Exascale and Beyond: Configuring, Reasoning, Scaling

Report of the 2011 Workshop on Architectures II: Exascale and Beyond, held August 8-10, Albuquerque, N.M.

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

2011 Workshop on Architectures II: Exascale, and Beyond: Configuring, Reasoning, and Scaling

Organizers: Lenore Mullin, William Harrod, Sonia Sachs (ASCR DOE), Richard Lethin (Reservoir Labs), Arun Rodridgues (Sandia National Labs), John Shalf (LBNL), Mark Snir (UIUC), Thomas Sterling (Indiana University)

Written By: Stephen Booth (EPCC), Dan Campbell (GTRI), Andrew Chien (University of Chicago), Richard Lethin (Reservoir Labs), Lenore Mullin (ASCR DOE), Arun Rodrigues (SNL) Ron Sass (UNC Charlotte), John Shalf (LBNL), Mark Snir (ANL), Tom Sterling (U Indiana).

Funded By: US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR)

Held On: August 8-10, 2011, Sandia National Laboratories, Albuquerque, NM

Contents

1 Introduction

The push toward exascale computing (1) and beyond is torn between contradictory and possibly incompatible goals. On the one hand, one wants a system that will be available within a short span of time, on a relatively small development budget, and with the least changes possible to the current software stack. On the other hand, one wants a system that is orders of magnitude more efficient in its energy consumption. An exascale system could be as different from current systems as scalable parallel systems were from vector machines at the time of the last major transition in the field, but the amount of application software developed at DOE in the last decades makes it undesirable to take a "clean sheet of paper" approach, if at all possible.

Furthermore, one wishes to explore and develop distinct exascale architectures to reduce risk while maintaining some unity so that application codes easily port among them. Finally, the design of an exascale system may involve a large number of vendors, some of which have limited experience in the field of High-Performance Computing (HPC). A proper integration of these multiple contributions requires a proper definition of many new interfaces, as well as consistent design principles for the handling of cross-cutting issues such as resilience, performance, and power management.

While DOE will not control the design of most of the components of an exascale system, it can influence their design and it can have a large impact on the overall system organization — both hardware and software[1]. Thus DOE Research and Development involvement can focus on these system level design issues and some degree of customization. One key area of this research is in modeling. The complexity of future exascale systems and the lack of practical experience with many of the technologies that may be required at exascale may well require a more formal approach to the definition of exascale system organization, departing from the ad-hoc evolution of supercomputer architectures in the last decades.

We believe that abstract machine models are key to supporting the Department of Energy's (DOE's) goals in exascale computing.

- They should define the general system organization principles: the key abstractions, their layering and interaction, key feedback loops, and the general philosophy for handling performance (computation time), energy consumption, and resilience.
- They should define layers that are compatible across systems. If, as some believe, exascale systems have to be treated as aleatory systems, then this definition becomes more complex and much harder to define and verify. The abstract machine model should provide a precise definition for what it means to "produce the same result," including a possibly aleatory computation outcome, characterization of errors and, last but not least, common performance model.
- They should provide a mechanism for exploring different implementation approaches to the same common model. The abstract machine model defines the exploration domain.

[1] This is the custom-commodity approach.

- They should facilitate verification and validation of complex exascale systems, providing a top-down approach to V&V that is a necessary complement to the bottom-up approach normally used in testing.
- They provide a framework for handling global trade-offs on the energy budget, reliability, and software vs. hardware support.

This report will do the following:

- Explain in more detail what we mean by "Abstract Machine Models"; how have such models been used in the past; and how they could be used in the future, for exascale,
- Explain how abstract machine models can be used to achieve the goals listed above,
- Outline the main roadblocks to the use of AMMs in achieving these goals,
- Propose research directions to overcome these roadblocks,
- Prioritize research in the use of AMMs for exascale

2 Abstract Machine Models

An *abstract machine model* (AMM) (or just "abstract machine") is a representation of a class of computing systems that facilitates design, experimentation, performance and cost projection, and refinement and optimization without having to develop a complete detailed design of a single particular system. An AMM serves as a conceptual interface between the low-level (bottom-up) implementation details, including hardware technology, architecture (organization and ISA), and operating system, and the high-level (top-down) semantic and policy details, including the overall execution model, application characterization, and possible programming interfaces.

An abstract machine model is defined over a set of dimensions to expose critical operational parameters over specific ranges in order to explore design properties, their functionality, and their optimality. An AMM may be manifest as a set of analytical relationships among system-independent variables (e.g., number of nodes) and the operational consequents such as performance or energy consumption. More complex abstract machine models may be captured as a numerical representation (e.g., a queuing model) or as a simulation. Depending on the development methodology, more than one abstract machine model, or equivalently refinements in the level of abstraction, may be employed to reflect different aspects of a single target system to be ultimately delivered. As a design interface, such a model can provide simultaneous access by different users of a given model. The result can be a rapid and well-informed sequence of design cycles exhibiting convergent optimality through iterative refinement.

An important use case for machine modeling is performance prediction. Performance prediction plays multiple roles both in R&D and in operation, enabling exascale systems. Most notably, performance prediction will be required for the following:

- Platform Design Space Exploration: during the design of a new computing system, estimation of the performance of that platform on applications of interest as a function of available design choices.
- Application Design Space Exploration: during the design of an application, estimation of the performance of the application on computing platforms of interest as a function of available application design choices.
- Compilation and Optimization: during the translation of the application from human-readable task descriptions to machine-executable instruction streams, estimation of the performance of the application on specific platforms of interest as a function of possible mapping variations such as algorithmic forms, temporal and spatial placement of instructions and data required to achieve the computing task.
- Runtime: during the execution of an application on a particular platform, estimation of the performance consequence of available adjustments to the platform and instruction stream. These adjustments might include such elements as voltage and clock frequency supplies to individual processors, configuration of memory topologies, placement of instruction streams and data onto specific execution units, and alternative organization of instruction streams.

These models of performance may include relating the resilience consequences with various adjustments such as describing the relationship between the rates of failures and the settings of power supply voltages on cores.

In all cases, the abstract machine model is a key component to the performance estimation. The purpose of the abstract machine model is to represent a physical computing platform in a form that can allow reasoning about aspects of the platform more quickly and cheaply than by direct observation. The required reasoning determines the aspects of the model that are required. In order to maximize the benefits of using a model rather than a physical system, it is useful to make the model as simple as possible while still enabling the needed reasoning. As a result, an abstract machine model is desirable, and selection of the appropriate level of abstraction is critical to the success of exascale system development and deployment.

An abstract machine model consists of the following:

- Some form of representation of program demands
- System component capacities
- System component functional properties
- Policies for orchestrating computation
- Costs associated with each instance of functionality
- A schema of composition that permits a system description comprising a multiplicity of functional elements
- Behavior metrics
- Objective functions

Different facets of a design process and/or team can contribute distinct elements within this composite abstract model. Individual modules of the model may initially be simplistic with only one or a few parameters and their interrelationships. These may later be replaced with higher fidelity descriptions to achieve superior resolution and confidence through advanced complexity of design and modeling.

