2015 Runtime Systems Workshop: Summary Report

March 11-13, 2015
Rockville Hilton, Rockville, MD

Editors

Sonia R. Sachs, DOE ASCR
William Harrod, DOE ASCR
Thuc Hoang, DOE NNSA

Report Contributors

Pete Beckman, Argonne National Laboratory
Andrew Chien, UChicago
Josh Fryman, Intel
Laxmikant Kale, UIUC
James Laros, Sandia National Laboratory
Pat McCormick, Los Alamos National Laboratory
Dave Montoya, Los Alamos National Laboratory
Rob Neely, Lawrence Livermore National Laboratory
Thomas Sterling, Indiana University
Kathy Yelick, Berkeley National Laboratory

Background

Proposed exascale computing architectures present scientists with a number of challenges to reaching DOE scientific goals. Future runtime system software must achieve significant improvements in efficiency and scalability in the context of user productivity, performance portability, and dynamic adaptation.

In particular:

- Computing platforms must become significantly more responsive to power constraints, faults, and new goal-based programming models. Current system software is often very static in nature – computing jobs are given fixed numbers of compute resources at the beginning of every job, power is not dynamically adjusted to meet computational goals, and parallelism is often fixed.
- Runtime systems must support new, highly dynamic task-based programming environments that span computing resources inside a node as well as globally across the platform.
- New software frameworks supporting introspection, autonomic tuning, programing tools to support debugging and performance adaptation need

runtime layers that can efficiently manage hierarchical memory, heterogeneous computing elements, and shared storage systems.

Advanced runtime systems must also be portable, have stable interfaces that can support the long development cycles of many computational science teams, and perform well across a variety of machines from different vendors or generations.

To address the research challenges outlined above, this workshop convened approximately 45 domain experts in High Performance Computing Runtime Systems (RTS) together for 2.5 days with the following high level objectives:

1. Propose, discuss, and determine the required characteristics of future extreme scale runtime systems
2. Devise metrics, measurements, benchmarks, and other means for testing and evaluation for prototypes of runtime systems,
3. Identify research questions that need to be resolved within the context of current experience and knowledge,
4. Discuss a research and development roadmap that will result in one or more high quality runtime system software packages that could be deployed in the 2023 timeframe, on extreme scale systems.

The workshop was a combination of invited speakers and open breakout sessions for directed discussion. The sections in the reports that follow capture primarily the discussions of those breakout sessions, which in turn covered the following topics:

- The architecture for future RTS software
- Runtime systems design
- Outstanding research questions
- A roadmap for the future

The participants of the workshop laid out current strategies for designing advanced runtime systems, and then projected forward to the requirements, needs, and architectures required to meet the needs of exascale computing. Currently, there is no ASCR Program focusing specifically on runtime systems, but instead they are a part of the X-stack program, the Exascale OS/R program, and several other projects. The NNSA ASC Program also supports RTS research and development in the context of advanced computing and programmatic needs for advanced applications. The current co-design centers have provided insights for the needs for future RTS software, but the co-design centers are currently not designed to support tight interactions with system software. Future configurations for new co-design centers could benefit from a trio of top-level participants: the Application, Platform Vendors, and System Software.

Spurring renewed interest in this area is the recent announcement of the ASC Program's Advanced Technology and Development Mitigation (ATDM) program element, which charters the ASC labs with developing the first "from scratch" set of applications in support of the weapons program since the start of ASCI in the mid 1990's. Several ASC teams are actively exploring using next-generation programming models (beyond

MPI+X), and work is starting immediately to get those off the ground. This revitalized application effort, along with the pre-exascale procurements in the CORAL (2017) and APEX (2020) timeframes led to a sense of urgency at the workshop for hardened research solutions.

Key Takeaways

Some of the high level issues raised and suggested next-step actions include:

Identified Issue	Suggested Action / Next Steps
A conceptual model for the multiple layers of RTS, and their role in software stack would be very helpful to focus roadmaps and requirements. A taxonomy and subsequent standard terminology would help tremendously.	• A shared architecture for the software stack should be developed, with at least several of the larger RTS components named and described in a shared taxonomy/nomenclature • Key terminology and definitions should be developed and referenced by the community as part of a new "standard model" for discussing system software. • Define which RTS layers live within a node, and which span a job, and which are global to the system?
The ecosystem for RTS components was not articulated. Specifically, which are: written by the vendor, vendor productized open source, community developed and supported. Sharing, reuse, and APIs for components/services follows the ecosystem model	• Working with the vendors and the computing facilities, the ecosystem model for developing or improving new components should be developed. • Identify which RTS services will be "stand alone", and reused by several components (for example a data movement library) and which RTS services are expected to be deeply embedded into larger components. For example, RTS support for specific language features. • As components are adopted, standard APIs much be identified.
Metrics are poorly understood. Latency and bandwidth are helpful for communication layers, but what about task-based layers? Introspection? Dynamic power, etc.?	• Beginning with the components and taxonomy described above, we must articulate the requirements for those layers and the metrics, benchmarks, and measures for evaluating them.
There is a natural tension between RTS layers and dynamic control. What does each RTS layer or component control, and how do layers coordinate toward shared, goal-oriented optimizations?	• As we unroll the RTS components and layers, we must identify the resources that are managed, and how layers coordinate and optimize. • What does a shared backplane for communicating between runtime layers look like? We must describe the requirements for

	such a backplane. • To support introspection that can be used hierarchically (node, job, machine), we must articulate the data that must be collected and shared.
Adoption of new RTS layers must be done in concert with application teams and other system software developers	• DOE must work harder to partner application teams with scientists developing new RTS layers in a co-design process. Co-design should include system software, applications, and platforms.
Runtime systems needn't only support resilience, but be resilient themselves	• Resilience services, but also design and implementation a exascale runtime services is a critical challenge •
The impact of RTS research needs to be broadly disseminated	• We need to catalog the open research questions, track those that we're learning the answers to, and work to share peer-reviewed success stories with the broader community

In addition to the table above, several of the session authors below elucidate their own research priorities in more detail. Their summaries are included below.

Session I Architecture for Exascale RTS

Chair: James Laros, Sandia

Summary

Runtime systems represent a broad class of software within the software stack. The group explored a wide range of architectural issues for designing, building, and interfacing to advanced runtime systems. The group used the guiding questions provided to the attendees as a backdrop for the discussions. They also agreed at the start of the session to avoid getting bogged down in terminology. However, the group noted that a better shared understanding on concepts and terms may be a worthwhile exercise for the community. In the summary below, each of the topic areas is presented, followed by a summary of the discussion.

Execution Model

We first discussed the domain of the runtime system, multi-node versus single node for example. This quickly led into a discussion of what can be expected or should be expected of the runtime system from other layers that comprise the execution environment. It quickly became clear to me that the answers to these questions would not have concrete answers or consensus among the group and often depended on the *service* the runtime was providing. We used the word *runtime service* consistently throughout the remainder of the discussion.

We noted that runtime services should be able to be bypassed. While there was general agreement on this topic there were exceptions noted where a service has a more global responsibility beyond, for example, a single application. System services like resiliency might fall into this category of being hard to bypass if doing so would affect other applications or services but was also one of the first examples of a runtime service that an upper layer might choose to bypass (possibly for performance reasons). It was agreed that if you choose to subvert a service you couldn't count on the benefits it might provide. This of course does not address the previous concern.

