cases, the analysis attempts to identify behavioral classes in the gathered data and then tries to either group or classify the data. This can either be used to select individual representatives from each set or to focus subsequent analysis steps onto a subset of suspicious tasks.

Another technique to reduce data, typically used in performance analysis is sampling, i.e., the statistical selection of subsets of data. It can help identify larger groups or sets of data and can also be used to support clustering algorithms. For the detection of outliers or individual faults, however, sampling must be applied carefully and new techniques may be necessary. Otherwise, it is easy to miss singleton events or small sets since they may not show up in the sample.

Overall, such data reduction analysis techniques can form the basis for subsequent analysis steps. An example for this could be the lightweight recording of reduced traces from a subset of processes. Such traces can then be used for symbolic replay approaches that can recreate the global system behavior based on a small set of tasks. However, the enable this layering of tools, it will be paramount to develop interoperable and composable tools and appropriate data formats or APIs that enable efficient data sharing.

Analysis techniques should be able to do provide the analysis during the initial application run. Machine time, especially for full scale runs on large scale machines, is typically highly contended leaving little room for multiple executions at full scale. Further, in case of non-deterministic problems, it is highly advantageous to be able to monitor the initial run and then not having to wait for the problem to re-manifest itself.

The analysis should therefore be run in situ with the application or concurrently in an online mode. This will require new techniques for efficient and light-weight tracking of groups and possibly a closer integration into the underlying runtime system. This naturally causes additional challenges both regarding the infrastructure and the amount of perturbation the tool causes. Both, using additional external processing resources, e.g., in the form of a tree based overlay network, or supporting the analysis with additional static information collected before the execution could be viable techniques to mitigate these effects.

At the same time, though, debugging tools must be able distill and store their online results for a comprehensive post-mortem analysis. For problems that require a long human analysis time it is not feasible to block machine allocations. Effectively, tools must separate "think time" from "machine time" whenever possible. This requires intelligent mechanism to select and extract data that is most likely required by the user to diagnose the problem post mortem.

Root Cause Detection Techniques

The final goal of debugging tools is to provide a precise identification of where the error is in the code or the underlying system rather than to report where it occurred. Such root cause analysis techniques have the potential to provide the largest benefit to the end user. Initial approaches in this area, e.g., analyzing statistical bug reports exploiting the similarity between tasks, or debugging coupled with static dependency and control flow analysis, are promising, but have not matured to the point that they are useful to code developers, yet.

3.1.2 Debugging Hybrid and Heterogeneous Architectures

Exascale system are generally expected to provide some kind of hierarchical programming approach. Debugging and correctness tools have to follow this trend and provide adequate support. Exascale heterogeneous architectures that combine Graphics Processing Units (GPUs) and traditional CPUs within the same compute nodes present numerous challenges with respect to debugging. Nevertheless, we expect developers in search of high performance to be more aggressive in using sophisticated techniques such as dynamic load balancing and asynchronous communication, and we also expect exascale systems to be more susceptible to component failure than current systems. Thus, the need for debuggers that will be effective on exascale heterogeneous GPGPU-based systems is extreme.

The challenges of debugging programs running on exascale heterogeneous GPGPU-based systems involve the greater complexity of their compute node architecture, the larger number of threads of execution running on each node, and the additional layers of abstraction between the code the user writes and what actually executes on the node's compute devices, as compared with the situation on traditional HPC systems. The node architecture of GPU-based heterogeneous systems is more complex than that of traditional systems. In particular, the GPUs in current GPGPU-based systems use a different instruction set architecture than the CPUs, and we expect this characteristic to be true for many GPGPU-based heterogeneous systems as we move toward exascale.

Because most current approaches for developing GPGPU programs combine the CPU and GPU code into a single executable file, users will expect to interact with a single tool when debugging both types of code. Thus, this single tool must be able to monitor and control threads of execution running on both CPU and GPU, parse both types of executable code, inspect and modify data held in potentially separate memory hierarchies and register files, and know how to traverse the execution stacks of those disparate devices — simultaneously.

Furthermore, the number of threads of execution within each compute node is expected to be much larger than that for traditional HPC systems. System-provided interfaces for monitoring and controlling threads has been notoriously fragile even with small numbers of threads per node, and we expect them to be no better as the number of threads per node to monitor is increased by several orders of magnitude. Yet a debugger is expected to deal with failures of such system-provided support without propagating such failures to other parts of the system, and without failing itself.

Also, in the exascale time frame we expect many users to program GPUs using higher-level abstractions than today's approaches like CUDA and OpenCL. Debuggers must allow users to interact with their running code at the level they programmed, but also descend to lower levels if necessary to understand the nature of a problem.

Even today, the development tools used to produce GPGPU-based programs do not expose enough information for the debugger to do this effectively (or the information they expose is incorrect). Despite these challenges, vendors like Rogue Wave and Allinea have already introduced products for debugging GPGPU-based programs running on many of today's important HPC architectures. These products are steps in the direction needed for effective exascale debugging, but they are limited (e.g., they only support CUDA programs). Much more work is needed on formal models and techniques for efficient debugging on GPU-based heterogeneous exascale systems.

3.1.3 Specialized Memory Systems

Managing and accessing memory efficiently is one of the most crucial challenges for exascale systems. This is due to several factors: memory per core is expected to shrink as we move to multi- and many-core systems; the memory subsystem (both statically and dynamically caused by data movements) will consume a significant amount of the system power budget; and data locality, i.e., reducing data movement, will be essential for both optimizing performance and power consumption.

These trends are likely to lead to new hardware developments, including but not limited to the deployment of scratch memory, adaptive cache architectures, and sophisticated prefetching techniques. Further, future systems will likely provide only limited coherency guarantees with chances for even cross-chip cache coherency slim to none. Additionally, hardware support for speculation (for transactional memory or speculative execution) will require multi-variant cache and memory structures, further complicating future memory systems. Such new features will either be provided transparently to the user or require manual management in the user's code. In both cases, though, we will require new debugging support.

In the case transparency is provided by the system, the debugging interface for the user should not change in most cases, but debugging tools themselves will have to be able to understand how a particular execution is mapped to the underlying system. Further, if transparency is achieved through automatic code transformations before execution, we need additional mechanisms to verify the correctness of such transformations to guarantee an equivalent execution. In both cases it is necessary for the underlying system to provide the necessary debugging interfaces exposing how code is executed on the underlying hardware and software stack.

In the second case, with the user responsible for managing new memory systems features, tools will have to have the capabilities to expose and present such hardware additions. For example, for explicitly managed scratchpads, debuggers will have to be able to display scratchpad contents, for multi-variant caches it will be necessary to explicitly view individual versions, and for non coherent systems a debugging tool must offer users to examine differences caused by different views on the same data.