There is a wealth of experience with simulation and emulation modeling tools and these should be applied wherever practical in realizing the new abstract machine model strategy of advanced machine design and system usage policies. Yet the existing state of the art in machine modeling is not sufficient for exascale. Certainly current methods are unlikely to scale to the large number of components and tasks needed for exascale. But more challenging is the number of machine attributes (such as aleatory) and the way that exascale systems will involve "vertical" execution models, making it difficult to reason about exascale at just one level of abstraction at a time while capturing cross-layer interactions. This report will highlight areas where new research and development is needed.

3 Definitions

Abstract Machine Modeling Language (AMML): An abstract machine modeling language is the formalization of a particular parametric model for a class of machines (a design space). This defines the space of machines that software tools for exascale must address, and informs programmers as they design algorithms and software that should utilize the machine and be portable across the space. It enables programmers to reason about what aspects of the machine are relevant for performance as they design their algorithms/apps and decide how to map them onto a family of machine architectures. It provides a substrate for developing strong evidence to influence vendors and understand trade-offs during the codesign process, by expressing what the valuable aspects of a machine are.

Abstract Machine Model (AMM): is a representation of a particular machine within the AMML. This can be used as the referent for mapping processes (automatic as in a compiler, manually by a programmer, or by a runtime process) to optimize software to that particular machine. The model may express ranges or sets of values for groups of parameters – expressing a range of values or configurations that might be used by runtime adaptive systems or co-design processes to choose an ideal configuration.

Execution Model: A set of governing principles guiding the co-design and interoperability of system component layers in performing a specified computation. It defines a family (or class, category, etc.) of machine types for which many distinct specific implementations are possible depending on requirements and constraints applied. It establishes the state objects, their naming conventions and hierarchy, sets of actions to be performed on them, semantics and control of parallelism, composability and interoperability of context and action domains, and other ancillary properties shared across machines within the genre (e.g., reliability, performance models, etc.). An execution model is motivated by new opportunities or challenges to be addressed, often as a result of technology advances or application demands. An execution model informs a system 'decision chain' that establishes the contribution of each layer in determining where, when, and what actions are to be performed optimally.

"Horizontal" Execution Model: An inadequate consideration of the execution model as being described by an interface or abstraction layer.

"Vertical" Execution Model: In order to reach the performance goals for exascale, particularly with power, optimization and new approaches must be taken in the design of execution models that cut through multiple traditional layers. For example, the use of futures and other novel synchronization mechanisms affect hardware, runtime, operation system, and programming interface. Execution model issues are cross-cutting.

Application Surrogate: A minimal representation of a larger application. E.g., a Compact App, Mini-App, Skeleton App, Proxy App, Mini-driver, or kernel.

Compact app: Small application, having fewer features and simplified boundary conditions relative to full applications.

Mini-app: Small, self-contained program that embodies essential performance characteristics of key applications.

Skeleton app: Captures control flow and communication patter of an application. Can only be run in a simulator.

Proxy app: General term for all the above "app" approaches.

Mini-driver: Small programs that act as drivers of performance-impacting library packages.

Kernel: This represents the core computation of an algorithm. Scheduling functions could map it to a single computational node in the machine, or spread it in a complex schedule across the machine.

Joe: An archetypal application program who is more concerned about basic functionality than with optimal performance. Has no desire to understand the machine in anything more than a basic level as relies on the compiler, libraries, and other tools to achieve performance.

Stephanie: An archetypal application program who seeks the highest levels of performance for her code. Is willing to delve into the details of the machine, such as understanding memory hierarchies, optimizing communication patterns, and pursuing performance at the sake of (some) portability.

Aaron: An archetypal hardware designer, who seeks to architect the exascale machine. Requires a high-level understanding of applications, but does not require (or want) to understand all applications in great depth. Is very concerned over the low-level hardware of the machine.

Exascale Machine Model: A system used to simulate or estimate the performance of a future exascale system. Its goal is to produce useful information that can impact decisions. The model will include a specification which describes the machine at some level of detail. The machine model can include multiple levels of abstraction to address different audiences (e.g., Joe, Stephanie, Aaron), or different uses (compiler target, simulation vehicle, verification referent). The machine model may include details on relative timings of operations, speeds and feeds, presence or absence of vector instructions, network (both on-chip and system) parameters, power/energy consumption, and reliability. The machine model addresses the performance of the whole machine (e.g., processor, network, memory, IO). Examples of partial machine models or machine specifications would include processor or memory specification sheets and ISA manuals.

4 Design Space Exploration for Exascale Machines with Abstract Machine Models

We propose the use of Abstract Machine Models (AMMs) to enable systematic engagement and co-design of exascale machines *and* software (applications and system). We expect this approach will allow tangible and focused interaction between the designers of exascale software and the designers of exascale hardware, allowing both optimization of the realization of a particular AMM in a physical machine architecture and associated runtime. An AMM can serve as a static or dynamic interface for software tools, such as compilers, debuggers, verifiers, runtimes, and control systems, for reasoning about and even changing the configuration of the machine (hardware plus software), in introspective and adaptive modes. An AMM can be used as an object for simulation, supporting varying levels of abstraction and definition, trading precision for speed. An AMM also defines the range (the multidimensional space and multidimensional time) of program mapping schedules (implicit, parametric, dynamic functions that map from the domain, software operations in the application, to the range, where and when they operate in the hardware).

Ultimately, a successful research and development program, supported by sophisticated simulation and modeling tools (applied to the AMM, system software, and application software) will enable a rigorous quantitative comparison of distinct AMMs, allowing assessment of distinct system organizational approaches based on deep insights from applications and systems software.

It is important to make the distinction between a specific AMM, and the language or logic in which that AMM is described. For the latter, we use the term Abstract Machine Modeling Language (AMML).

Ideally, the research into machine modeling for exascale would be undertaken by several distinct teams, each with a charter to explore distinct AMMs (or more ambitiously >1 AMM per team), within a common AMML across the exascale R&D program. Such a process of deep collaboration, focused by the definition of the important exascale hardware and software design spaces (as embodied in the evolving cross team collaborative definition of the AMML), the comparative quantitative evaluation of AMMs, and the co-design optimization within the design space defined by an AMM together represent a highly-rigorous, robust approach to achieving usable exascale computing capability in a 2018 time frame, thereby enabling continued DOE leadership in computational science.

Abstract Machine Modeling Languages (AMMLs) define design spaces, defining the complete fundamental structure and mechanisms for use of a computing system and thereby allowing the definition and exploration of system software (runtime and operating systems) and applications structure. This definition and exploration can be *ab initio* or based on evolution from existing software and algorithm structures. Having a common

AMML across the program allows for comparison between AMMs because they are expressed within the same language and in the same terms.