The challenge of getting these layers to work together or be aware of each other vertically or horizontally was noted and discussed with no resolution other than identifying this as a difficult task. This was a repeatedly expressed need. See later in the document where the sharing of semantic information is important.

Asynchrony

We almost immediately reframed this question as Performance Variability. We noted that today we view our world as regular, increasingly not true due to hardware variability, contention, algorithm irregularity, and other issues. Question, how do we do resource management when access to resources is stochastic?

It was noted that asynchronous runtime systems, or programming models, could be used to address or tolerate variability of communication (network or memory access for example). It was also noted that these asynchronous methods could also introduce other types of variability. Performance variability, whether cased by system variability or variability introduced by other means might be handled by load balancing. Finding the balance of how much coordination is appropriate was identified as a research question. We need to know much more before understanding the opportunities for using hardware

to solve these issues. Things like fast atomics and lightweight task scheduling may be candidates. The behavior of software on the hardware or the equivalent of global performance counters could be very beneficial.

System Fragmentation

The glib answer to this topic was yes. Using the term service, it was recognized that it depends on the service. If the service has a responsibility that spans the entire system then it would have a global span. Likewise if the service or utility provided by the service were application centric it would only span the resource used by that application. It was additionally postulated that a service might only have a single node perspective.

Relationship between OS and runtime system

Some of the key points that came out of this portion of the discussion were also framed in terms of runtime services. We discussed the distinction of services used by an application and services that are used between applications. This conversation while focused on the relationship between the OS and runtime had parallels to many of the other topical discussions.

Part of the discussion that more specifically addressed the OS/runtime relationship was the statement that the OS should act as a hardware abstraction and as much as possible be able to get out of the way, or out of the critical path. My personal perspective on this was this desire paralleled that to the motivations that produced early lightweight kernel operating systems. This conversation overlapped the following topic. It seemed that the tendency of the group was to consider this vertical path from the OS to the programming model in deciding how things could or should be abstracted.

Relationship between Programing Models and Runtime Systems

What is exposed to and what is hidden from the programming environment. Also at what layer does the hiding, if things are hidden, occur. Continued conversation identified many layers here and the fact that much research was needed in this area. The application, compiler, runtime and OS and what is presented and exposed at each layer. Continued discussion identified that the balance of performance, portability, productivity and understandability are critical to these choices.

The group discussed and recognized, without conclusion or further direction, the need to capture the connection to other types of services like data management, security and performance monitoring when considering the relationship between the runtime and programming model (or by extension the application and compiler). Many of these do not fall under our definition of a runtime system.

The group also discussed and supported comments made earlier in the day in Josh Fryman's presentation (I believe) regarding passing semantic information between both runtime services and other layers. The group agreed with the importance of finding a balance here and the potential impact to performance, scalability etc. that these considerations could impact. It was agreed that this information is critical to expose to the programming model so the platform could be properly abstracted. It was observed during this conversation that we would need more influence or control over compilers and JIT compilers could also be considered a service and some went as far to say that the compiler should be considered an integral part of the runtime system. This transitioned well into the next topic.

Compile-time information, guidance, and constraints

We also changed how this topic was worded for our discussion to – Compile derived information. The primary take home from this portion of the discussion is that a large amount of research is needed in this area. We do not currently have enough impact in this area -- was a generally accepted theme. The need to have a two-way street of information through layers or services was again stressed. The term workflow became more actively used as we went up the stack in our discussions. This and the previous topic were discussed more as part of a whole so I won't repeat the observations I already listed in the previous topic section.

Evaluation

We framed this question in a number of ways including asking ourselves these questions. Who defines what is good? Should we define what success is on future systems, which might be very different from success on current systems, by improvements in workflow? What are performance metrics and how do we represent them. Should there be an abstraction of performance metrics? Too many performance metrics again produces variability (see previous topics). The question of what is missing was also raised. The value of relative metrics was also discussed. Appropriate baselines to show the effect of future decisions are important. Relative metrics are also important in evaluating research prototypes especially when evaluating capabilities that did not previously exist. We also need to evaluate on factors that were also not previously considered important like power and energy.

We discussed the need for the system/platform to present the information we need. Not currently the case. This relates to a previous topic, what do we need from the hardware. Even if the system provides this information, how do we collect it? How do we collect it efficiently? A short list of what we want includes; time to solution, time to solution with failures, time to solution with system variability, time to solution under power/energy constraints, runtime overhead (CPU, Time, Space – memory). We want to evaluate the benefit provided versus the overhead introduced. Many of us have heard that there may be an acceptable loss in performance given these additional values.
The portability of the runtime system was also determined to be important. Algorithms and data structures should be portable across multiple architectures. Runtime should provide program portability.

Session II Architecture for Exascale RTS

Chair: Josh Fryman, Intel

Summary

In the exploration of future runtimes and execution models, a fundamental question is whether there can be a distinction between a *runtime* (distributed or not) and an underlying *operating system*. The evolution of features in both of these systems-software areas has created a blurred boundary, confounded with terms indicating "hierarchy" properties (such as middleware) that do not have a standard convention or abstraction. Collectively, it undesirable to have imprecise vocabulary, but the imposition of a rigid hierarchy such as the standard network layers will require community-wide effort. Similarly, the dividing line between a programming model and the underlying runtime is also blurred by techniques such as dynamic compilation, interpretive languages, and complex hosted environments, with Emacs itself being a program which defies classification.

Without a defined set of behaviors, it becomes hard to classify boundaries between layers. These boundaries may be required to support a measurable level of interoperability between components, as well as composability. Such boundaries not only define a clean set of abstractions for basic functionality, they also are the interfaces that will be used for standard management of access permissions, isolation, fragmentation, etc. How each of the major and minor components interact with the underlying platform – from basic kernel hooks of local hardware resources to filesystem support – will require some form of standardization for the interoperability and composability issues to be tractable problems. The evolution of these interfaces should be matched to the integration of other supporting infrastructure: in-situ workflow management, tools and analyzer frameworks, etc.

The lack of a clear taxonomy with well-defined characteristics and vocabulary will continue to hinder effective integration and sharing of efforts across projects. This will also limit the effectiveness of mapping different programming models, execution models, and runtimes across various criteria to determine optimal strategies for a given problem domain or algorithmic approach.

While such boundaries are lacking, an alternate issue to consider is what success should be characterized by for future programming models, execution models, and runtimes. Once success can be defined as a measurable outcome, differences between these three inter-related components may suggest a natural division between what a specific use-case requires and how it is best paired to a given runtime. This measurement of success needs to account for tolerance of variance within extreme-scale systems (frequencies, resiliency, power, etc.) as well as overall performance criteria (throughput, wall-clock time, cost per job, etc.).

A further requirement for the success criteria to be defined is to address the balance between basic features and over-arching goals for an end-to-end language-to-runtime solution. Since "success" to the programmer and the hardware vendor will appear very different at a semantic level, it is important to establish the requirements for each module in the hierarchy from high-level-language (HLL) to the actual hardware. These requirements will impact the assumptions and design models for each component, and in turn will reveal the type of information each component needs to receive or pass

through in order to be successful at evaluation. Fundamentally, the programming model should be exposing parallelism, while the programming environment and underlying execution components should be utilizing that parallelism. The simplest form of success is the quantification of the efficiency in each component to capture and realize the parallelism potential of a specific problem.