Additionally, since such new and enhanced memory systems are likely to increase code complexity as well as the likelihood of bugs in codes using them. Correctness tools both dynamic (for dynamic checking of assertions) and static (for the identification of likely problems) will be of great help to the end user.

3.1.4 Domain Specific Languages

Domain specific languages are one of the promising approaches for programming exascale system. Their main advantage is that they allow programmers to specify their problem at a higher level. The system will then translate this information, while exploiting domain specific knowledge, into the code that will be executed on the machine. While this kind of abstraction is convenient for the user, it comes with the additional challenge that tools must provide the same view and map its

data to the same abstraction, in addition to supporting low-level views within the implementation.

This approach leads to two important challenges: debugger and correctness tools will have to focus on several different audiences and provide the necessary information for both basic users who rely only on the provided abstractions and advanced users who have a solid understanding of the system complexities and are willing to break abstractions where necessary to achieve performance. Each of these groups requires a different view and hence different debugging support. This could either be achieved through an integrated tool that allows users to switch perspectives or a series of targeted tools for different user groups. In the latter case, though, tools must be interoperable and allow for an integrated workflow between all tools.

Second, the principle of domain specific languages is that they are designed and written for a single domain and hence by definition not useful for general program development. As a consequence, there will be a large number of such languages available to cover all domains. It will not be possible to create custom tools for each of these languages or adapt an existing tool to work with all language individually. We will therefore need a common interface across domain specific languages that allows debuggers as well as other tools to reason about the abstractions provided by the language and how they are mapped. To provide useful information to high level programmers, mapping between levels of abstraction will need to become a first class and extensible feature of debuggers (and other tools as well). Mapping must support both forward (from programmer abstraction down to machine detail) and reverse mapping (from hardware details back up to programmer abstraction). Such mapping information must include both static information gathered at compile time (e.g., through extended DWARF support) and dynamic information gathered by the domain language specific runtime.

3.1.5 Mixed Precision Arithmetic

Exascale systems are likely to employ mixed precision arithmetic. Due to the power requirements of computation, and the bandwidth to move data, it will no longer be possible to conduct all arithmetic in IEEE double precision. Moving to mixed precision provides great opportunity to gain performance, but also can introduce subtle errors into programs. Correctness tools to help programmers develop both mixed precisions algorithms and to test and debug them will be required if the promise gains from mixed precision are to be achieved. Such tools will likely include both tools to help check the accuracy of converted code, and also automated tools to assist in identifying where in a program mixed precisions might be appropriate.

3.1.6 Adaptive Systems

Exascale architectures, systems and applications are expected to be highly adaptive to react to changing system conditions. This includes adaptivity for fault tolerance, load balancing, power management, communication optimization and data locality. If this adaptivity is fully hidden from the user by the system, the existence of such adaptivity should only play a minor role for functional debugging. As above, any debugging or correctness tool will have to be aware of the adaptivity and provide the same level of transparency. Depending on how the adaptivity in the system is implemented this additional requirement could range from no necessary change to a significant monitoring of any adaptive decision.

However, if any of the adaptivity is exposed to the user, debugging tools need to be aware of any adaptivity in the system and present it to the user in a way that is easy to understand and can be combined with the rest of the debugging information. This will require the tools to monitor any adaptive component in the system and to correlate the changing system information with

the debugging information gathered during the application's execution. To enable such debugging approaches, any adaptive component will have to expose its decisions to the rest of the system through a set of APIs.

An additional property of adaptive systems is that they lead to non-deterministic executions since decisions are made a runtime and change from run to run or even on a per timestep basis. This can lead to new and previously unobserved behavior during each application and can lead to the exposure of latent bugs in the application or the system stack. Further, left uncoordinated, multiple adaptive system components might make conflicting decisions possible leading to worse performance or, in the extreme case, deadlocks or incorrect executions. This is yet another case for which users need to be prepared and will require tools that help distinguish what happened and how to fix it.

3.1.7 Correctness Tools

The development of software for future architectures is expected to be more complex. A significant amount of complexity will be put on the application developer to write software correctly. Correctness tools have an opportunity to improve the productivity and make the development of future software more tractable to build. The development of HPC software using just a compiler and editor may be relegated to history; the expected complexity of writing future HPC software may require a range of specific correctness tools to provide portability and performance.

Tools for correctness can range from tools for finding bugs (logical or performance bugs) to tools that validate numerous properties via proof techniques. The goal of correctness tools is to identify problems in the code as part of its development and guide the application development around such problems. Tools may be specific to individual source code languages, operate on the binary executable, or both.

Infrastructures to support building correctness tools can significantly simplify and make it tractable and economical to build new classes of tools. Compiler and/or binary analysis support and infrastructure may be integral parts of the development of new classes of tools.

Static Analysis Tool Research

Static analysis has long been a basis for the analysis of software for security and identification of numerous bugs and portability issues. Such work often requires either source code or binary executable analysis infrastructure; but static analysis has limitations, usually with regard to imprecision in context sensitive analysis and/or pointer aliasing (which is technically undecidable). The analysis of large scale codes is further complicated because analysis can be non-linear in both time and space. However, modern machines have grown significantly large in memory and performance and previously unreasonable approaches are today quite tractable. The HPC specific requirements for static analysis tools further narrow the types of problems that should be identified for DOE applications. As a result, static analysis based tools may be of significant use in the development of HPC code. As examples, static analysis tools might be tailored to:

- detect locality and parallelism, and guide the user in exploiting it,
- enforce programming model usage rules (tailored to each programming model),
- detect correct use of synchronization,
- identify memory access patterns,
- support automated extraction of performance models, or

- provide data for dynamic analyses.

Model checking is a type of support that has been popular partly because where it detects a problem it provides an example of how it was derived. Such tools that automatically emit counter examples to rules that are enforced can be extremely useful to support application development because they communicate narrowly and well defined problems and have impact greater than a list of issues that could be mostly false positives. Model checking tools narrowed to address the requirements of specific HPC issues could be especially useful in supporting HPC software development, e.g., for MPI and other specific programming model implementations.

Dynamic Analysis Tool Research

Dynamic analysis tools address some of the problems of static analysis tools, but add new problems such as dependence on inputs and performance penalties. However, dynamic analysis can be the only way to know some specific and required information to make an analysis more precise. As an example, dynamic analysis is able to see and support analysis of the data layout in the heap, which is generally at the limits or beyond the capabilities of static analysis (e.g. static shape analysis). Dynamic analysis can be used with instrumentation infrastructures at either the source code or binary executable level; each with some specific advantages and disadvantages over the other. Dynamic correctness tools can further exploit debugging or profiling interfaces within the system stack, where available, to enable a precise and lightweight instrumentation.