Several technology studies by distinguished scientists [DARPA-HW, DARPA-SW, DOE-HW, DOE-applications] have found that exascale systems pose new and distinct challenges in scale of parallelism, dynamism of performance, and resilience (soft and hard errors and faults). These challenges dictate a prudent expansion of the design space of AMMLs to include new dimensions:

- Mechanisms that support advertising error rates, mechanisms to notify software or errors or faults, and potentially interfaces which allow software to specify error rates (at some cost in energy/performance)
- Mechanisms that support synchronous and asynchronous messaging
- Mechanisms that support thread / task creation / termination / synchronization and scheduling

Abstract Machine Models for exascale machines might also plausibly include mechanisms for the following:

- Performance monitoring
- Load balancing
- Notification of performance change imposed by the hardware
- Heterogeneous functional units
- Deep hierarchy of layers and mixed types of layers
- Global memory address, varied consistency models, and new virtualization and security mechanisms
- Means for nimble dynamic regulation of voltage, clock, placement

It is likely that different approaches to these novel exascale system issues will, in fact, define different AMMs (classes of machine designs) and as such determine how they are addressed is a critical element of the co-design process. Further, the development of distinct Abstract Machine Models (AMMs) will allow the comparison of distinct capabilities and approaches. This will further allow the alignment of layers/abstractions across different system designs, which will enable rigorous comparison for what it means to "compute the same results" and the related challenges of verification and validation of both systems and applications. Within these AMM frameworks, system implementers can safely explore varied approaches and techniques, and compare results in a meaningful and constructive fashion, moving towards exascale systems that are robustly better. Finally, AMMs will allow the instantiation of a system architecture for "self-awareness" and global management of complex tradeoffs involving software complexity, performance, energy, and reliability.

It is important to note that while the Abstract Machine Model for an exascale machine needs to include these facets, a successful exascale system might in fact hide many of these hardware services/interfaces from the application software, successfully encapsulating them in runtime/firmware systems. The modeling mechanism for such encapsulation may mirror the way that the AMM supports multiple levels of abstraction for different simulations. Such success would protect applications from the associated machine complexity, and provide a simpler application execution model.

4.1 Applications and Metrics

During the design of potential exascale computing platforms, it is beneficial to evaluate the likelihood of that platform delivering the desired execution capabilities before the time and expense of constructing the proposed platform are expended. The desired capabilities may be expressed in several ways, but must ultimately represent an improvement in the ability of the platform's users to perform the required computation. Historically, this has been represented by the peak rate of floating point operations that the platform is either theoretically capable of, or can achieve on an idealized benchmark, such as LINPACK. It is well understood that this representation does not correlate well with the performance of a platform on most applications. Fundamentally, a platform's capability requirements are determined by the applications that are likely to be run on it. Since different applications are limited by different aspects of the computing platform, a single metric (such as peak FLOPs) can not suffice to predict the performance of an exascale system on applications of interest.

Because peak FLOPs is not a sufficient metric for the capability of a platform to execute applications, the criteria for achieving exascale is often described [1] as one thousand times petascale (or the most capable computers as of approximately 2010). This can be viewed either as a 1000x improvement in time to solution for existing problems (strong scaling), or a 1000x improvement in problem size that can be computed in a given period of time. The design criteria, and thus, the goals of design space exploration for a DOE exascale computing system must be defined in terms of applications critical to DOE. The deployment of exascale platforms will cause applications with new classes of computation to emerge in response to the available resources, but it is only possible to make statements of relative capability in comparison to applications that currently exist. Therefore, to evaluate the suitability of a proposed design, it is necessary to estimate whether that design can deliver a 1000x improvement in existing applications critical to DOE.

The combination of an abstract representation of application needs and an abstract machine model that represents sufficient information about the platform may allow prediction of performance with acceptable accuracy. Recent success [2] in isolating the aspects of applications that most critically map system capabilities to achieved performance give reason for optimism that such performance predictions may be successful. A key goal for early phases of exascale systems research will be to find the appropriate abstractions that allow sufficiently accurate performance estimation of known applications on designed architectures.

In existing petascale systems, both energy consumption and execution time are driven mostly by communication [3]. Since total energy consumption is a constraint on exascale system design, it is likely that this situation continue or become even more pronounced in exascale systems [1]. Therefore, a machine model abstraction that facilitates estimation of the total communication cost of an application is likely to be an absolute requirement for performance prediction.

The communication required by an application is a property of both the application and the platform. For example, the size of first-level cache will affect how many loads from more remote memory must be made. Furthermore, the cost of communications is, on many platforms, at least partially determined by the temporal and spatial properties of the sequence of communications. In most applications, the sequence of communications has some bounded degree of flexibility which can be exploited at compile- or run-time.

For the purposes of design space exploration, it is necessary to construct the model of a particular machine in terms of a relatively small number of simple metrics that can easily be extracted or found by inspection. Empirical models of complex subsystem behaviors of a full system will not be available during the design phase. Benchmarks may be run on simulators only if they are extremely simple. Existing modeling approaches for communication complexity consider alternatively traffic between cache and main memory, the cost of communication between nodes in a message passing cluster, and the impact of network topology on communication costs, but are not integrated and each considers only specific execution models [3].

A unification of the disparate communication cost models may yield a manageable number of metrics which could be found by microbenchmarks run on simulators. These metrics could include latencies and bandwidths between all possible connected nodes on the communication network of a system, and quantity of storage available at each node; each of which as observed in the context of various communication sequence types that constitute the bulk of current DOE applications.

The work in study of patterns of computation, memory access, and parallelism has shown that most applications can be categorized into one of a handful of computational patterns, which contain varying proportions of a handful of memory access patterns, and are parallelized in only a handful of patterns. Analytic and empirical techniques may be employed to find the quantity and proportion of each of the patterns and, for weak scaling, predict them at the desired problem size. A fairly small number of microbenchmarks have been developed to quantify the cost of existing communication patterns on current systems. These metrics and their corresponding microbenchmarks may be extended to account for the heterogeneity and more complex communication hierarchies that are likely to be present in exascale systems. Likewise, the communication models of applications may be extended to account for arbitrarily deep hierarchies, and heterogeneity within the platform.

With such extensions, it is feasible that a lower bound of communication cost for an application on a particular platform may be found by manual or automated analysis. Since it is likely that communication costs will dominate both time and energy for exascale computing, such analysis of the intersection between application and proposed architecture may yield feasible bounds on performance.

5 Verification, Computational Reproducibility and Uncertainty Quantification

As an example of the benefits of having formalizations of machines available as abstract machine models, consider verification of exascale systems. The software of an exascale system provides a formalization of the application; with the addition of a model of exascale machines we can pose (and answer) through rigorous automated methods, questions about performance and correctness that are vital. Once enabled by formal models of the machine, we can reach for new heights in program specification with concurrent benefits in performance and in reduced software life cycle costs for scientific applications, as applications expressed at a high level provide properties and semantics for more parallelism and more performance.

A focus on verification is important for exascale abstract machine modeling for two reasons. First, pragmatically, verification of a model forces rigor in definitions of models and helps resist the temptation to throw undefined or unsound concepts into the machine model. Second, and more importantly, the anticipated complexity of exascale machines will demand more emphasis on automatic verification than has been the case in current HPC efforts. This complexity stems from the scale of the machine (number of devices), the expansion of the set of machine attributes (new voltage controls, synchronization, mechanisms, heterogeneity, etc.), and the nondeterministic, or aleatory, behavior (from faults or other probabilistic mechanisms). Further, it is possible in modern VLSI to provision machines with far more transistors that can be simultaneously powered; the result is that regulation systems must be in place to assure the machine operates within limits, maintains integrity, and is safe. With such complexity and risk, verification approaches are needed and thus can provide confidence that the behaviors that emerge at runtime are within expectations, that the results of computations are meaningful, and that the exascale hardware will not e.g., melt.