The loss of semantic knowledge from HLL to eventual application binary image is a fundamental problem. Future dynamic systems will need more knowledge about data structures, communication patterns, and computation requirements regardless of the underlying execution model. This information will enable better support for system management of performance, throughput, and resiliency. Current tools and infrastructure are a “lossy” push-down of knowledge from the HLL to the application binary, with very little return channel information outside of profile-guided optimization tools and very slow binary instrumentation libraries.

When the original application is created in a HLL that may be a domain specific language (DSL), understanding why a problem in performance or correctness arises when there is a dearth of semantic information in both directions becomes akin to the problem of debugging a heavily optimized binary without symbolic information. This gap of semantic exchange and metadata annotations from the HLL to the binary image, and in reverse, must be addressed. In order to address this gap, however, the nature of the information to exchange needs to be identified. The combination of defined modules in the software stack with well-defined metadata exchange interfaces will reduce the “finger pointing” phenomenon when errors inevitable arise in a given module. It will reduce the search space, as well as provide a reproducible test case to demonstrate the underlying problem(s).

Using the requirements analysis of each major component in the software stack, for example, will indicate the basic types of metadata that are needed for success. By ensuring that each component is capable of passing information required by higher and lower components in the overall stack, the basic interfaces can be exposed – which facilitates interoperability and composability. The gap that remains is to standardize a metadata representation formation, as well as the specific mechanisms for representing different types of metadata. From source-to-source translators and DSL compilers to low-level compilers or optimizers, attributes pertaining to data (streaming, random, dense, sparse, etc.) and compute (integer, real, bit-string, etc.) and communications (reduction, multicast, barrier, etc.) are only the low-hanging fruit to be used. Further analysis about the nature of a data structure’s relation to a computational kernel (stencil, halo exchange, etc.) would allow for better optimization and utilization of the program on a given machine, as well as providing better insights to the application developer for how certain constructs have mapped to that machine.

Additional guidance or metadata pertaining to boundaries for a specific run (QoS requirements, overall allocation of power, compute, memory, or bandwidth, etc.) should be exposed as hints from the application programmer, yet allow for run-time (command-line) over-rides or precedence settings. These would in turn drive policy selection scenarios at the runtime and machine behavior level, up to and including throttle feedback to the source program analysis that resource starvation was observed due to limits.

Such limitation would also be necessary to limit thrash or poor interactions when multiple agents are attempting to utilize the same resources, such as different applications and runtimes sharing a common pool of compute agents with overlapping resource allocation (over-subscription of physical resources). This problem of poor interactions comes to a core issue of who determines the granularity of resources and jobs (or job compute tasks) that the underlying software components operate upon, and how that granularity feeds back into the completely execution environment. Given that future machines are likely to be elastic, with resources appearing and disappearing as components fail and are replaced during a live operation, the software components must also be capable of dealing with elasticity in instantaneous views of overall system capabilities.

An over-arching problem is that resiliency is typically considered in isolation and after a programming model and/or execution model are extant. In reality, the resiliency model carries fundamental constraints on both the programming model abstractions and primitives, as well as the manner in which the execution model components (runtime, kernel) will interact. The architecture of a given resiliency approach should be established concurrently, if not in advance, of a given programming or execution model. By making resiliency and the requirements for resiliency a first-class citizen in the design and evaluation of future components, the ideal outcome is to avoid the need for disruptive retrofitting and breaking abstraction boundaries of the fundamental components in the total software solution.

Ultimately, the problems remain of what are the “right” things to measure in the runtime and execution layers – what should be measured, what is a metric for measurement, what requirements do those modules carry, and how does the metric assess the optimality of an approach to that module? In a related fashion, how can evaluation of prototype platforms be fairly compared to production environments that enjoy decades of hardened and optimized implementation efforts? It will remain a research problem of how to disambiguate the merit of a design or approach from the quality of a specific instantiation.

Session III and IV – Runtime Systems Design

Chairs: Laxmikant Kale, UIUC; Pat McCormick, LANL

Summary

The design of effective and efficient runtime layers within the exascale software stack faces numerous issues. These challenges are rapidly evolving due to diverse processor-, node- and system-level architecture designs and the need to address the growing complexity of programming and the overall scientific workflow. Specific issues include the need to manage increasingly complex memory hierarchies and types of memories, support flexible data locality and processor affinity, supporting asynchronous computation and data movement, and various issues related to resilience and energy efficiency. These complexities must all be addressed in such a way that it will be possible to effectively and flexibly leverage future hardware designs and also provide multiple levels of introspection for the complex interactions between the hardware and the associated software layers. This need extends well beyond the specifics and design choices of hardware architectures and into the semantics of the supporting software layers; including the higher-level programming models presented to application developers and the lower-level runtime systems that are responsible for supporting them.

The Memory System

The necessity of addressing the complexity of the memory hierarchy is initially motivated by the impact of chip-level architectural design decisions (e.g. domains of relaxed coherence, stacked high-bandwidth memories coupled with slower off-chip memories, etc.) and the corresponding details of appropriate algorithmic selection and the corresponding implementation details. At larger scales, locality concerns remain complex with additional levels of non-uniform memory access costs, the introduction of non-volatile memories, the potential use of processor-in-memory technologies, distributed address spaces and the overall relationship with supporting large-scale, persistent storage systems. Furthermore, dynamic application, software and system hardware behaviors combine in ways that are extremely difficult to fully reason about in a purely static manner. These characteristics will likely all be on aspects of the critical path to achieving high-performance. The requirements, design and implementation details of supporting runtime systems are therefore critical in helping both programming system and application developers address these concerns.

The impact of memory system design decisions on algorithmic selection and implementation clearly point towards the necessity to provide adequate controls for both the placement and layout the data. Concepts such as naming (in place of memory addresses) and higher-level data models must be exposed or abstracted in such a manner to allow application developers to reason about application behaviors and corresponding performance characteristics. Furthermore, the runtime must be flexible in supporting a range of static, semi-dynamic/static, and dynamic uses of the memory system to match the needs across a wide range of applications. When coupled with emerging trends in storage system architectures, it is critical that aspects such as persistence and composition across potentially differing namespaces be considered as part of the design aspects of a runtime system specification and corresponding implementation. Open questions remain about the separation of the memory and

storage systems. These aspects of data movement and the representation within the runtime, and across the various levels of the memory hierarchy and storage system, have consequences in supporting higher-level programming constructs as well as aspects of supporting an efficient scientific workflow.

There are several considerations and challenges about the interfaces between runtime software layers and the underlying operating system. In addition to determining what capabilities belong at each of these levels, the memory hierarchy presents some specific questions. In particular, there are interesting choices to be made throughout the software stack. In particular, there are interesting choices to be made as to how the levels of the memory hierarchy are virtualized and if they are best managed by the general policies of the operating system or instead by the runtime layers where more explicit controls can be adjusted/customized to meet the needs of application-driven performance optimizations. In the later case, it is important that details be communicated in a manner that composition and dynamic decisions can be effectively and efficiently communicated throughout the software stack. It will be critical to explore how these issues can be presented in an abstract manner that allows for the composition of different programming models/systems, supports platform portability and yet also allows developers to effectively reason about and optimize the performance of their applications.