Dynamic tools have been successfully applied to checking the correct usage of APIs. In particular for MPI several tools exist that detect incorrect usage of the MPI API as defined by the MPI standard and report such behavior to the user. Similar approaches could also be used for other APIs, such as CUDA, the OpenMP runtime, or even higher-level math and numeric libraries.

Mixed Static and Dynamic Analysis Tool Research

This approach to building tools provides the best of both static and dynamic analysis, but often is not done because its requires expertise that spans disciplines that frequently don't interact. Research groups doing both static and dynamic analysis are a bit more rare, thus such tools are less common. Mixing these forms of analysis together permits dynamic analysis to be used more efficiently (optimized) via static analysis, e.g., by augmenting dynamic analysis or specializing dynamic tools to particular applications. This can ultimately lead to automatically generated application specific tools. Conversely, static analysis can benefit from runtime information that can make it more precise, at the risk of only being a bit more specific to the program inputs.

3.2 Cross-cutting Issues

Besides the debugging and correctness tool specific issues discussed above, we also have to face a range of cross-cutting issues, with respect to both external constraints and challenges on debugging and enabling technologies.

3.2.1 Dealing with Faults and Fault Tolerance

The rising complexity and size of future machines will have a dramatic impact on the probability of faults. Further, the need for power reduction may force chip developers to produce less reliable architectures, forcing some of the fault handling to software. All of these trends lead to fault tolerance and resiliency being one the key design challenges for exascale systems.

This has several severe implications on tool developments in general and in the area of correctness and debugging in particular: a) tools need to be resilient themselves; b) tools must coordinate

with our layers to be informed about faults; and c) we need a new class of correctness tools that helps ensure the correct execution of code, even on faulty hardware.

First, tools must tools be resilient themselves to continue working even after an application has suffered a failure. In particular for debugging tools, the tool must survive crashes and help the user determine whether a system or an application problem caused the application to fault.

At the same time, if part of the system fails, but the application can recover and continue, the tool infrastructure must detect this and adjust itself. For example, after a checkpoint/restart, tools must track the newly created code, carry over any state collected before the failure and continue running. This particularly important for long running and interactive tools, such as interactive debuggers.

Overall, this puts high requirements on the collection infrastructure as well as the graphical user interface. Changes in the system, such as migrated tasks must be reported to the user, any state tracking must be adjusted, as well as displays updated.

For the third point, the need for a new class of correctness tools, we need to distinguish different fault types. For soft and transient errors, such as memory corruptions, tools could provide help in early detection mechanisms for memory locations, or could be preventative by using replicated execution strategies. If application programmers are involved in writing fault detection or recovery mechanisms, then tools to support this will be required. Such tools might include programmable fault injection systems to make rare events happen on demand. Other fault types like start/stop scenarios, require a different handling and will most likely build on infrastructure that enables a dynamic view on the application's execution and can tolerate a varying number of nodes.

3.2.2 Dealing with Power Constraints

Debuggers, like any other tool, will require their own resources, in particular when run at scale. This can include extra threads on compute nodes, additional dedicated processing nodes, or increased I/O pressure caused by extensive logging. In all cases this will also have an impact on the overall power consumption of the system.

If the system is running at power capacity limit or is managed by an adaptive runtime system that continuously tunes an application to be on the limit, we cannot deploy tools without changes to the system. This can be as simple as reducing the speed of the application or instructing the runtime system through a steering API to leave room for the additional demands of the tool. This will perturb the application and debugging tools have to take the resulting skew into account.

3.2.3 Shared Infrastructures

Designing and developing scalable solutions for debugging and correctness tools requires a significant amount of engineering work as well as novel research and solutions in the underlying data transport, analysis and storage infrastructure. We can no longer afford to redevelop this underlying work over and over again for each tool.

The needs for debugging and correctness tools with respect to the necessary infrastructure will be quite similar to those of performance tools. Further, there is a large overlap with other components of the system software stack, including the I/O system and the data analysis and visualization pipelines. We will explore these infrastructure aspects in more detail in Section 4.

3.2.4 Co-Design Needs and Opportunities

Debugging and correctness tools rely on information from virtually any other part of the system stack. They therefore require the necessary APIs, especially in newly developed components of the

exascale ecosystem, ranging from establishing new APIs to enable tools to communicate with the system and runtime stack to debug information in domain specific languages. Further, debugging tools could make use of additional tracing and monitoring capabilities in the base architecture. For example, hardware mechanism to trace memory accesses could help the design thread checking tools, new hardware counters in particular in the memory and network subsystem allow new measurements, and hardware supported memory watch points can enable fast state tracking for interactive debuggers. An additional feature that is helpful for debugging tools (and beyond) is a global time stamp that enables a clean correlation of events across nodes, processes and threads. The latter is especially useful for debugging adaptive systems where a temporal correlation of changes in the system is essential.

3.3 State of the Art

The state of the art in debugging and correctness tools consists of a variety of tools supplied by hardware vendors, independent software vendors (ISVs), and University/Laboratory groups. Each of these sources contributes to the overall HPC tool ecosystem. In addition, users still find manually debugging with print statements to be useful. However, given the number of threads expected at exascale, even getting a few bytes of output from each thread via a print statement will likely not work.

Most of the hardware vendors supply tools with their systems. Both Cray and IBM supply tools designed to work on their integrated HPC platforms. The Cray tool suite is more focused on performance tools (such as CrayPat), but for debugging they rely on ISV supplied debuggers. The IBM tools consists of primary of pdbx, which is a basic parallel debugger designed for their shared memory HPC systems as well as ISV supplied debuggers.

Two major debugging tools are currently available from ISVs. DDT (from Allinea Software) provides traditional breakpoint based debugging for parallel computing. To help manage complexly, it includes a process group viewer. It also includes a parallel stack view that shows a merged view of all of the stacks of the current threads. Totalview (from Rogue Wave Software) provides traditional breakpoint debugging, but like DDT includes commands to manage groups of processes. Totalview also includes an array examination capability to allow users to look at slices of their arrays both as tables and graphically. Both DDT and Totalview now include some capability to debug CUDA code. However, both systems assume that code is written directly in CUDA, and seem to lack tools support for code compiled into CUDA. There is also an effort to create an Eclipse-based parallel debugger plug-in called PTP.

