COMPUTER
ARCHITECTURE
LABORATORY
EXASCALE DESIGN SPACE EXPLORATION

Abstract Machine Models and Proxy Architectures for Exascale Computing

Rev 1.1

J.A. Ang[1], R.F. Barrett[1], R.E. Benner[1], D. Burke[2],
C. Chan[2], D. Donofrio[2], S.D. Hammond[1],
K.S. Hemmert[1], S.M. Kelly[1], H. Le[1], V.J. Leung[1],
D.R. Resnick[1], A.F. Rodrigues[1],
J. Shalf[2], D. Stark[1], D. Unat[2], N.J. Wright[2]

Sandia National Laboratories, NM[1]
Lawrence Berkeley National Laboratory, CA[2]

May, 16 2014

Abstract Machine Models and Proxy Architectures for Exascale Computing

J.A. Ang[1], R.F. Barrett[1], R.E. Benner[1], D. Burke[2],
C. Chan[2], D. Donofrio[2], S.D. Hammond[1],
K.S. Hemmert[1], S.M. Kelly[1], H. Le[1], V.J. Leung[1],
D.R. Resnick[1], A.F. Rodrigues[1],
J. Shalf[2], D. Stark[1], D. Unat[2], N.J. Wright[2]

Sandia National Laboratories[1]
Albuquerque, New Mexico, USA

Lawrence Berkeley National Laboratory[2]
Berkeley, California, USA

To achieve exascale computing, fundamental hardware architectures must change. The most significant consequence of this assertion is the impact on the scientific applications that run on current high performance computing (HPC) systems, many of which codify years of scientific domain knowledge and refinements for contemporary computer systems. In order to adapt to exascale architectures, developers must be able to reason about new hardware and determine what programming models and algorithms will provide the best blend of performance and energy efficiency into the future. While many details of the exascale architectures are undefined, an *abstract machine model* is designed to allow application developers to focus on the aspects of the machine that are important or relevant to performance and code structure. These models are intended as communication aids between application developers and hardware architects during the co-design process. We use the term *proxy architecture* to describe a parameterized version of an abstract machine model, with the parameters added to elucidate potential speeds and capacities of key hardware components. These more detailed architectural models are formulated to enable discussion between the developers of analytic models and simulators and computer hardware architects. They allow for application performance analysis and hardware optimization opportunities. In this report our goal is to provide the application development community with a set of models that can help software developers prepare for exascale. In addition, use of proxy architectures, through the use of proxy architectures, we can enable a more concrete exploration of how well application codes map onto the future architectures.

Acknowledgment

Support for this work was provided by the Advanced Scientific Computing Research (ASCR) program and funded by the Director, Office of Science, of the U.S. Department of Energy. Lawrence Berkeley National Laboratory operates under Contract No. DE-AC02-05CH11231. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Thanks to Prof. Bruce Jacob of the University of Maryland for input on NVRAM trends and to Jeanine Cook and Mike Levenhagen for their participation in some of our discussions.

Contents

Chapter 1

Introduction

In this report we present an alternative view of industry's exascale system hardware architectures. Instead of providing highly detailed models of each potential architecture, as may be presented by any individual processor vendor, we propose initially to utilize simpler, abstract models of a compute node that allow an application developer to reason about data structure placement in the memory system and the location at which computational kernels may be run. This abstract machine model (AMM) will provide software developers with sufficient detail of gross architectural features so they may begin tailoring their codes for these new high performance machines and avoid pitfalls when creating new codes or porting existing codes to exascale machines. While more accurate models will permit greater optimization to the specific hardware, it is our view that a more general approach will address the more pressing issue of initial application porting and algorithm re-development that will be required for future computing systems. Once initial ports and algorithms have been formulated, further refinements on the models in this document can be used as a vehicle to optimize the application. These models offer the following benefits to the research community:

Simplified model Abstract models focus on the important high-level hardware components, which in turn affect code structure and algorithm performance – implementation specific details are omitted.

Enable community engagement Abstract machines are an important means of communicating to application developers about the nature of future computing systems so they can reason about how to restructure their codes and algorithms for those machines.