Introspection

Given the increasing complexity at exascale, it is critical that application developers can understand details relevant to the performance and nuances of both software and hardware. Even in the case of very regular applications issues such as system-wide power capping and job scheduling nuances can introduce performance impacts that can be difficult to reason about. Such aspects, and how they are communicated to the higher-levels of the software stack, should clearly be considered in the design of runtime systems. This also requires the consideration of a well defined set of policies and abstractions that allow reasoning about the behaviors of the system in a meaningful way without burdening the developer with complex, hardware-centric, low-level details. Key questions remain in terms of defining these key abstractions and how they are aggregated in applications that utilize multiple higher-level programming models and supporting runtimes. There is a clear benefit from additional activities that look to identify and potentially standardize a common API (or data description) for introspection across runtimes and hardware interfaces.

Support for adaptivity within the runtime system requires dynamic monitoring of the overall behaviors of the application and hardware. Application observables may include computational loads of individual work units, communication patterns among the logical entities (as opposed to those between processors), memory footprints of individual logical units, etc. Hardware observables may include such metrics as core temperatures, power consumption, etc. The information needs to collected and presented at multiple levels of granularity, to enable appropriate runtime adaptation strategies. In an ideal situation the cost of gathering introspection data should have little to no impact and support should range from gathering data from hardware resources that range from individual chips, the full system and finally out to the details of the supporting computing facility (e.g. power monitoring). In cases where the introspection data is utilized by the runtime system itself the costs of data collection should only be done where adaptive techniques are desired. This suggests providing developers with the ability to turn

introspection and adaptive behaviors on and off to support a cost-benefit analysis about both application and hardware behaviors and the impact the runtime system has upon them.

It must be recognized that even when rich and detailed local information is inexpensive to collect, the composition of these results across all the nodes of a job is problematic. Aggregation techniques, as well as methods for understanding the temporal nature of the data, are necessary. Therefore, techniques and introspection databases must meld local information that is detailed and up-to-date, with global information that is aggregated and somewhat stale. Furthermore, when such information is presented to the developer (at any level of the software stack) there is a potential to be overwhelmed by significant amounts of data. As a result, introspection introduces an additional data analysis and visualization problem to aspects of the entire workflow.

Reliability

The lowest levels of resilience mechanisms must be enabled at the level of the operating system. To fully support this capability it is critical that hardware resources allow their state to be determined and evaluated. From this point there are many implications for runtime systems. The runtime must support the need to store persistence/recovery (potentially versioned) data at different locations within the memory hierarchy (including parallel file systems). Furthermore, applications must be able to be assured that any lower-level implementation is guaranteed/recoverable and have enough information to be controlled (if so desired), understood and reasoned about in terms of the associated costs and impact on the application. The attribution of errors should allow the software stack to identify problematic hardware resources and respond accordingly. It must be possible for faults to be considered across the entire software stack and raised to the location best suited to address the details of recovery.

Further consideration and studies need to fuller explore the vulnerabilities of both applications and runtime systems themselves to faults. While tasking-based models provide finer grained units of recovery, they do present a challenge to having to manage more complex state recovery if they were subject to the fault. Furthermore, the runtime should allow the application level code to help determine faults (e.g. soft errors) and drive associated recovery mechanisms as needed.

Given the complexity of interactions between application codes, libraries, runtime systems, the operating system and hardware, the implementation of fault recovery from any given error might be overly expensive and, sometimes, it might simply be better to let the application fail. Further studies need to occur to better understand the balance between coverage, functionality, the impact on the overall software stack (including applications) and the eventual likelihood of faults at exascale.

Energy/Power Management

Power management involves considering four distinct metrics: power, energy, temperature and execution time. In general, the power level and core temperatures provide constraints, while the others are part of the objective function that is being optimized. From this perspective, three different software aspects of runtime systems are responsible for power management: the job scheduler, one or more job-level runtime

systems, and a node-aware set of software layers. Each of these levels can potentially be responsible for managing some aspect of their control over the overall power bounds. However, an open question remains about how much control applications need over power/energy details. It is currently the case that many such details are currently hardware-specific and significant work would be needed to generalize this information and make it useful at the application level across a range of different architectures.

The overall benefits of detailed application-level control (or hints) over power/energy remains unclear and further study is needed. Lower-level interfaces and control are potentially easier entry points but many details and studies will need to be better understood. Such efforts should consider how best to portray the cross product of the metrics, appropriate cost models and how they are impacted by various hardware design decisions (e.g. power throttling) and system-wide and facility power management requirements. Furthermore, broader efforts should be taken to explore common power APIs can be leveraged across various system designs and generations. Finally, it remains unclear if energy or execution time should be the primary criteria for the exascale era.

Scheduling & Resource Management

Scheduling directly relates to the aspects of sequencing a set of operations that are ready to execute on a given hardware resource. Separately, resource management encompasses decisions based on memories, communication and processor resources and can be considered as prioritizing different work-units based on the availability of these resources. It includes load balancing by controlling the location of work units and data units. The determination of these choices can range over a wide set of options that can once again benefit from application-level awareness. This includes decisions for reducing latencies introduced by data movement, available processor types in heterogeneous system architectures, and the power metrics discussed above. Special considerations must be given to how these choices can be expressed in ways that can be tailored to the goals achieving performance on a given system architecture; but at the same time maintaining a reasonable level of performance portability.

Additional related challenges arise from the manner job schedulers currently allocate resources. In particular, today's techniques tend to assume a rigid model where a particular job is provided a fixed/static node allocation at runtime and has little to no flexibility after a job has been launched. At exascale, there are potential benefits from allowing jobs to become malleable, so the resources allocated to them can be changed at runtime, either by the job itself or by the job scheduler (global runtime). The runtime system as well as the application will need to adapt to, or potentially help control, such malleability. .

A complex set of considerations arise when an application consists of the composition of multiple software layers and runtimes that have conflicting approaches to scheduling. Decisions made at one level are potentially in conflict or detrimental to choices at another. A careful delineation and set of design decisions are necessary to help address this concerns. While introspection abilities are clearly of benefit to addressing this concern, steps must be taken to allow application level developers to clearly understand the impact of composition. From the perspective of the runtime system, applications, compilers and hardware should provide knobs, with clearly defined

effects, while allowing the runtime to control the knobs based on the evolution of the runtime and system conditions.

Tool Infrastructure

Each of the aspects discussed above have an impact on how we consider the design and development of a set of tools to help developers understand and reason about both application and system performance and characteristics. This highlights the importance of maintaining a semantic awareness throughout the toolchain and within all levels of the runtime infrastructure and presents a much more transparent nature to the details within the runtime infrastructure than is typically the case in present designs. Furthermore, it is recommended that aspects of the tool infrastructure, including the compiler toolchain, be considered/designed in concert with the runtime and higher-level programming infrastructure(s).