The exascale project has a breadth of established verification techniques from other areas of computer science to draw on. In the field of hardware design, there is a rich set of advanced technologies and off the shelf commercial tools for writing specifications and checking them. The field of embedded systems provides verification technologies for modeling at multiple levels of abstraction, combining those models in hybrids (cf. Pnueli, Lee/Ptolemy), reasoning about faults, and for assuring the correctness of real time. In systems software, there are techniques involving higher order logics for expressing knowledge ontologies and behavior of complex concurrent and nondeterministic software, and for finding, expressing, and checking proofs of correctness.

These formalizations can assist with component development. Specifications of the expected hardware behavior can be the referent for verification processes in hardware design, e.g., formal verification of chips before they tape out. Providing specifications to hardware vendors that come from the system level modeling is vital to prevent “dropping the ball” with uncommunicated requirements, particularly in the case where the components are breaking from the standard improvement curves on conventional processors and system design.

Formalization can also assist with production. When machines are being assembled, tested, and running acceptance tests, these abstract models can provide the vital referent for determining if the system is operating to spec, and for isolating any faults or defects in construction. Note how this checking for production flows naturally into processes for operation, defecting defects in operation or rapidly isolating failed components in the machine. Specification intended for operation, that are driving resilience and fault recovery processes in operation through aleatory behaviors, can be used in production and testing to determine if machines are correctly built. Consider – if the nature and scale of exascale means that devices will run closer to margins and will occasionally fail, how does the customer for an extreme scale machine determine whether a delivered machine passes acceptance tests? How does the vendor assure that the machine will pass tests before shipping? Models may be useful up front even in the procurement process to specify requirements for vendors producing the hardware and software of the machine, and for vendors to communicate requirements to subcontractors providing devices, compilers, runtimes, etc.

While these established technologies are the foundations upon which a science of verification of computational scientific codes and exascale hardware can be built, there are many pragmatic challenges in applying them, and expanding the scope of these technologies to exascale.

We note some additional benefits of a verification-oriented approach to abstract machine modeling.

This focus on formalization of the machine will spill over into more rigors in the engineering of the programming approaches and the program mapping approaches. In particular, high-semantic-content specifications of the software and the scientific system under study will be needed for combined verification of the correctness of applications and their implementation on exascale systems. Exascale research and development should strive for the goal of developing systems rigorously specified to allow joint expression and checking of proofs of correctness for the combined application + mapping + system software + hardware combination, even in the face of nondeterministic and aleatory hardware behavior.

Furthermore, this rigorous specification will have benefits in terms of providing for performance and productivity. Specifications of information not normally found in typically HPC programs, such as formalizations of properties of commutativity, provide additional degrees of mapping freedom for compilers to exploit in order to find new dimensions of parallelism, for the massive increase in required concurrency for exascale hardware. Furthermore, specifications of the applications, including information needed to verify systems with aleatory properties, such as required precision in values, provides for productivity and reduced life-cycle costs, as this information documents the system and provides information that allows for rapid and correct porting and scaling of software to future systems.

One major benefit of taking a verification orientation in writing HPC software, by moving to formal documentation in high-level specification of the algorithm, science, and assumptions in the source code, is that it provides a sensible approach to the process of moving the DOE software base forward to the new exascale era. This can address the barrier found with some existing DOE codes, that are so old that the original software designers are long gone, and the risk of breaking them is so high, that they become "untouchable." With an emphasis on verification we can move to a new domain of system productivity by producing longer- lived and richer software. This may be the solution to the dilemma where the massive DOE software base makes it costly or undesirable to take a clean sheet approach to exascale software. Rather than further embellishing the DOE software base with additional pragmas, unrolling, and platform-specific tuning that obfuscate the semantics and frustrate forward (manual AND automatic) portability, the software could instead be simplified and clarified with specification information that aids in the forward porting and translation of code, through automated means. If and as one takes on the task of updating and rewriting the DOE software base, the high-level approach offers the best long-term way to control life cycle costs.

5.1 Features of scientific code to exploit in verification

Scientific code has some unique features that facilitate validation. Scientific codes are based on strong mathematical theories, such as linear algebra, numerical algorithms, group theory, and linear systems. These features, available at many levels of scientific modeling, provide a rigorous basis for formulating and checking properties and invariants that can build confidence in an implementation of an algorithm on a particular computer instance, or to help build assurance into the quality of a code base.

Also, scientific code is based on the underlying scientific theories. In physics codes, there are properties such as energy conservation, reversibility, and symmetries that can be used in static and dynamic checks to gain confidence in the fidelity of a system. For QCD codes, for example, these symmetries provide the basis for checking the results of scientific computation, whether the results are in fact invariant under the symmetries required by the physics. If they are not invariant, there is a bug, somewhere – in an ALU, memory, bus, compiler, algorithm, etc. Such techniques are currently used manually to check QCD codes prior to and during long and expensive computing results to assure that the results are meaningful.

As another example, the DE Shaw DESMOND molecular dynamics simulation software implements molecular dynamics models in a perfectly reversible form, to the bit level using synthetic fixed point representations of values. It is possible to detect bugs in DESMOND by changing the sign of time, and running the simulation a few steps. The full reversibility of the underlying classical physical models, and the bit-perfect simulation, indicates that if the model does not reproduce its initial conditions when reversed, there is a bug somewhere. This property was invaluable in the coding of DESMOND, allowing it to be developed very rapidly.

A final aspect of scientific code is that the solution or simulation of a system can often be performed at multiple levels of abstraction, or with different kinds of models or algorithms, with reasonably well defined relationships between the bounds achieved by one level compared to another level. Thus simulations can be repeated or co-simulated at these varying levels of abstraction, or with related but different models, in order to detect variances and deviations that result from errors in the system or implementation.

Typically, currently, the validation of codes and scientific computations is done in an ad-hoc, application-specific manner. There is an opportunity for DOE R&D investment to develop technology, and culture of common and powerful tools and software engineering methodologies for scientific computing based on validation.

One should keep in mind the distinction between verification and validation. In embedded systems, verification is typically used to describe the checking of the correctness of a particular artifact with respect to the given specification of correctness. This is the sense that the question above seems to be focusing on when it asks about validation, but according to the common nomenclature it really should be using the term verification to be consistent with its intent. The term validation is used to refer to the process of determining whether the specification itself is sufficient for the application; does this application really prove anything about the scientific system that is under tests? (Stated another way, in the context of aerospace engineering, verification proves that the aircraft meets the specification of what it should do; validation checks the specification against the intended mission to assess if it will be useful when it works).

For both cases, verification and validation, the implication for software engineering is that the programming methodology for scientific computing must move strongly toward high-level, semantics-rich expressions. These expressions provide the referent – the formal expression of the specification, the symmetries, the invariants, and so forth, for automated and interactive tools to formally pose and answer questions about the correctness of exascale computing systems.

There is also a need for the software engineering methodologies to formalize the ad-hoc and seat-of-the-pants knowledge of the underlying mathematics and physics. This would be an ontology for enabling moving the knowledge and the prose in math and science domain papers and the even more fragile human know how into formalized bases that could be applied in automated tools across many applications and systems.