At the DOE laboratories and universities a number of innovative debugging and correctness tools have been developed. These tools vary in maturity from proof of concept research ideas to stable tools used in production environments. The Stack Trace Analysis Tool (STAT) [20], developed jointly by LLNL and the University of Wisconsin, provides a lightweight tool to identify and cluster stack traces from a large number of nodes. Another example of correctness tools are memory utilization tools. An early lab tool for this topic is MUTT from LANL.

Another rich area of correctness tool research has been race detectors. A number of tools have been built using either the lock-set [35] or all-sets [9] algorithms. Both of these approaches work well for small scale cache coherent shared memory, but can be subject to large runtime overheads. Recent work [17] has looked at using exisitng hardware to try to accelerate race detection. Also recnetly, new approaches to make classic vector clocks faster have been developed [14]. Despite these advance, run-time overhead is still so high that many programmers either will not use such tools or are discouraged by the perturbation caused by these tools. In addition, the problem is exacerbated by the increasing core count on processes; handling more threads compounds the overhead.

4 Cross-cutting and Tool Infrastructure Issues

From the challenges laid out in the sections above it has become clear that stove pipe solutions, as they are currently common, are no longer feasible and will not enable us to provide application developers with the capabilities they need on exascale systems. In particular, the following crosscutting aspects need to be considered:

- We are facing challenges, with respect to changing programming and execution models, complex architectures and applications, and extreme scalability requirements in all parts of the system layer, that cannot be solved by a single tool. Instead, we need a set of interoperable and compatible tools — static and dynamic — that can help users tackle individual problems as they arise.
- This required flexibility can only be achieved by componentizing tools and making each component available as part of a tool set through a set of compatible APIs. This has the advantage that each component can be developed, optimized, and maintained separately reducing duplicated effort. Further it provides the ability to quickly prototype new tools for newly arising challenges.
- Looking at the broader picture, the development of effective tools will require a close collaboration and interaction with the entire system stack. In fact, to fulfill many of the requirements posed by the user community, tools need access to more in depth information across all system layers than is currently available
- Tools need to be available early in the development process of both applications and the system software stack, especially in the context of successful co-design, since they will not only be used to fine tune an almost completed application, but rather must be available to accompany the full development process, in particular on new hardware or software architectures.

It will be important to clearly define these cross cutting issues and API requirements as well as responsibilities for their implementation. Further, these APIs need to be system independent to allow for portability of tools across a wide range of architectures.

4.1 Crosscutting Demands on Tools and Tool Infrastructures

Exascale computing presents at least two particular sets of cross cutting challenges for the tools community that are driven by the requirement to deliver an exaflop of computation in 20 Megawatts or less [2]. An "evolutionary" path towards exascale that embraces a traditional execution model will be forced in the direction of supporting billions of extremely light-weight, power efficient cores with extremely limited per core memory and bandwidth and strict requirements on data locality. This "evolutionary" path places unprecedented scaling requirements on existing tools and drives the need for new tools to address fine grained analysis of energy consumption and data movement through the memory hierarchy.

However, a "revolutionary" path toward exascale, in which old paradigms are discarded in favor of possibly radical new approaches to the manner in which applications interface with hardware, may drive the tools community in an entirely different direction in support of new algorithms and methods. Possible outcomes of the "revolutionary" path may involve the development of new modes of processing in which data is static and operations are dynamic, frequent errors are not only acceptable but expected and correctness is no longer based on strict determinism or reproducibility.

How does one perform correctness checking or even debug, for that matter, a code that only needs to be "good enough"?

As a consequence, the tools community depends heavily on the architecture, programming models and systems software communities to accurately and thoroughly specify the execution model that will become the exascale ecosystem for the tools community. In addition to the execution model, which defines the "sorts of actions" that an application will and will not be able to perform on a given architecture, the API's that specify communications across the system stack are critically important to the tools community. What sensors, monitors, counters and controls will be available to tools in the exascale timeframe? This information is critical to delivering successful tools for exascale.

4.2 Connection of HPC to How Embedded Software is Developed

The embedded software community has a long history of using tools to support their software requirements. They have been forced to deal with the constraints of low power for a long time and in this way their history may foretell how HPC software might be developed in the future. The next generation of hardware for HPC may share significant features with embedded processors if both are driven to use similar processors with similar demands on power efficiency. In the embedded software community there has been a significant reliance on deep tool chains to support software development and the same should be considered for HPC software development. The popularity of tools has been driven by both productivity and software complexity, the same issues may drive HPC software.

The extent to which the exascale tools community will have the opportunity to leverage tools available to the embedded community depends largely on the proposed system architecture and execution model. Software development tools in the embedded community fall largely into two classes: (1) design tools for architecting a system and (2) development tools for programming the system. Traditionally, and important difference between embedded architectures and DOE architectures for scientific computing has been the focus of flexibility. Embedded computing offers a high degree of flexibility in designing the system from the circuit level up, and the tool chain reflects that. Scientific computing, on the other hand, has tended to embrace commercial components for both system hardware and software. On the flip side, embedded computing frequently targets a very limited range of applications for a particular architecture, possible even one. By way of contrast, scientific computing focuses its diversity at the application end, and once again the tools must reflect that diversity.

To first order, the extent to which tools developed by the embedded community can benefit the exascale design and development efforts will be a function of the degree to which exascale chooses to pursue architectural diversity and algorithmic specialization. In that case, an "evolutionary" approach to exascale that emphasizes extrapolation of traditional architectures and execution models will find less overlap with embedded computing in terms of tools. However, a "revolutionary" approach to exascale could in fact create the opportunity to exploit a wide range of commercial tools develops specifically for the purpose of designing reliable, highly energy efficient HPC systems. It is also possible that the same or even similar tools will not be used for both exascale and the embedded system communities, but that the HPC application development process will only be similarly dependent upon a different set of tools and require a similarly rich software development tool infrastructure.

4.3 Tool Interfaces, Abstractions, and API Issues

The development of tools supporting development and evaluation of large scale applications on future platforms requires some practical considerations. Specifically, we will need to address as a community: tool interfaces, connections to programming model abstractions, and general API issues.

4.3.1 Interfaces with Programming Models and Hardware

It is essential that tools allow the users to relate any information that is being gathered back to the application, its source code, and its data structures. This will only be possible if tools gain access to abstractions provided by the respective programming models. A lower level access to the programming model abstractions would be welcome as a way to connect and define tool interfaces. Current approaches only provide such information in a very limited way making it hard for tools to interpret the performance data and measure features being abstracted. We therefore need new APIs that can allow both the compilers and runtime libraries for programming models to deliver this information to tools.