Specific requirements of developer-facing tools must provide information about why, when and where various choices were made by either direct programmer involvement or by dynamic runtime level decisions. Being able to effectively support attribution of performance bottlenecks is a difficult but important aspect to consider in the interactions with tools and runtime systems. In cases where nondeterministic behaviors potentially exist, it must be possible for the developer to enforce controls (e.g. ordering) at the cost of potential performance impacts. This will be a critical feature to simplify the tasks of debugging and reasoning about correctness. To support running at scale a clear necessity will be to include the integration of data aggregation and visual representations to help quantify and understand overwhelming amounts of data and complex relationships between hardware and software in a meaningful manner.

Additional challenges stem from analyzing programs that are leveraging more than one programming system and therefore potentially leverage more than one runtime system. In this case tools must be able to reflect the model of computation presented so developers can reason in terms of the supporting higher level abstractions and not only the lower level details of the underlying software and hardware infrastructure.

Developers must also be able to understand the detailed decisions that the runtime system has made (e.g. thread affinity and data locality) in a way that they can reason about implementation and algorithmic tradeoffs in a knowledgeable but flexible and coherent manner. In this process is must be possible to support both the isolation of each system as well as to gain an understanding of the impacts of sharing between two or more runtimes and higher-level programming models. Furthermore, tools should support presenting results using multiple levels of abstraction – for example, the presentation and/or levels of detail an application developer wishes to see may significantly differ from those of a runtime system developer. Finally, in the case of dynamic system and application characteristics real-time profiling of an application can be critical in understanding the dynamic behaviors -- in this case an interactive visualization of program execution could prove to be extremely valuable in the detection of certain behaviors that would be extremely difficult via a post-processing session.

Evaluation

There are many metrics that can be used to evaluate the success of a runtime system. The simple viewpoint suggests that its usage/adoption often implies a

significant number of other aspects including scalability, flexibility, portability, completeness, ease of use and suitability for the given use case (e.g. appropriate level of abstraction). Although metrics can clearly be beneficial, the associated details can be complex and often difficult to both enumerate and measure. Attempting to use only mini/proxy applications is not necessarily meaningful due to a lack of suitable complexity to fully evaluate the system. Similarly, the implementation of a full-scale application considers one aspect of complexity but only addresses the needs of a focused area and set of methods. Although some layers of the runtime provide support for applications, they are not directly exposed to the application developer. For example, they may provide a layer of functionality needed by higher-level runtime capabilities and/or are designed as a target for compiler code generation. These layers should clearly be evaluated differently from those intended to address higher level, application facing, programming challenges.

The reality of the situation is that various details and nuances all contribute towards a successful runtime system and many different aspects need to be evaluated and considered throughout its design and development. Where appropriate it is critical that application-aware requirements and activities (even if driven by higher levels within the software stack) and multiple architectures be considered as part of the overall evaluation efforts.

Session V and VI – Runtime Systems Research Questions

Chairs: Thomas Sterling, Indiana; Andrew Chien, UChicago

Introduction

The runtime system in distinct forms has served computing systems principally in the area of virtual machines (e.g., JVM) to facilitate advanced programming and only to a small degree for HPC (e.g., OpenMP). For pathfinding system architectures to achieve general and portable exascale performance, runtime systems are proposed as a key enabling innovation to greatly improve efficiency and to achieve dramatic increases in scalability. The driving opportunity is the exploitation of real-time information about the system status and application execution to dynamically manage the system resources and task scheduling. Runtime systems are also considered essential for future methods of energy efficiency and resilience. The workshop on exascale runtime systems examined the potential promise of runtime systems in their many possible roles to enable exascale computing. These discussions included insights derived from current generation experimental runtime software developed under the DOE XSTACK program and other research initiatives as well as questions still inadequately resolved, determined necessary prior to deploying fully comprehensive and robust exascale systems.

An indirect consequence of the understanding of future runtime system software is its interoperability, interface protocols, and complementary roles and responsibilities with other system layers including programming models, compilation techniques, parallel operating systems and highly scalable computer architecture. Research is required therefore both directly in the design and implementation of the exascale runtime system and in its supportive means and methods of the rest of the exascale system and applications.

This report identifies and describes a set of research questions that must be addressed in order to guide the development, deployment, and application of HPC runtime software as part of a future system software stack for exascale computing. These critical-path research issues are presented as a set of over-arching strategic questions that establish the broad framework of a future research program and an additional, sometimes overlapping, set of detailed research questions that expose, albeit incomplete, insight into the challenges facing the potential promise and means of exploitation of exascale runtime system architecture and implementation software.

Strategic Questions

The runtime system is an important innovation in HPC distinguishing it from conventional approaches to current high end computing. The expectation that the transition to dynamic adaptive control of both resources and tasks is promising but as yet unproven although early experiences suggest strong potential at least in some cases. Research is required to determine functionality, software architecture, interoperability with other system layers, control policies, and achievable performance advantage. While there are many detailed questions to be resolved, some depending on specific approaches and assumptions, overriding strategic questions critical to the success of exascale computing and relevant to all likely possible runtime systems need to be pursued. These strategic questions are identified and briefly described below:

1) *Parallel Tasks* – what are the forms of the schedulable tasks that are managed by the runtime system, the nature of the computation they encapsulate including lighter weight parallelism, the criteria determining where and when they are executed, and conditions of preemption if allowed. These may include conventional threads, processes, atomic micro-actions, codelets, compute complexes, and other executable and independently schedulable objects.
2) *Memory Model* – from the perspective of the abstract machine model, what is the assumed memory structure including vertical hierarchy and lateral distribution employed, managed, and optimized by the runtime system on behalf of the user application? What are the performance trade-offs and performance opportunities of dynamic allocation and redistribution?
3) *Name space* – fundamental to the operation of the runtime system is the naming of local and distributed objects. A key research question is the trade-off between the merits of global named objects with simplicity of access representation and distributed or localized naming that reduces overheads of virtual addressing and translation. Questions include what constitute first class objects including the possibility of named executables (e.g., threads) for program control. PGAS represents one example of name space definition.
4) *Interface Semantics and Protocol* – the Runtime complements both the compiler and the operating system. It supports the compiler by bringing information about the system state while supporting the operating system by bringing information about the application requirements. It exploits architecture mechanisms where possible and provides additional functionality for dynamic control where necessary. The flow of information in both directions requires an advanced protocol between the runtime system and the boundary conditions of the OS and compilation layers.
5) *Introspection* – the runtime system can be open loop or closed loop as determined by the degree of introspection incorporated in the control algorithm and policies. There is a wide range of freedom of choice and runtime systems may vary significantly depending on the control strategies adopted. The control space defined and the control state transition methodology distinguish among models. Machine learning may be employed both in real time and post mortem. A major research issue is the control model for runtime system introspection.
6) *Locality and Distribution* – the major tradeoff of a scalable system computing is the exploitation of locality for reduced latency effects and the distribution of data and work to benefit from parallelism. The runtime system as part of its control strategy must balance these to match the demands of the application and the capabilities (and overhead costs) of the system architecture. This active control is a critical question in the design of the runtime system and its interaction with the compiler and OS.
7) *Reliability* – future reliability techniques may have to extend well beyond checkpoint-restart as the MTTI diminishes below the critical point where the time to checkpoint is greater than the time between single point failures. The runtime system will play an important role in supporting fault tolerance techniques in cooperation with the operating system and the programming/compilation contributions. How this is to be done is a research question of importance.
8) *Energy* – reduction in energy and bounding power is critical to the practical deployment and use of exascale systems. The runtime system may play a role in reduction of energy consumption in conjunction with the architecture, OS, and

possibly the compiler. An open research question is the strategy for such energy mitigation and the means by which the runtime system may contribute.