In making this recommendation, toward higher semantic expression of algorithm, scientific and mathematical knowledge, and the automation of verification tasks in scientific computing, we acknowledge that this is not necessarily easy. This is hard, but the payoff is incredibly large. We note that there will always be gaps in the ability to formalize this knowledge. In some cases the theories will be incomplete. For example, there is no universal theory (yet) on the right choice of the messy business of preconditioning systems for particular problem instances. But even in this case where the knowledge is heuristic, formalization of the knowledge will be useful. We can envision capturing the knowledge, such as it is, into expert systems associated with preconditioner design that increase the productivity of scientific computing experts and novices, speeding more rapidly to good results and eliminating time and energy consuming experimentation. The project of encapsulating mathematical knowledge in formalisms, developing the metalogical frameworks for integrating these logics, will require significant expertise. But recent work and successes in building proof based systems, integrating theories and logics, in the foundational proof carrying code area, is a promising foundation for application of these techniques in the scientific computing area.

5.2 Aleatory behavior and verification

An area in which demands for modeling for exascale exceeds current capabilities, and new technologies are needed, is in modeling the random or aleatory behavior of exascale machines that is expected to emerge as devices move closer to scaling limits, as circuits run closer to noise and power margins, and as tradeoffs such as power vs. unreliability reliability move in the direction of unreliability, with system level mechanisms providing for a reliable machine built of unreliable components. The underlying nondeterminism in exascale applications will also stem from unreliable components, and from "don't care nondeterminism" as in the Pingali University of Texas Galois system.

Transforming the unreliable system into a reliable one burns more power. Making the unreliable transistors into reliable transistors requires increasing the power supply voltage, which increases power dissipation per switching event, which might require clock frequency to diminish in order to meet operational constraints such as machine room power (and this will overall be at lower efficiency). Similarly, removing the opportunity for don't care nondeterminism in the software may reduce the concurrency of the system and limit the degree to which static power dissipation can be amortized and diminished by getting the result more quickly. Power and reliability will be linked for exascale. Research is needed in developing was of modeling this linkage.

Typically application programmers are willing to accept some performance overhead in development mode; they run in debugging mode or with thread checking on to detect latent bugs, race conditions and hazards. So in debugging mode we can envision a high tolerance for higher power, because the runs will be shorter and the priority is on software development productivity.

In production mode, there are probably ranges of tolerance for performance overhead in order to get bit reproducibility. In other words, it depends.

Fundamentally nondeterminism in the results (bit errors) makes programmers worry because it is hard to distinguish a bit error that is within the bounds of "don't care." No application really can be asked to tolerate outright bugs and violations of their application semantics. Therefore there should be low tolerance when the bit errors are signals of bugs or are undetected, or in violation of the specification of what degree of error will be allowed (this raises the question of how the specification is formally expressed to the system, or how a result could be checked for being within specification or not).

In some applications, such as optical proximity correction (an inverse computation to design a mask that compensates for diffraction in lithography in order to achieve a target physical pattern, in VLSI manufacturing), there is very low user tolerance for nondeterminism and error because a defect in the computation could lead to defective chips, costing millions of dollars. This was behind early resistance to using GPUs for OPC before GPUs had ECC. In contrast, OPC companies were willing to use IBM Cell chips because they did have ECC. This highlights the distinction between bit errors that are due to bugs (which are unacceptable), and bit errors that are due to something else, and thus acceptable. Because the underlying manufacturing process for masks has its own tolerance – masks are not being fabricated to more than a certain level of precision (1 in a million) so there is some level at which it truly doesn't matter whether the answer deviates by a smaller amount than this tolerance. It is just that the user has to have confidence (somehow) that this deviance is not due to a bug.

So in production mode, in principle, as long as the bit-level nondeterminism is somehow guaranteed to be within a certain level of tolerance based on the application, it seems like there should be a high willingness to accept this in order to get greater performance or computational efficiency. The engineering challenge is in designing the system to guarantee the conformance to a tolerance specification for bit-level nondeterminism. A starting point is the ability to express or write down the tolerance specification, in order to check and implement (by the programmer, compiler) the system to achieve it, and to verify that the implementation is correct.

This suggests that the modeling technology for exascale for describing resilience is related, somehow, to other facilities for reasoning about and proving properties related to the precision and accuracy of numerical computations. Such features may also be useful for reasoning and optimizing with respect to the representations, e.g., utilizing mixed precision in certain computations, which has recently been shown to be valuable by several library specialists, e.g., Dongarra. This is further related to uncertainty quantification and relates deeply to the properties of numerical algorithms themselves, e.g., in inverse problems where techniques such as regularization are needed to damp the sensitivity to small errors. Thus, there is a significant research and development challenge involved in providing assurance that bit-level variation in results is not an error but within some specified bounds.

5.3 Formal verification technologies relevant to DOE exascale

It is well established in the field of model checking for hardware design that system symmetries can be exploited to reduce the size of models. The early work in model checking of cache coherence protocols worked to reduce state explosions using symmetries. (See McMillin's 1992 thesis.) This does result in some loss of model fidelity (creating some false positives/negatives), but the resulting scalability may enable verification toward large parallel computers.

Certainly the foundation of formal verification can start with existing techniques used for verifying sequential and concurrent systems; model checking, temporal logics, SAT solvers, and more advanced technologies such as higher order logics, foundational proof carrying code, and proof assistants. These techniques can be applied directly to the verification of the envisioned exascale nondeterministic, asynchronous subsystems, e.g., new versions of MPI, dataflow-based systems such as ParalleX, or systems for irregular problems such as Galois. They can be used to verify buffer properties, active message protocols, resilience against message ordering variation, flow control, and other system mechanisms. Even though these are asynchronous and nondeterministic, potentially, in exascale, powerful applicable proof technologies exist. For example, foundational proof carrying codes provide methods for reasoning about asynchrony in complex forms; it is currently having success for reasoning about locking code implementations and uses for example, in operating systems.

The exascale programming language should provide for the specification of proofs to allow for the application of proof checkers/proof assistance in proof carrying code techniques.

Existing techniques in foundational proof carrying code provide ways to inject proofs, within the existing programming language syntax, into the advanced type system of the compiler, for generation to the proof objects that drive the small proof checkers. While these proofs would be constructed as part of the design process, they would ensure that forward generations using the software would have formalized explanations of the underlying theories of correctness and operation of the code to assist in having confidence (certification) in the correctness of the system in future compositions, ports, etc.

Reasoning about performance requires modeling, and model evaluations at other complementary levels to functional models (used for correctness). For example, reasoning about performance requires network models, processor models, and also algorithmic analysis to understand how those hardware elements affect performance. Ideally performance models would be expressed in a common framework with functional models, to share the schema for naming elements and for describing their relationship, aka machine models. Thus, while performance models seem to be something of a layer above the abstract machine, in a way this collapsing of layers occurs.

This may be related to Sterling's conception that the exascale execution model must be "vertical." That is the performance model, the machine model (which describes connectivity) and the functional model (which defines the behavior of the machine, e.g., and abstract machine) should be expressed in a common framework.