Performance counters are the most common access to metrics at the hardware level, this limited level of insight into performance is not enough. To understand the performance implications we will require the evaluation of new metrics (e.g., for network traffic, for energy consumption, or for accelerators) and other introspection capabilities (such as memory reference tracing or external environment control). These capabilities must be accessible to tools through standardized cross platform API. Specifically, it must be possible to use performance counters safely in both caliper and sampling based tools.

4.3.2 Interfaces with OS, Runtime, and System Software

The requirements for good quality APIs extends to the OS level, runtime systems, and other system software. Support requirements include: debug interfaces, symbol tables, thread allocation, programming model and runtime system behavior, feedback and notification mechanisms in case of adaptivity in the system stack, and scheduling data as well as resource utilization. Even specialized node kernels (reduced functionality OS support on the nodes) have specialized requirements to support debugging for example. These requirements may exceed the more common API requirements of the applications. Programming models and Domain Specific Languages (DSL) will present abstractions and runtime systems to support them and these need to be exposed through low level API in order to support tools.

Further, many of the features discussed above, in particular the offload capabilities for asynchronous processing of performance data, require additional resources. Tools need interfaces to allocate and control these resources that are provided by the system software in a scalable manner. For example, tools need to be able to locate process and thread information, launch tool daemons on suitable nodes and request additional processing nodes for online analysis and data aggregation.

Dynamic tools, where tool analysis results will be used for run-time optimization, must be able to feed results and response options to the OS and system software. For instance, if a dynamic analysis leads to a discovery of application imbalance, such information must be able to used to initiate a coordinated response by system resource managers, applications, and system response capabilities such as process migration.

Finally, tools share infrastructure requirements similar to the data analysis, visualization, and I/O software stacks. Like tools for performance monitoring or correctness checking, data analysis tools will increasingly need to rely on in-situ or concurrent processing on an external set of nodes

to avoid streaming unwieldy amounts of data to storage. Common infrastructure for tools, data analysis, visualization, and I/O should be explored.

4.4 Requirements on Tools Infrastructures

In addition to new capabilities that tools need to offer to the user community, future computing systems also pose significant challenges to building and maintaining the tools. Tool infrastructures make it easier to build tools that would otherwise be intractable to support; and yet have a wide range requirements to make them suitable to tool developers.

4.4.1 Scalability

Tools themselves have to be scalable and must be able to operate efficiently across the entire machine. Each tool component must be scalable by itself and should not contain algorithms or data structures that scale linearly (or worse) with the number of processors. The tools themselves must use the parallel resources available in the system for their own processing. Debugging systems have special demands that are discussed in 3.1.1; these include efficient use of parallel resources. In contrast, correctness tools have different demands depending on whether they address static or dynamic analysis; or both. Frequently for static analysis performance and levels of correctness (fidelity) is associated with the size of the code or critical sections. Where as for dynamic analysis, scalability may be more closely associated with the amount of instrumentation and how it competes for resources (e.g., for memory usage). The scalability of tools will be increasingly important on future computing platforms (e.g., exascale).

4.4.2 Processor Architecture Heterogeneity

Processor architecture heterogeneity is expected to be more common in defining future processors where applications will be executed and where tools will have to run. Such system exist today and there is even initial experience with this type of increased complexity. Heterogeneity raises the complexity of both tools and tool infrastructures. Heterogeneity can have several faces. Examples are combinations of GPUs with CPUs or many-core CPUs containing both fat cores (fewer more powerful and less efficient cores) and thin cores (more numerous, low power, and more efficient cores). Future architectures many span a wide range within how they mix heterogeneous elements, thus further raising the complexity of tools. Performance tools have specific requirements, discussed in 1.2. For static analysis, tools have to understand GPU languages (e.g., CUDA and/or OpenCL) or some level of intermediate representation of the code (e.g., LLVM or PTX). For dynamic tools, there are additional levels of complexity associated with APIs that will provide information to support such tool chains.

4.4.3 Relevance of Power

Future computing systems will be more dependent on power optimizations, as discussed in 1.2. Power cross-cuts many aspects of tool research from optimization of applications, to communicating relevant metrics within performance analysis. In this way future platforms (exascale and beyond) will dramatically reshape how we do analysis (for performance and power) to optimize applications, what tools are most relevant to application groups, how the tools are designed to use architecture specific features, and how trade-offs are handled between performance in time and power. Power is of course cross-cutting to subjects far outside of the domain of this report, for example to the

mathematical algorithms that would be used and alternative communication latency tolerant forms of new and old algorithms.

4.4.4 Memory Constraints

Exascale platforms will be severely limited in the amount of memory they provide per core. Applications are likely to exploit the available memory to the fullest extent leaving only very little room for tools running alongside these applications. Therefore, tools have to be written with even more care regarding the memory they consume. Data structures that are linear (or worse) with respect to the number of cores are likely to severely limit the applicability of the corresponding tool.

4.4.5 Addressing the Languages and Instruction Sets Used within DOE

Where tools interact with source code or binaries, they must support the languages common to applications in DOE or, in the case of binary tools, operate on the instruction sets that are a part of DOE computing platforms. Tools have to operate on a wide range of platforms and across a wide range of applications to support HPC within DOE.

4.4.6 Fault Tolerance

Exascale tools will be required to integrate seamlessly into the fault-tolerance infrastructure of an exascale system, which could require a great deal of cross-layer coordination between application fault-tolerance interfaces and the remainder of the software stack as discussed previously in Sections 2.1 and 3.2.1. Particular consideration should be given to libraries provided by third parties or vendors that may or may not support fault-tolerant execution. In addition, correctness tools, performance tools and particularly debuggers will require some level of hardening to accurately collect and parse data generated by potentially faulty components or interfaces. Debuggers ought to be able to recognize and trap errors that generate failures and invalid data, but not mask them from the user. The scope of possible failure modes at exascale if far too broad for the tools community to address them comprehensively, so they will depend upon on the systems and applications communities to articulate a prioritized taxonomy (e.g., soft errors, permanent faults, or silent data corruption).

In some cases, users that implement a resilience strategy that attempts not merely to detect and contain faults, but to run through faults, may need tools to support robust debugging of resilient software. Research should explore the possibility of using debuggers to inject faults to test error handling and other resilience features of an application. Correctness tools are another area where research on fault-tolerance may bear fruit. Providing the static analysis tools to determine whether or not a code will "do the right thing" under certain fault scenarios may be of particular value for exascale. Tools and metrics that aid in evaluating the tradeoffs of power, performance and reliability are also of interest in the exascale timeframe. An integration of correctness tools, performance tools and power profiling might be useful in giving the user a better sense of the risks and tradeoffs in creating fault-tolerant algorithms.