9) *Interoperability* – future complex user jobs may involve multiple interoperating application and data analysis codes working in ensemble to produce a final scientific result. The runtime system will be responsible, at least in part, for the data flow and control signaling between the user application and other constituent executables making up the entire workload. A current research question to be resolved is the contributing role of the runtime system to the application interoperability.
10) *Architecture Support* – currently conventional architectures include limited hardware support for large-scale computation or runtime systems. Thus many low level mechanisms will have to be implemented in software with the consequence of overheads and the implications for parallel granularity. Research in runtime systems will help answer the question of what innovations in architecture may advance the goals of efficiency and scalability.
11) *Performance modeling and evaluation* – The design, operation, and assessment of runtime behavior will depend on as associated performance model. Such a model should be employed to guide the design, control the operation, and support comparative evaluation of alternative approaches and runtime systems.

Detailed Questions

Prior art under the XSTACK program and other endeavors have educated many in the field? But this emerging field has not come together to discuss open research questions in the context of this work. This body of work should be exploited to augment other discussions as plans are being formulated to guide future research projects under ECI.

- What aren't we answering the questions within the context of the XSTACK prototypes and other runtime research that has been accomplished?
- What remains to be achieved from the accomplishments to date?
- Within the scope of current findings, what does the runtime community agree and disagree on?
- How do the application drivers benefit or not from the adoption of runtime system techniques?
- If old approaches are not working at least for some classes of driver applications, to what degree are dynamic runtime methods likely to succeed?

Task is a general term for the work units or modules that capture an ensemble of work which is separately schedulable and named (possibly with separate id space). Modularity, encapsulation, hierarchy, precedence constraints, preemption, granularity, and intra-task fine grain parallelism are all issues to be determined through research. This is a major property of a runtime system and distinguishes among the many versions of runtimes that may exist. Included are:

- What is appropriate task granularity? This is a trade-off of overhead and parallelism. How to mitigate overhead to achieve the finest practical task size?
- How do tasks communicate? Are they pure value-oriented or do they engage in global mutable state? Do they communicate with each other as through externally accessible registers?
- In what way do tasks synchronize? Are synchronization objects like dataflow or futures employed or simple BSP barriers?

- What are the scheduling policies? Is a task non-preemptive (goes to completion) or preemptive to allocate execution resources to more urgent work?

Communication and messaging is fundamental to parallel computation for data transfer and control. The runtime system makes use of low level networking media and communication protocols as provided by the hardware architecture supervised by the operating system. A diversity of alternative logical and physical methods may be implemented and deployed. The choice of these determines the form and function of the runtime systems and their strengths. Specific issues include:

- Communication primitives including point to point, one to many, and all to all.
- Possible generalization to encompass all data movement requirements.
- Intra-runtime communication across multiple nodes facilitating performance portability.
- Quality of service issues including associated resilience support.
- Virtual communication fabric for introspection information.
- Message-driven computation and scheduling to move work to the data.
- Should the control layout be matched to or independent of the data layout?

Resource Management along with task scheduling is a major part of the responsibility and role of the runtime system. The OS is assumed to allocate system resources to the runtime to manage for the execution of an application. The runtime applies these resources to the application and returns them upon application termination. Resources extend beyond cores and memory to include energy and system-wide communication channels. Among issues of concern are:

- How to best achieve load-balance and how does the application level interact with the runtime resource manager?
- What is a useful level of granularity of work to be allocated to resources and can the runtime perform aggregation of tasks to compound tasks on the fly for efficiency?
- How does the runtime expose and exploit locality? What are the dimensionality factors of locality and how is it measured?
- What is the protocol between the OS and the runtime associated with the transfer of resources and how can the OS take back resources when necessary from an executing runtime system?
- Which aspects of the applied resources need to be adaptively controlled? What are the feedback loops and the control policies they employ? Does this involve advanced Kalman filters or game theory to devise?
- To what extent does the user advise the runtime about required (or recommended) resource usage and how is this information conveyed?

Resilience is essential to effective operation and application of exascale systems. New methods of achieving reliability are likely and many questions of what such strategies are possible and how the runtime system software will contribute have to be resolved. Detailed questions to be addressed throughout the research program include:

- Interfaces to the applications, programming models and environments, and other runtimes and the protocols for information transfer among them.
- Reliable stores at every level and memory types.
- Implications of reliability for task scheduling and distribution.

- Means of error detection provided by the runtime system and the types of errors (error model/coverage).
- Runtime error notification framework with other system components.
- Roles and responsibilities of runtime in response to detected errors for recovery.

Program Actions are required to move the research forward. While a total research program has yet to be devised, some key elements are already recognized as essential at the initial stages. These include:

- Need for analysis of common and distinct concepts recognizing their similarities and differences.
- Need for studies of applications to identify those that could benefit from runtime strategies and methods.
- Determining what requirements have to be satisfied. Is there a particular class of applications being addressed? What can the applications people tell us about their applications?
- Need for point design studies of emulated or projected exascale systems.
- Need for studies of runtimes on existing systems and emerging systems.

Summary of Critical Research Questions:

Resilience
- What types of resilience interfaces and underlying implementations are appropriate for exascale?
- What types of reliability protocols and composition capabilities?
- What are the right divisions of labor/functionality between runtime/application/OS?

Introspection (information)
- What are the right tradeoffs between quality and cost?
- What information should be provided at each level? And what granularity/aggregation?

Naming
- What virtualization of naming is needed/affordable for resilience/load balance/elasticity/etc.?
- What forms of local/global/regional naming are appropriate? And are they visible to applications/runtime/hardware?

Location
- Analogous questions as for naming

Communication
- What sets of primitives?
- What inter and intra communication services and are they virtualized? QoS?

Scheduling and Placement
- How to compose schedulers and placement services?
- What are exposed as controllable by other layers? Backpressure?
- What new capabilities are needed for scheduling and resource management to deal with dynamicity and task execution models?

Resource management
- How to best allocate X (e.g., power, cores, memory)?
- What kinds of cross-layer interfaces, coordination and control?
- Detection/measurement of locality and exploitation?
- What is software/runtime/OS/hardware partnership for locality?
- What role does adaptive control play? And where should it not play? Fairness? X-layer control?

Session VII and VIII – Runtime Roadmap

Chairs: Dave Montoya, LANL; Kathy Yelick, LBNL

The target timeframe for an exascale system is 2023. Work is going on across all the ECI areas related to this goal by Laboratories, Universities, and Industry through various programs. The focus for this report is the Runtime area within the ECI. Challenges include quickly evolving and diverse new hardware architecture development, emerging programming models to further implement asynchronous processing capabilities, evolving deep memory hierarchies, increasingly complex application workflows and others related to power and resilience requirements. The runtimes of tomorrow needs to be more communicative than past models, dynamically interacting and scheduling between application and system resource layers.