(Performance models also need means for describing their domain for correct application; e.g., a simple linear model of a network may be appropriate at nominal loads, but not accurate for predicting behavior under congestion.)

Other aspects of the machine model and abstraction machine that can aid with verification include exploring how the machine model can be dynamic (expressing changing conditions in the machine) or writeable (allowing for some forms of reconfiguration) and empirical (supporting implicit models based on performance profiling).

Working with such formalizations from systems software provides a degree of rigor in the design of AMMs and AMMLs. It helps move the definition of AMMs away from hand waving and prose into true formalizations with well defined underlying logics and semantics.

6 Reasoning about Architectures and Methods of Simulation

6.1 Reasoning about Architecture

Our ultimate goal is to design and build exascale hardware and software in order to solve significant problems that are important to our user communities. Abstract models are the spaces in which these designs live. Simulation uses these models to produce results that allow us to make design decisions.

It is impractical to attempt to evaluate every possible design in the entire design space. Instead any design process (including both hardware and software design) explores the design space in a tree-like manner starting out with designs at low levels of detail using very abstract models and progressing to successively more complex and less abstract models until ultimately the design is sufficiently detailed to allow the system to be built.

Each of these iterations involves major design decisions that need to be evaluated against our requirements. These are not only functional requirements (the system should be able to perform its intended task) but also a large number of generic performance requirements. Here we include not only the traditional concept of performance (time to solution of the calculation) but also any requirement on any aspect of the system that needs to be measured/estimated such as:

- Power
- Cost to manufacture
- Maintenance cost
- Cost of the remaining design effort
- Risk

This ability to evaluate our abstract designs and reason about their applicability is of vital importance to the whole design process. If we can identify a bad design during the early abstract phases then we can quickly eliminate large volumes of the design space from consideration. If the same problem is only identified much later in the process it will be necessary to back-track to an earlier and simpler version of the design (discarding much of the intermediate work) and start again down a different branch of the design tree. Therefore our abstract models need to contain models of the quantities of interest (power, cost, execution time etc.). This will naturally result in parameterised models where the parameters represent as-yet undetermined aspects of the design. For example at a particular level of abstraction we might choose to represent the power cost of a communication as some linear function of the rate bytes are being transferred and the distance of the communication. In the same model the corresponding hardware cost might be represented as being proportional to the wires or optical fibres being used. However the value of the constants of proportionality will not be known until further design decisions are made, such as the choice of optical or copper interconnects. As a result of this much of our design reasoning will have to take place using sets or ranges of possible parameter values or educated guesses that will need to be validated later in the process.

Though in some cases it will be possible to reason directly about the relative merits of different design choices, the scale and complexity of the systems we are trying to design mean that many of our design choices will be made based on simulation of the abstract models.

It should not be necessary to develop models for all parts of the design space. For example if we can, at an early stage, eliminate from consideration a class of hardware architectures using a very simple and abstract hardware model then there is no reason to go on and develop a more detailed and less abstract model for this class of architectures. Our models and to some extent our simulators are therefore part of the design process and are refined over time driven by the design decisions made earlier in the process.

Fortunately, at a sufficient abstract level the same models will be equally applicable to current systems as to the systems we are trying to develop. This gives us the opportunity to validate our initial models before progressing to the unique challenges presented by the Exascale. Unfortunately much work still needs to be done, particularly in identifying target application problem areas and developing abstract models that reflect their requirements.

6.2 Methods of Simulation

6.2.1 Machine Models

It is already common in hardware design for machines to be represented simultaneously or selectively at varying levels of abstraction. A design expressed in a HDL can be expressed at the functional level, register transfer level, gate level, transistor level, and even to some extent the physical level. Syntactic mechanisms and tools are common for expanding or flattening designs to desired degree of abstraction or multiple levels of abstraction. There is no mystery in the particular syntactic mechanism used to express the design; among Verilog, HDML, UML, EDIF, or XML, none is semantically that much more powerful than saying that the design is expressed in ASCII. What powers the modeling of architectures is the underlying logic that these models refer to. For simulation, this could be declarative, or imperative (event driven etc.)

Machine models for exascale computers will encompass a wide range of representations. This diversity is a product of the number of audiences that require models and the variety of design decisions they must address. However, all machine models will share one common feature – that they can be used to reason about the machine and produce useful decisions.

More than a single, static representation, "machine modeling" for the exascale is a multilevel, holistic, and iterative process. High-level abstract models will be used to develop low-level models, and as the community discovers new problems and bottlenecks, the models themselves will have to change. Due to our limited experience in designing and reasoning about exascale machines, the models cannot be created before we start reasoning about the system, but must be created and refined once we have some ideas about what the key and useful parameters are. This is true for any model, but for the exascale project, this concept of evolution and reinvention is critical. We face multiple moving targets – from the applications to the architectures and even the fundamental fabrication processes which construct the machines.

6.2.2 Standard Models

If machine model specifications were standardized at some level it would benefit simulation tasks by providing a uniform format to build simulators around and by making "apples-to-apples" comparisons of different machine models easier. Already, there are close to de facto standards for some computer components at some levels of simulation. For example, the processor core specifications used in the SimpleScalar or M5 simulators are generally accepted in the architecture community, and used in a variety of experiments. However, even these two standards are not interoperable and only describe one system component at one level of detail (processor cores at a cycle-approximate level).

When constructing exascale machine models and simulation tools, some level of standardization is desirable to achieve commonality and interoperability, but this must be balanced against the need for flexibility and evolution. These conflicting needs would rule out traditional standardization bodies (IEEE, JEDEC, etc…) as they are too slow to produce a standard in time.

6.2.3 Compiler Machine Models (Q3)

Most compilers contain different machine models to assist in optimizing code for different processor architectures. These machine models, like the GNU Compiler Machine Description, contain descriptions of the instructions available for a machine and how to convert a program's internal program representation into these instructions. This may include timing and performance details on these instructions to allow better optimization of the produced code. This is a low-level compiler machine model.

Though a useful example or starting point for a machine model in action, these low-level compiler machine models are incomplete. These models are very processor-centric, and ignore the network, IO, and memory aspects of the system. Additionally, they are very tied to a particular compiler, and to the needs of compilers (translating an intermediate representation into machine code) rather than a general model of machine performance or behavior.

The DARPA PCA program is one example of an effort to produce higher level machine models that include the higher level information. The Morphware Machine Model [4] has been used in advanced research and commercial software such as the R-Stream compiler. This machine model is expressed in an XML syntax with a schema that can express arbitrary levels of hierarchy, network parameters, heterogeneous complexes, "vertical" dimensions as referents for vertical placement through the hierarchy and horizontal dimensions across processing elements for horizontal placement in scheduling. Synchronization operations including hierarchical barriers can be expressed. The model also indicates varying execution models, e.g., allowing the expression of concurrency control as in OpenMP or as in CUDA, and also the combination of the two. Scratchpad memories and high-level DMA operations can be described, including the modes of concurrency and synchronization between processors and DMA engines.

While such existing models are not yet complete for exascale, using such a machine model as a starting point for exascale has the benefits of an existing scheme onto which new levels of abstraction and detail needed for exascale can be added.