4.4.7 Componentization: Tools and Tool Infrastructures

We will need sophisticated tools to address the complexities of the target applications and systems. No single tool will be able solve all problems - instead we will need the ability to create and maintain custom tools for particular problems or target platforms. Individual groups will bring specific expertise to the problems of developing tools for future platforms. The depth of the problems to

solve will best be supported by building on each others tools and infrastructures, and by defining a community to support the development of research tools and research tool infrastructure. Such approaches to support tools using componentization can be especially effective in permitting new tools to be assembled quickly.

Such component tool infrastructures for tools will not only avoid stovepipe solutions and enable interoperability between tools, but it will also enable quick tool prototyping and the creation of custom or even application specific tools. This will allow tool providers to quickly react to new, unpredictable problems and provide users with quick and direct support without having to create specialized tools from scratch.

Tool consumers can either be humans, other tool components, or other parts of the tool chain or the system software stack. In case data is delivered to the end user, any data needs to be attributed to code and data structures and presented in an intuitive manner mapped to the domains the user is most familiar with and that allow for an easy evaluation. This requires both new display and visualization techniques capable of displaying data gathered at scale and new analysis techniques that are capable of extracting all relevant information and features.

4.4.8 Flexible and Effective Instrumentation

Gathering data from a running program is a key technology for debuggers, performance profilers, testing tools, tracing tools, and many others. Flexible and effective instrumentation tools are needed for programs, both at the source and binary code levels. At the source level, such tools need to be incorporated into source-to-source translations systems. At the binary level, tools need to support both static instrumentation (binary rewriting) and dynamic instrumentation. Static instrumentation is crucial on leadership class systems where control of a process at runtime is too complex or expensive to use. Dynamic instrumentation is crucial in cases were adaptive techniques are needed, either to respond to runtime conditions or for efficiency reasons.

Ideally, such instrumentation tools would work in concert, allowing the simultaneous use of source code instrumentation, static and dynamic binary instrumentation, couple with runtime sampling. While pieces of this functionality have long been studied, such a comprehensive facility is still to be developed.

5 Path Forward

5.1 Summary of Support for the Evolutionary Path

An evolutionary path to exascale will focus on scaling of traditional programming models towards a goal of 1000x the Flops performance of current petascale systems. However, to meet the 20 MWatt power goal for exascale, computations will be performed by computer cores that are simpler and slower than those found on current machines, which means that the total number of concurrent threads of operation will grow much more than 1000x in the exascale timeframe. Thus, billion thread parallelism is likely, and extreme data locality will be required to minimize the joule per bit costs of moving data over wires between memory and CPU.

The high level implications for these changes are that debuggers need to scale to well over 1000x in parallelism and performance. In addition, correctness and memory tools will be needed that can provide developers with increased insight into optimizing fine grain parallelism and data locality in their codes. Techniques for performing analytics on data gathered by the tools should be explored, as the tools will be generating far more information that is manageable by a human. Research needs to look for new approaches to identify performance bottlenecks, suggest code transformations to improve data locality and synthesize debugging information to isolate bugs in codes running on a billion threads.

5.2 Summary of Support for the Revolutionary Path

The revolutionary path to exascale involves adopting new execution models for exascale, that will in turn drive revolutionary architectures and programming models. The impact on tools would be a paradigm shift on what kind of data is available to the tools, and what kind of information is expected by the developer. For instance, one might imagine programming in an environment where there are more threads of execution than pages of memory, in which case one might want to debug from the perspective of the memory unit as the fixed reference point with millions or billions of threads moving in and out of it and modifying its state. Basic research is needed to understand what kind of tools are needed to support such a model of programming, and what kind of tools can help provide "backwards compatibility" of legacy codes within the context of the new model of execution.

Other concerns on the revolutionary path could include tools that allow users to optimize for power or even reliability, as opposed to performance, and analytics tools that perform discovery on the dynamic data collected from the running application. The amount of performance and debug data being generated on an exascale system will in itself be exascale, which means that research is needed to determine what data to collect and how it will be used in diagnosing performance, reliability, power and correctness issues on the system. Further, research will be required to define techniques and interfaces for gathering and communicating information across layers of the system stack in the new execution model, which may not be process based as our traditional models have been.

5.3 A Sustainable Robust Infrastructure for HPC Tools

Clearly, application users require robust tools to support their research work. The tools themselves must be significantly more robust than the applications where they are being used. This requires significant attention to the software development of tools, their testing, evaluation on different platforms, and open-source release into the DOE community. The road to production-ready tools is long and frequently not supported wholly by research funding. This will be an ongoing difficulty

for research groups attempting to release production-ready software components for the tool community to build upon, and the application groups to use such tools to support HPC application development. It will become more complex as the architectures become more complex; and maybe especially difficult for future exascale platforms. DOE may need to consider creating an organization separate from research that is chartered with transitioning tools and tool infrastructure from research to production and maintaining then long term.

In order to provide production-ready software for both application developers and for the tool development across an increasingly diverse set of architectures, we emphasize the need for a sustainable, robust infrastructure for HPC Tools, which includes the following:

Interoperable APIs to enable componentization. Interoperable APIs for tools provide numerous benefits, including promoting reuse, portability, easier maintenance, composability, and modularity. The research community should strive to outline, prioritize, and implement components and pertinent APIs for the necessary exascale tool functionality.

Modular, reuseable components for scalable tool infrastructure. Modular tool components that allow the development and composition of a set of tool functionality necessary for solving specific performance analysis or debugging tasks. This infrastructure should provide components using a variety of mechanisms that include libraries, runtime systems, software source code, user interfaces, and standard APIs.

Integration of tools with the target hardware and software ecosystem. Successful software tools require intimate knowledge of the target architecture and software system. Vendors must provide this knowledge to external developers in some form, either by adhering to standards or by providing specifications, documentation, software, and early access to systems. Furthermore, future systems are unknown at this time, and most importantly, the underlying system architectures could be quite diverse. In many cases, existing tools must be ported and validated against these new systems. This task is made much more difficult if the new systems use novel architectural features, or non-compliant or proprietary software.

Access to large scale platforms to test and evaluate tools. Access to system testbeds for software development and testing continues to be a challenge for development of software tools. In particular, software development tools must be able to run at production scale and in the same environments as production users. In some cases, these developers must be able to modify the system software, such as the operating system, to perform their tests.

Focus on software engineering. In order to bring software tools to broader audiences their usability must extend beyond the tool authors or similar tool experts with intimate knowledge of the software. Attention to software design, robust deployment, documentation, and user support will allow tools to extend their impact from being simple research prototypes to resources that enjoy wide usage by science teams. In order to move in that direction attention to software engineering as exemplified by private sector software creators would be beneficial.