Roadmap approach and scoping

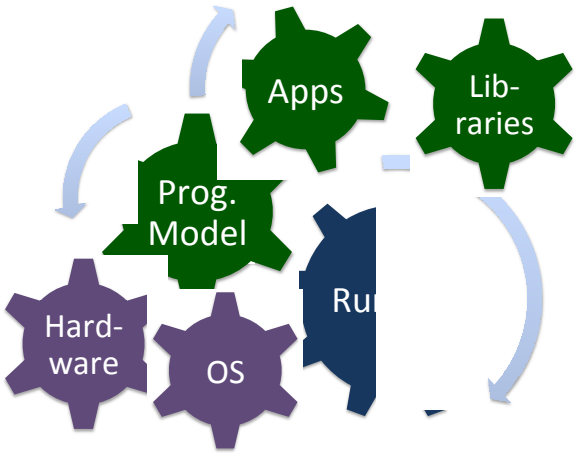
Runtime research is being done through ASCR X-Stack and OS/R programs, Design Forward projects, the PSAAP II program and other efforts. ECI must interface with research in Programming Models, Operating Systems, Application and Library development, and Hardware Architectures. A key strategy is to identify a Production Implementation Strand and a Research Strand, with a process for selecting and hardening research results and moving them into production. This is important to establish initial targets for higher-level software, while at the same time permitting innovation in runtime research that may lead to radically different solutions with better overall performance and productivity.

This process also helps move research efforts forward. As requirements are identified from experience on the Production Development Strand it will drive exploration of new research problems, while at the same time informing the research activities of Best Practices.

Bounding

The questions that were repeatedly brought up during the workshop was, “What do we mean by “runtime” and what are its requirements?” While there was not entire agreement on the boundaries, there were generally two interpretations of the terminology, one focused on user-level constructs that exist within a single executable and are part of the programming model implementation and the other involving management across workflows, interaction with external networks and data sources, and use of privileged instructions reserved for the operating system. The ideas of virtualization from the commercial data center world further complicate any clean separation, as a single application may be packaged with its own operating system image and multiple executables. Discussion around requirements came from applications needs through Programming Models and also from the system environment, hardware through the operating system and associated interfaces. New usage models,

such as computing based on real-time arrival of data or massive high throughput workloads from uncertainty quantification also add to runtime requirements. In spite of these difficulties, separating the Programming Model runtime from the more inclusive System Runtime would probably facilitate future discussions and may allow interfaces to develop to aid in separate research and development activities.



As suggested by the above figure, the Runtime components interact with other parts of the exascale system, generally the Programming Environment above and the System Environment below. The Programming Environment is the interface to the applications and libraries; this needs to include communication, and needs to co-exist and interact within the larger environment for resources and scheduling. The System Environment includes the Hardware, Operating System (Node OS, System OS, Enclave OS), which include various implementations. The System Runtime has long-term footprint within the environment, which includes workflow and data analysis implementations, and has to interface to protection and resilience components. We need an exercise to bound what we see as the Runtime in order to facilitate useful discussions and eventually develop common interfaces and components.

Questions that need to be incorporated into this bounding exercise include:

- Can your runtime work with multiple Programming and Execution Models
- Can different runtimes (or components) use shared resources or shared data from multiple applications
- What are different usage models / application patterns
- Can a runtime be incorporated into a library approach
- How does data move between runtimes

Refining the bounds of the Runtime is an on-going effort but needs a process and deliverable points to the production strand discussed earlier. This may also require an organization to set dates and target deliverables. The approach is further discussed in the following convergence discussion.

Convergence

As we have seen with past runtime convergence efforts such as MPI, significant community experience was required before standards processes and organizational approach for support is developed. There were several message passing interfaces that

had been widely deployed in applications prior to the definition of the MPI standard, and it continues to evolve based on application and system requirements. It is felt that in regard to an ECI Runtime it is to early in the process go down the standard route and develop a specification, although it should evolve to this at some point as it matures. It was recommended that a best practices approach could be used to identify common runtime ideas that have been demonstrated to be effective and evolve to the point where interfaces and eventually standards could develop. However, fluidity is needed to adjust to changing programming models, OS, hardware architecture, etc. A process for establishing an initial production strand and a process for incorporating research results into that production strand on an ongoing basis should be established within the next 6 months.

An approach discussed at the workshop is to view the runtime as a set of services rather than a monolithic layer. The first step would be to identify a set of service categories and then a minimal set of services in each based on a best practices approach. The services would identify interfaces and communication points with other services and with other parts of the software stack. These services may have more interdependencies than in the past because of feedback loops needed for adaptability and there are still open research questions regarding the ability to have separable services and how they would interact. Developing a common API as in the Argo/Hobbes backplane is a good example of a consolidation effort.

This initial process should begin by taking a survey and inventory of current efforts, their service/interface points, and the extent to which the difference services can be decoupled. This will be an introspective exercise by the community and would need to include runtime efforts by programming model developers as well as any standalone (programming model independent) runtime efforts. Having separate discussions about these Programming Environment runtimes and the System Runtime seems appropriate, since the inventory of services and interface points for the latter require expertise further outside the core group and may require discussions with some of the OS and industry efforts for things like an interface for hardware resource arbitration.

This would allow us to converge on a few runtimes that interoperate or establish attributes for interoperability for all to move toward. Bringing narrow communities together to understand their similarities and differences is a goal of this process. This exercise should be done within the next 6 months.

Industry Integration

Industry is key in attaining exascale systems that deliver application results, and they will be expected to deliver the machine and production environment to attain that goal. The challenge is how to incorporate the research efforts that are occurring within the DOE and in industry to develop the solution needed. There is also the case that if we want to influence hardware to support capabilities such as tasking models, it needs to be incorporated into their hardware designs up to 4 years before delivery for proper design, integration and testing. We need to both leverage and influence what industry is developing regarding OS and hardware architecture.

Issues encountered include that companies vary on willingness to share ideas and collaborate depending on what they consider to be intellectual property and existing

partnerships. The common factors are that we all need use cases to better define requirements and the availability and use of simulation and testbed environments to prototype research ideas in pursuit of the use cases.

As part of the convergence discussion, Industry efforts of importance to the Runtime should be mapped to understand potential overlaps and to define service/interface points. This exercise should be done in the next 6 months

Characteristics of an Exascale Runtime

It was not the goal of this breakout group or workshop to define the services that a runtime system might support or to identify the interfaces between them. However, the discussions covered a number of questions that may aid in future planning activities.

How dynamic should an exascale runtime be?

There are many reasons to desire dynamic decision making in the runtime systems. Because of hardware variability and application level adaptivity, attempts to estimate the runtime cost of various components in advance is very challenging, so using such information for load balancing and scheduling can be difficult to impossible. There are various interpretations of the term "dynamic," which can range from job launch time, at discrete points mid-execution (which can be globally), continuously on-the-fly (which probably means locally or at least hierarchically). The overarching questions for the research community are whether dynamic runtimes based on one of these definitions are required for exascale performance, whether they can improve productivity and conversely are they capable of achieving exascale performance. There is evidence that dynamic runtimes can and have been effective on petascale machines, so another version of this question is what application-specific information needs to be communicated to the runtime system and what policies and mechanisms can be built into a runtime system in an application-independent manner.

How much parallelism should be exposed to the runtime system?