6.2.4 Program Models

Since we require simulation with multiple levels of detail, it is necessary to represent programs at multiple levels of detail. That is, we need to build "program models" or "program representations" other than the compiled code of a program itself. This program model needs to "fit" with one or more machine models to be used in driving a simulation. Some examples of high-level models of programs would be "dynamic" scalable traces (such as Aspen), state machines (such as the SST/Macro skeleton apps), application statistics (such as the NMSU Stochastic Models), and generalized dependence graph (such as R-Stream).

Some key characteristics of these program models:

- **Adaptive:** The representations must be adaptive; static traces are insufficient because they cannot capture the causal relationships between system events. For example, a static network trace (when sends occurred) will miss the causal relationships between messages (e.g. request/response).

- **Scalable:** Many effects (congestion in networks, load imbalance between ranks, system noise) only become apparent at hundreds or thousands of nodes.
- **Comprehensive:** Local interactions can have global performance implications, so a program model should address as much of the system as possible. For example, changes in local memory access can have impact on global communication performance due to contention at the memory controller.
- **Reflective:** Program models should reflect aspects of the implementation, like the execution model. Decisions about how and where computation occurs will need to be represented, and ideally allowed to vary. This will require multiple models per problem to define the space of possible implementations.

The range of application surrogates (minimal representations of a larger application), such as Compact Apps, Mini-Apps, Skeleton Apps, and Proxy Apps, are good examples of high-level program models which may be easier for a simulator to use than a full application.

6.3 The Hardware/Software Interface

The overall behaviour of the system is a complex interaction between hardware, operating system, application code and the run-time. These therefore all need to be co-designed and evaluated together. It is worth noting however that software is inherently more flexible than hardware. In particular once the hardware has been built or is close to being built it is generally too expensive to redesign and rebuild this part of the system so any unforeseen problems or changes to requirements that occur after this stage would need to be accounted for in software. A key part of the design process will therefore be an iterative design of the interface between hardware and software.

To illustrate the problems in this area let us consider the vitally important communication sub-system. The basic set of functional requirements for the communication sub-system can be extracted from the set of operations needed to support our desired set of parallel programming models. A parallel programming model represents an aspect of the design space for a parallel application and the choice of parallel programming model is generally a very early design decision made during the development of an application. This set of required operations becomes an abstract run-time model and would contain operations such as the following:

- Point-to-point communications
- Collective reductions
- Barriers
- RDMA operations
- Active-message communications
- Etc.

In the early design stages it is important to keep this run-time model abstract and to resist the temptation to define a common run-time API too early. Adding detail to a design too early constrains the possible implementations of the operations. For example the tag and rank matching syntax in MPI is more specific than, and constrains the possible implementations of, the more abstract point-to-point operation.

In a more detailed design the communication sub-system would be implemented by a combination of hardware operations and a software run-time. This still leaves a large possible design space that needs exploring. For example point-to-point communications can be implemented using RDMA operations or active-messages. It is also possible to implement RDMA and active-messaging using a combination of point-to-point and threading though the performance behaviour would be totally different in this case.

Many of the programming models that are expected to be important for exascale machines rely on the concept of communication driven scheduling where the arrival of data can either cause the creation of a new thread of execution or a change to the runnable state of a suspended thread. This represents a potential performance problem as on current systems, thread scheduling is a concern of the operating system and entails relatively high delays and overheads. The impacts of these overheads needs to be explored with some urgency as implementing message driven scheduling in hardware would impact the design of both the communication hardware and the processor.

6.4 Methods of Simulation

Simulation will need multiple levels of detail to address the multiple audiences and classes of design decisions that exascale codesign will require. Most likely, there is close to a continuum of simulation methods and design experiments, ranging from back of the envelope estimation to hardware prototyping. A few useful simulation methods are in common use and are presented below.

Most likely, an exascale design project will use several of these models, with considerable feedback between high and low-level models and continual validation and evolution of the models themselves. The sheer scale of exascale systems mean that highly detailed simulations will only be able to simulate a small fraction of the total system, so any issues related to the global scale of the problem will have to be simulated at a lower level of detail.

6.4.1 Abstract Models

Abstract analytical models represent the machine in a closed form equation that takes gross machine characteristics (e.g., network bandwidth and latency, FLOPs) and application performance (number of operations) to determine performance. Examples would include the LogP or Postal models and even more abstract models such as the PRAM model.

These models benefit from simplicity and can provide quick results, allowing very fast exploration of a large design space. However, they can obscure important machine details.

6.4.2 High-level Simulation

High-level simulations represent the machine as an abstract automata or queuing model to simulate the machine's function. These abstract automata may be coupled with abstract equation based models to determine performance. For example, the SST/Macro simulator can represent node-level behavior as a state machine, and use a simple queuing model or LogP-like equation to model network performance.

These models provide more complexity than abstract models while still maintaining high performance. Additionally, they allow a mechanism for application or algorithm writers to explore different high-level organizations of their program and see the effects on a given architecture. For example, changing the state machine could allow quick exploration of the effects of different communication patterns. These models can reveal more detail of the machine, but still run the risk of obscuring important aspects of the architecture.

6.4.3 Medium Level Simulation

Medium level simulations combine detailed low-level (cycle accurate or cycle approximate) simulation of some components with more abstracted simulation of other components. For example the SST simulator can perform simulation with the detailed M5 processor simulator connected to a simple network and memory simulator, or detailed network simulation with the Red Storm Router model connected to a simplified state-machine node model.

This method of simulation allows the designer to focus in on one aspect of the machine design while avoiding paying the performance cost of fully detailed simulation. However, this may neglect feedback effects (e.g., memory and network interactions) and leads to uncertainty in how simulation errors are computed.

6.4.4 Low-level Simulation

Low-level simulation involves detailed models of all major components in the system (usually the processor, network, memory, and IO system). Each of these models is generally cycle-accurate or cycle-approximate.

These simulations have the benefit of detail, but often require lengthy simulation runs and considerable complexity to build and validate detailed simulation models.

6.4.5 Mixed Simulation

Mixed simulation combines software simulation with extremely detailed hardware simulation, such as on an FPGA. The FAST project is an example of this technique.

This simulation method potentially combines the flexibility of software simulation with the speed and detail of hardware prototyping. However, it can also combine the inflexibility of hardware simulation with the inaccuracy and slowness of software simulation.

6.4.6 Prototyping

Hardware prototyping can be used to run experiments of systems at less than full scale, or to explore a new component technology. This can be used for architectural exploration (e.g., exploring the capabilities of a new network card) or to explore lower level fabrication issues (e.g., a new packaging technique). Hardware prototyping can use FPGAs, custom boards, or the full design and production of a custom ASIC.

This level of simulation provides extremely detailed information on component performance, but has the drawbacks of high cost and low flexibility.

7 What Needs To Be Done

Creating a series of machine models and simulation tools which can address the novel requirements of exascale, have some level of standardization and interoperability, and are still flexible and can be evolved will require an intensive research program and later, for adoption coordinated community effort.