Build systems to support portability of tools. Hardware and software systems that address portability will allow tools to leverage through re-use the investments already made in tool design. Hardware systems that adopt open standards to expose important performance and correctness data will see more rapid implementation of tools than in cases where such interfaces are hidden or accessible only to the hardware vendors. Likewise software efforts that build platform independent APIs for accessing such data, such as the Performance API (PAPI),

make tool development significantly easier and promote interoperability of tools through common points of reference. Lastly, portable tools have a substantial advantage since users are able to find familiar tools and methods between machines as opposed to climbing a new learning curve for each new architecture.

Transitioning research prototypes to production tools. Many of today's successful research tools could benefit from sustained funding to transition the tools to production software. The pathway from research prototype to a software tool that is widely available, production quality and actively supported is not clear. In most cases, the funding researchers receive is targeted toward specific research goals, and not necessarily to provide tool porting, testing, documentation, standardization, or user support. In addition, tools may have to be integrated into the larger software stack as provided by the vendor or facility. This integration takes planning and effort.

Supported infrastructure for tool maintenance and hardening. Similarly, academic and laboratory performance tools researchers and developers rarely possess either the skills or the desire to transition research ideas to production code, with concomitant support. However, the government rarely funds long-term maintenance and tools support. A new model of software tool support is needed if we are to address current and future needs.

5.4 Fault Tolerance and Reliability in Tools

The systems and applications communities must articulate a prioritized taxonomy of failure modes in order for tools to be prepared to address them. Then a cross-layer collaboration will be needed to define data and interfaces that must be gathered and communicated in order to make the tools themselves resilient, and provide feedback to users and developers on the fault-tolerant performance of their applications. The same requirements will hold with regards to power concerns, as the tools community will need to work closely with the entire system stack to correctly identify *what* needs to be monitored and *how* it should be measured.

Success will rely upon a strategy of "layered interoperability" based upon a shared tools infrastructure. The impact of changes to the tool infrastructure on the ability to do science must be carefully monitored through close collaborations with the developer and user communities. This collaboration will be crucial to make certain that available tool resources are focused on solving the right problems in the exascale timeframe.

5.5 Intellectual Property

IP is a major challenge for software design and interoperability, and it is not simply confined to HPC. Two approaches can help mitigate collaborative issues stemming from IP issues.

First, aside from standardization of interfaces to target system architecture and system software components, developers of software tools could benefit from standardization on specific infrastructure components within their own community. For example, APIs, tracefile formats, and user interfaces could all benefit from standardization. This standardization would promote tool interoperability among other benefits.

Second, open-source software gives the labs the opportunity to fix bugs and add features through internal efforts and external contracts. Communities can develop around open-source software resulting in cost sharing; however, we must recognize that the community of leadership class facilities is, by definition, small. Also, we must find ways to ensure open-source research projects evolve into

robust, easy to use, well-documented software. Whether we chose open-source software or proprietary software, the community must be engaged in software development tools and their underlying infrastructure.

5.6 Application Engagement

All too often, performance tools are developed in the absence of detailed understanding of user and application challenges. Conversely, users are often unaware of the technical difficulties underlying tool design and support. Bridging this gap with a collaborative tool development and extension process, where promising ideas are identified and tested early, then enhanced and supported across the application development and support cycle, would ameliorate the expectations gap. Recent experiences in both the Office of Science and NNSA affirm the distinct advantages of having computer science experts engaged with applications and domain experts. Working together, these two groups can best map the applications to the architectures.

A supporting path forward is to examine factors that impede the use of tools in the application and user communities. Reporting from the annual NERSC user survey and from wide-spread discussion in the HPC community makes clear that ease-of-use is key to the adoption of tools by users. A successful program for exascale tool use must especially focus on readily deployable turnkey tools as the complexity of likely exascale HPC architectures will bring risks that an already difficult to use environement makes tools harder to use.

5.6.1 Education and Outreach

Another necessary step in the exascale tools path forward is a means to develop two-way communication between providers about what is available and input from users regarding what works and what does not. Continued tutorials and trainings programs at venues where application scientists gather should prove a valuable asset as we move toward exascale.

Outreach also serve to broaden the scope of what we consider to be the HPC exascale community. Sharing performance and correctness tool expertise with the private sector and finding application of these tools in new venues outside DOE computing centers serves multiple goals. Firstly it increases the impact of these tools. Secondly it serves to extend the longevity of tools software as the number of stakeholders in adopted tools increases. An example of the latter are the performance tools that now come as part of a standard Linux distribution (valgrind, pin, perf_events, etc.) that were once the province of a much smaller community.

5.7 Strategies for Validation and Metrics

Power and reliability are becoming first class concerns in the exascale timeframe and are creating unique challenges for tool validation. In order to validate, one must be able to measure. In order to measure, one needs a metric. Currently, there are no universally agreed upon metrics or "benchmarks" for reliability. There is some consensus in the community that "time to a correct solution" and "joules to correct solution" are valuable metrics, but these metrics are highly application dependent.

A good place for the community to begin would be to agree upon a handful of metrics for power and reliability, define a benchmark suite of applications to be measured and implement methodologies for measurement. With this as a foundation, tools will of necessity be validated based on their ability to accurately measure power and reliability metrics on benchmark codes as compared to implemented measurement methodologies. For power, work has already been done by the vendors which could form the basis of a test suite for tool validation (e.g., the HP

Intelligent Power Discovery Infrastructure or energy management through the Dell OpenManage suite), however, in the case of reliability there is much foundational work to be done to define what is to be measured and how to measure it.

6 Bibliography

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22:685–701, April 2010.

[2] D. L. Brown, P. Messina, D. Keyes, J. Morrison, R. Lucas, J. Shalf, P. Beckman, R. Brightwell, A. Geist, J. Vetter, B. L. C. E. Lusk, J. Bell, M. S. Shephard, M. Anitescu, D. Estep, B. Hendrickson, A. Pinar, and M. A. Heroux. "Scientific Grand Challenges: Crosscutting Technologies for Computing at the Exascale". http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Crosscutting_grand_challenges.pdf, February 2–4, 2010, Washington, D.C.

[3] H. Brunst, D. Kranzlmüller, M. S. Muller, and W. E. Nagel. Tools for scalable parallel program analysis: Vampir NG, MARMOT, and DeWiz. *Int. J. Comput. Sci. Eng.*, 4:149–161, July 2009.

[4] B. Buck and J. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[5] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[6] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[7] J. Caubet et al. A dynamic tracing mechanism for performance analysis of OpenMP applications. In *Proc. of the Intl. Workshop on OpenMP Appl. and Tools*, pages 53–67, London, UK, 2001. Springer-Verlag.