Exascale systems will have much higher concurrency than current petascale systems to address the performance growth without clock speed improvements, In addition, concurrency is needed to mask latencies of memory accesses, communication, I/O, and synchronization. For portability across systems an attractive approach is to have programmers somehow express all available parallelism (or at least much more than they expect to need) and have either compilers or the runtime system map this to the limited hardware resources. The question from the runtime perspective is how much concurrency should be exposed to the runtime and how can that be managed to keep resource utilization (memory, cores, bandwidth, etc.) under control. Traditionally, runtime systems have had to "throttle" some concurrency, but better hardware and system support for operations like thread scheduling may make this less important. In general, the group felt that the two extremes -- all available application concurrency or just what is required for available hardware cores--are probably both impractical.

How should application information about locality and load balance be communicated to the runtime?

Many application domains have information that can be used to optimize resource utilization, e.g., computation and memory load balancing. For example, some adaptive mesh refinement codes are currently balanced using a global analysis based on space filling curves to evenly distributed load based on approximate notion of cost. Direct linear algebra solvers have a complex set of task dependencies that can be expressed as a directed acyclic graph that can be dynamically scheduled; yet layout of the data and tasks is often static. Dynamic runtimes systems today often allow some type of control over mapping in a distributed memory setting, but the mechanisms for expressing those mappings are not common across approaches. There was also extensive discussion about how load balancing should be done, what units of work should be used, what type of naming should be used, and how dynamic both load balancing and scheduling should be. Current research projects are exploring various approached in this space.

How should the runtime interact with other parts of the system?

The complement to the previous question about application information also arose in considering interactions with the rest of the system. For example, how the runtime system should interact with the storage system, with operating system protection and resource management, with resilience mechanisms (failure detection and recovery) and energy management. The overarching question for each of these topics was whether a feature of the system should be exposed to the programming model, ignored entirely, or hidden by the runtime system.

2015 ECI Runtime Systems Workshop - Agenda

Location: Rockville Hilton, Rockville, MD

Wednesday, March 11, 2015		
	Topic	**Speaker**
1:00pm - 1:20pm	*Welcome and Introduction – RTS* *Exascale Computing Initiative* *ECI NNSA ASC Perspective*	*William Harrod* *Director of Research, Advanced Scientific Computing Research (ASCR)* *Thuc Hoang* *Program Manager, Advanced Simulation and Computing (ASC)*
1:20pm – 1:40pm	*ParalleX/HPX Runtime for Exascale*	*Thomas Sterling* *Indiana University*
1:40pm – 2:00pm	*Open Community Runtime*	*Joshua Fryman* *Intel*
2:00pm – 2:20pm	*Adaptive Runtime Systems for DEGAS*	*Kathy Yelick* *Lawrence Berkeley National Laboratory (LBNL)*
2:20:pm – 2:40pm	*ASCR Runtime System Summit Report*	*Milind Kulkarni* *Purdue University*
2:40pm – 3:00pm	*Break*	
3:00pm – 4:30pm	*Parallel Session I: Runtime Systems Architecture (set 1)*	*Chair: Jim Laros* *Sandia National Laboratories (SNL)*
	Parallel Session II: Runtime Systems Architecture (set 2)	*Chair: Joshua Fryman* *Intel*
4:30pm – 4:45pm	*Report Back Session I* *Runtime Systems Architecture*	*Jim Laros* *SNL*
4:45pm – 5:00pm	*Report Back Session II* *Runtime Architecture Debate*	*Joshua Fryman* *Intel*
5:00pm	**Adjourn**	

Thursday, March 12, 2015		
	Topic	**Speaker**
740am – 8:30am	*Continental Breakfast*	
8:30am – 8:40am	*2015 ECI RTS Workshop*	*Sonia Sachs* *ASCR*
8:40am – 9:00am	*ASC's Code Needs*	*Todd Gamblin* *Lawrence Livermore National Laboratory (LLNL)*
9:00am – 9:20am	*Exascale Runtime Systems Architecture and Design*	*Ron Brightwell* *SNL*
9:20am – 9:40 am	*DAG-Based Runtime Systems A Uintah Perspective*	*Martin Berzins* *University of Utah*
9:40am – 10:00am	*Adaptive Runtimes: Charm++ Case Study and Lessons for Exascale*	*Sanjay Kale* *University of Illinois Urbana-Champaign (UIUC)*
10:00am – 10:20am	*ARGO*	*Pete Beckman* *Argonne National Laboratory (ANL)*
10:20am – 10:40am	*Break*	
10:40am – 12:00pm	*Parallel Session III: Runtime Systems Design (set 1)*	*Chair: Pat McCormick* *LANL*
	Parallel Session IV: Runtime Systems Design (set 2)	*Chair: Sanjay Kale* *UIUC*
12:00pm – 1:00pm	*Lunch*	
1:00pm – 2:00pm	*ASC Panel on RTS Topics* *Optimizing RT, Tools and Interfaces A Task-Based PM for SC*	*Kevin Pedretti,, SNL* *Martin Schulz, LLNL* *Josh Payne, LANL/Ben Bergen, LANL*
2:00pm – 2:15pm	*Report Back of Session III*	*Pat McCormick* *LANL*
2:15pm – 2:30pm	*Report Back of Session IV* *RTS Design Session*	*Sanjay Kale* *University of Illinois Urbana-Champaign (UIUC)*
2:30pm – 2:50pm	*MPI and OpenMP Runtime*	*Pavan Balaji* *ANL*
2:50pm – 3:10pm	*Global Arrays Runtime*	*Daniel Chavarria-Miranda* *Pacific Northwest National Laboratory (PNNL)*
3:10pm – 3:30pm	*Break*	
3:30pm – 5:00pm	*Parallel Session V: Runtime Systems Research Questions (set 1)*	*Chair: Thomas Sterling* *Indiana University*
	Parallel Session VI: Runtime Systems Research Questions (set 2)	*Chair: Andrew Chien* *University of Chicago*
5:00pm	**Adjourn**	

Friday, March 13, 2015		
	Topic	**Speaker**
7:40am – 8:30am	Continental Breakfast	
8:30am – 8:50am	*Legion: Runtime System*	*Pat McCormick* *Los Alamos National Laboratory (LANL)*
8:50am – 9:05am	*Report Back of Session V* *RTS Questions Working Group*	*Thomas Sterling* *Indiana University*
9:05am - 9:20am	*Report Back of Session VI*	*Zoran Budimlik Rice University*
9:20am – 10:30am	*Parallel Session VII: Runtime R&D Roadmap (set 1)*	*Chair: Kathy Yelick* *LBNL*
	Parallel Session VIII: Runtime R&D Roadmap (set 2)	*Chair: Dave Montoya* *LANL*
10:30am – 10:45am	Break	
10:45am – 11:00am	*Report Back Session VII* *Runtime Roadmap VII*	*Kathy Yelick* *LBNL*
11:00am – 11:15am	*Report Back Session VIII* *Roadmap_Session VIII*	*Dave Montoya* *LANL*
11:15am – 11:35am	*Scalable Storage I/O workshop report summary*	*Rob Ross* *ANL*
11:35am - 12:00pm	*Workshop Summary*	*Chairs: Pete Beckman, ANL* *Rob Neely, LLNL*
12:00pm	**Adjourn**	