7.1 Fundamental Research and Development

The sections above have detailed several areas in which there is a gap between current practice for HPC machine modeling, and what is needed for exascale, exist. This gap should be addressed by the following:

- **Best Known Practices:** There are several areas where there are new technologies emerging from other areas of computer science that should be adopted immediately. For modeling, there are techniques for formal verification and hybrid modeling, that can be adopted to increase the rigor and power of modeling efforts., for modeling complex dynamic nondeterministic and asynchronous processes. The DARPA PCA program, and subsequent follow on development, provides languages and schema for modeling large heterogeneous, hierarchical, accelerated, computing devices. The field of embedded computing provides tools for modeling and verifying real time and other hard performance constraints in complex distributed systems with adaptive control, which may be useful in assuring the operation of even safety of exascale computing systems.
- **Focused Modeling Research:** The report above has identified several areas where existing practice may not be sufficient to meet the needs of extreme scale. These include areas addressing the scale of extreme scale and modeling new considerations such as aleatory behavior and greater precision of communications. Co-simulation and modeling of multiple levels of abstraction to address the cross cutting nature of execution models across all levels of the system will be required. New techniques for exploiting symmetries and other acceleration as well as non-simulation automatic evaluation of large models will be required. Understanding of the degree to which emergent behaviors will occur and can be controlled, in network, processors, and in network-processor-software complexes at extreme scale, are needed.
- **Formalize Existing Ad-Hoc practice:** techniques that are already used within the HPC field, but in informal, ad-hoc, or in prosaic manners, should be formalized and incorporated into modeling approaches. These include efforts to formalize the science and mathematics of high-performance computing into knowledge ontologies that can be used by automated tools for verification and optimization of systems. These knowledge ontologies can provide leverage to the practice of HPC software development to increase productivity, scalability, and generality of software and to reduce overall DOE software lifecycle costs.
- **Build Cross Cutting Communities:** the technologies of extreme scale architecture modeling relate to efforts in extreme component design, system design, software design, system verification, and operation. Communities must be formed that can exploit synergies across these communities through new modes and types of

sharing of models. New linkages, such as e.g., exporting models and specifications from application programs and required precision to the verification of VLSI chips and novel floating point representations, should be formed. These communities should span multiple institution types, including DOE laboratories, commercial software and hardware vendors, and academics. The community for sharing these models may well span international boundaries.

7.2 Developing Standards

A plan for building standard machine models requires substantial community involvement and an ongoing effort to refine and evolve the standards. This effort must build on existing knowledge and experience without limiting the design space. Possible steps in this plan:

- **Identify Audiences:** The consumers of machine models and simulations must be clearly identified and their needs/requirements (the questions they will ask) understood. Already, highly divergent audiences (Aarons, Stephanies, Joes, others?) are known to exist.
- **Survey:** A survey of existing simulators to identify commonalities and opportunities for interoperation must be performed. This will allow a gap analysis as well as an understanding of existing standards.
- **Standardize:** Creation of an initial standard format. Initially, this will tend to represent a "most common denominator" of existing simulation models, but should be constructed to not overly constrain the models. Additionally, there will be multiple levels of detail in these machine models.
- **Evolve:** The standard will be allowed to evolve as needed. This will be required to capture specialized capabilities and to encompass new issues, technologies, and opportunities as they arise. During this evolution process we must be aware that we are guided by what we think we can use today, and care must be taken to avoid ignoring other ideas due to lack of existing tools.
- **Validate:** In parallel with the evolution of the standard, we must constantly validate models against current systems and against other levels of model to ensure our results can be trusted.

7.3 Standard representation

Some degree of standardization of our models is highly desirable:

- A common way of representing application models will allow many different user communities to develop models that can be evaluated against different machine models using common simulation infrastructures.
- Common ways of representing machine models will allow the results of the same models to be compared using different simulation environments.

As the basic requirement is for portability of models the same benefits could be achieved by implementing tools to translate between formats. However as the models become more detailed the syntax needed to describe a design will quickly become very complex and automatic translation a very difficult problem in its own right. For example even a fairly abstract application model will need something not dissimilar to a programming model to describe it.

Significant questions exist about the nature of these models. Some level of consensus will need to be formed within the community, or at least within the tool and simulation builders:

- **Format:** Should the standard model specification be a detailed file format (e.g., an XML schema) or simply a list of key parameters that must be specified for a given level of simulation.
- **Trust:** How do we trust the models, and what level of trust do we need? The answers to these questions will be time and audience dependent. Transparency about how the models are constructed and what they contain will help build trust. Similarly, the use of formal verification methods will also help acceptance of the models' results. However, we must also recognize, and be comfortable with, substantial error bars in the simulations, especially with high-level abstract simulation.
- **Validation:** What HW/SW mechanisms do we need to validate the models? How can we design systems today which can be more efficiently validated for performance? Should we be limited future systems to ones which we can clearly understand and predict performance? For example, the real-time embedded community has consistently chosen predictability over raw performance to be able to make performance guarantees.

7.4 How to encourage adoption

Creating standards is useless without adoption. There are many things that can be used to encourage adoption of standards and promote interoperability:

- **Build Trust:** Standard models and simulation frameworks will only be adopted if the community trusts them. This includes validation and also a clear understanding of the limits of different simulation methods.
- **Identify clear needs/benefits:** Unless the standards have a clearly identified benefit, they will be viewed as simply another distraction and an unwanted bureaucratic mandate. The benefits of interoperability must be clearly shown, and we must recognize that different audiences will place different priority on interoperability. Adoption of the standards and interoperability should be encouraged where there is need, not simply for its own sake.
- **Funding:** Even if standards are trusted and a need is identified, adoption can be slow due to organizational factors, effort required to adopt, and "not invented here" syndrome. In these cases, a "big stick" approach could be considered, where interoperability is made a prerequisite for receiving funding.

8 Conclusion

The scale and complexity of exascale computing systems, the requirements of managing power and resilience, and the development and porting of software, present many challenges that cut across all levels of exascale design. Abstract modeling of machines is identified as a key enabling and operational tool for achieving extreme scale computing, from device development up to applications programming, and from design through production through operation. This report has described the role of modeling, and in particular how modeling for exascale must move beyond being a tool for just simulating new architectures, but one for communicating among all participants in the extreme scale development – the hardware, systems software, and applications developers and vendors. Modeling has been identified as a tool for managing, reducing, and controlling exascale life cycle costs, in particular by rationalizing and formalizing the software development methodology.

To achieve these high-payoff and program risk reduction goals, new exascale abstract modeling research and development is required to fill gaps from adopted existing best practices. These research goals are identified in this report. There may also be cultural changes needed within the HPC community to make modeling a first class citizen in HPC design, given its critical role. These considerations – the need to fill gaps in technology, and cultural changes to build new channels for exploiting the benefits of modeling, suggest that a standalone and concerted research program be established in exascale modeling that can nurture the academics, commercial and laboratory community in this area.

9 Works Cited

[1] Peter M. Kogge (editor), "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," Univ. of Notre Dame, CSE Dept. Tech. Report TR-2008-13, Sept. 28, 2008

[2] Automatic Recognition of Performance Idioms in Scientific Applications, J. He, A. Snavely, R. F. Van der Wijngaart, M. Frumkin, 25th IEEE International Parallel & Distributed Processing Symposium, 2011.

[3] http://www.orau.gov/archII2011/presentations/snir_m.pdf

[4] http://www.morphware.org/PCA101/PCA%20101%201.0%20022104.pdf