[8] CBTF Project Team. Building a Community Infrastructure for Scalable On-Line Performance Analysis Tools Around Open—SpeedShop. http://ft.ornl.gov/doku/cbtfw/start/, May 2011.

[9] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 298–309, New York, NY, USA, 1998. ACM.

[10] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of spmd codes using expectations. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 13–22, New York, NY, USA, 2007. ACM.

[11] D. Cortesi, J. Fier, J. Wilson, and J. Boney. Origin 2000 and Onyx2 performance tuning and optimization guide. Technical Report 007-3430-003, Silicon Graphics, Inc., 2001.

[12] B. Dally. Power and programmability: The challenges of exascale computing. Presentation at ASCR Exascale Research PI Meeting, October 12 2011.

[13] L. DeRose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of IEEE/ACM Supercomputing '02*, Nov. 2002.

[14] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[15] N. Froyd, J. Mellor-Crummey, and R. Fowler. Efficient call-stack profiling of unmodified, optimized code. In *Proceedings of the 19th ACM International Conference on Supercomputing*, Cambridge, MA, 2002.

[16] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[17] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-driven software race detection using hardware performance counters. In *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 165–176, New York, NY, USA, 2011. ACM.

[18] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.

[19] K. A. Huck and A. D. Malony. PerfExplorer: A performance data mining framework for large-scale parallel computing. In *Supercomputing 2005 (SC'05)*, page 41, Seattle, WA, November 12-18 2005.

[20] 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.

[21] G. Mercier and E. Jeannot. Improving mpi applications performance on multicore clusters with rank reordering. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 39–49, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface Standard*, 1997.

[23] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, 1999.

[24] B. Mohr et al. A performance monitoring interface for OpenMP. In *Proceedings of the Fourth European Workshop on OpenMP*, Rome, Italy, 2002.

[25] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2001. CD-ROM.

[26] S. Moore et al. *A Scalable Approach to MPI Application Performance Analysis*, volume 3666 of *Lecture Notes in Computer Science*, pages 309–316. Springer-Verlag, 2005.

[27] P. J. Mucci. PapiEx - execute arbitrary application and measure hardware performance counters with PAPI. `http://icl.cs.utk.edu/m̃ucci/papiex/ papiex.html`, 2007.

[28] 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.

[29] N. Nethercote. *Dynamic Binary Analysis and Instrumentation.* PhD thesis, University of Cambridge, 2004.

[30] Open Solaris Forum. Man solaris - getitimer (2). http://www.opensolarisforum.org/man/man2/getitimer.html, June 6, 2001.

[31] Oracle. Oracle Solaris Studio Performance Tools. Oracle White Paper, http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-performance-tools-183986.pdf, Nov. 2010.

[32] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualize and analyze parallel code. In P. Nixon, editor, *Proceedings of WoTUG-18: Transputer and occam Developments*, pages 17–31, mar 1995.

[33] R. Preissl, N. Wichmann, B. Long, J. Shalf, S. Ethier, and A. Koniges. Multithreaded global address space communication techniques for gyrokinetic fusion applications on ultra-scale platforms. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 78:1–78:11, New York, NY, USA, 2011. ACM.

[34] P. Roth, D. Arnold, and B. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of IEEE/ACM Supercomputing '03*, Nov. 2003.

[35] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[36] M. Schulz and B. R. de Supinski. PNMPI tools: A whole lot greater than the sum of their parts. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Reno, Nevada, Nov. 2007.

[37] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. Open|speedshop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 16(2-3):105–121, 2008.

[38] M. Schulz, J. A. Levine, P.-T. Bremer, T. Gamblin, and V. Pascucci. Interpreting performance data across intuitive domains. In *International Conference on Parallel Processing (ICPP)*, Sept. 2011.

[39] Silicon Graphics, Inc. (SGI). SpeedShop User's Guide. Technical Report 007-3311-011, SGI, 2003.

[40] D. Skinner. Performance monitoring of parallel scientific applications. Technical Report LBNL-5503, Lawrence Berkeley National Laboratory, 2005.

[41] R. W. Software. Acumem ThreadSpotter Manual, version 2011.1, April 2011.

[42] R. Stevens, A. White, P. Beckman, R. Bair, J. Hack, J. Nichols, A. Geist, H. Simon, K. Yelick, J. Shalf, S. Ashby, M. Khaleel, M. McCoy, M. Seager, B. Gorda, J. Morrison, C. Wampler, J. Peery, S. Dosanjh, J. Ang, J. Davenport, T. Schlagel, F. Johnson, and P. Messina. A decadal DOE plan for providing exascale applications and technologies for DOE mission needs.

Presentation at DOE Advanced Scientific Computing Advisory Committee (ASCAC) Meeting. March 30, 2010. http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/mar10/Awhite.pdf.

[43] H.-H. Su et al. GASP! a standardized performance analysis tool interface for global address space programming models. Technical Report LBNL-61659, Lawrence Berkeley National Laboratory, 2006.

[44] N. R. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 229–240, New York, NY, USA, 2009. ACM.

[45] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. Scalable fine-grained call path tracing. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 63–74, New York, NY, USA, 2011. ACM.

[46] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel. Diagnosing performance bottlenecks in emerging petascale applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 51:1–51:11, New York, NY, USA, 2009. ACM.

[47] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 269–280, New York, NY, USA, 2010. ACM.

[48] J. Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *International Conference on Supercomputing*, pages 245–254, 2000.

[49] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 240–250, NY, NY, USA, 2002. ACM Press.

[50] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proc. of the* 8^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, 2001.

[51] J. Vetter (Ed.). Software development tools for petascale computing workshop report. http://science.energy.gov/~/media/ascr/pdf/workshops-conferences/docs/Sdtpc_workshop_report.pdf, Aug. 2007.

[52] F. Wolf and B. Mohr. EPILOG binary trace-data format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Julich, May 2004.

[53] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient pattern search in large traces through successive refinement. In *Proc. of the European Conference on Parallel Computing*, Pisa, Italy, Aug. 2004.

[54] F. Wolf, B. Wylie, E. Abraham, D. Becker, W. Frings, K. Fuerlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, and Z. Szebenyi. Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications. In *Proceedings of the 2nd HLRS Parallel Tools Workshop*, Stuttgart, Germany, july 2008.

[55] P. H. Worley. MPICL: a port of the PICL tracing logic to MPI. http://www.epm.ornl.gov/picl, 1999.

[56] C. E. Wu et al. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 2000. IEEE Computer Society.

[57] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.