Enable design space exploration The AMM is the formalization of a particular parametric model for a class of machines that expresses the design space and what we value. Additionally, the purpose of each of the presented models is to abstractly represent many vendor specific hardware solutions, allowing the application developer to target multiple instantiations of the architecture with a single, high-level logical view of the machine.

Enable programming systems development A high-level representation of the machine also enables design of automated methods (runtime or compile time) to efficiently map an algorithm onto the underlying machine architecture.

In the face of a large number of hardware design constraints, industry has proposed solutions that cover a variety of possibilities. These solutions range from systems optimized for many ultra-low power processor cores executing at vast scale to achieve high aggregate performance throughput to large, powerful processor sockets that demand smaller scales but provide performance at much higher per-node power consumption. Each of these designs blends a set of novel technologies to address the challenges laid out. To assist developers in reasoning about the disparate ideas and solutions, in Chapter 2 we will present an overarching abstract model designed to capture many of the proposed ideas into a single, unified view. Then we present a family of abstracted models that reflect the range more specific architectural directions being pursued by contemporary CPU designers. Each of these models are presented in sufficient detail to support many uses, from application developers becoming familiar with the initial porting of their applications through to hardware designers exploring the potential capabilities of future computing devices.

Next, in Chapter 3, we discuss memory architectures. An example of how an abstract model may be applied to assist users in their transition from current machines can be seen in the memory systems of future machines. It is likely that due to a rising number of cores per socket the memory hierarchy will become further subdivided to maintain a reasonable amount of memory available per core. This subdivision will make trade-offs of capacity versus bandwidth at different levels, forcing programmers to manage vertical locality more explicitly than currently required. In addition, maintaining cache coherence constitutes a large percentage of on-chip data traffic. Future designs are leaning toward only maintaining cache coherence among only a subset of cores, forcing programmers to manage data consistency through explicit programing constructs.

In Chapter 4 we depart from a model-centric view and discuss issues to be considered from a programming perspective when studying the viability of each model. We provide some basic background information on how various hardware features will impact programming and ultimately application performance. As the HPC community drives toward achieving exascale, new metrics of energy efficiency, increased concurrency, programmability, resilience, and data locality will play an increasing role in determining which hardware solutions are feasible and practical to utilize for production-class *in-silico* scientific research. From the application programmers' perspective, programming to each potential exascale hardware solution presents an unwelcome situation in which multiple rewrites of the application source may be required – in some cases demanding radical shifts in data structure layout or the re-design of key computational kernels. Put simply, many supercomputing centers will be unable to afford the luxury of frequent application rewrites either due to the sheer cost of such an endeavor or the number and availability of programmers needed to undertake the activity.

Chapter 5 presents parameterized instantiations, or *proxy architectures*, of some of the machine models outlined in this report. Proxy architectures are an especially useful communication vehicle between hardware architects and performance analysts. Models can be developed based on the AMM descriptions and the parameter space identified in the associated proxy architecture. When a sufficient range of parameters is applied, we foresee that the models may be shared openly between users, academics, and researchers. A specific parameter set that closely relates to a point design will likely be proprietary and, therefore, only shared within the appropriate disclosure constraints. Simulations using these models will explore the ever-expanding definition of performance. In addition to the Holy Grail of minimizing application run times, power usage and data movement and maximizing resiliency, programmer productivity are also parts of the equation that must be considered in identifying usable exascale systems.

Finally, we conclude our report in Chapter 6 and add a discussion of future work and important codesign and programmatic interfaces for other research and development areas in the overarching Department of Energy (DOE) exascale program.

Chapter 2

Abstract Machine Models for Algorithm Design

In order to sufficiently reason about application and algorithm development on future exascale-class compute nodes, a suitable AMM [9] is required so that algorithms can be developed independent of specific hardware parameters. It is useful to think of the AMM as a way to simplify the myriad complex choices required to target a real machine and as a model in which application developers can frame their algorithms [10]. The AMM represents the subset of machine attributes that will be important for code performance, enabling us to reason about power/performance trade-offs for different algorithm and execution model choices. We want an abstract model of the underlying hardware to be as simple as possible to focus on the durable cross-cutting abstractions that are apparent across machines of the same generation, and to represent long-term trends for future generations of computing systems. While there exist many potential machine attributes that could be included in our model, we instead take one of three actions to concentrate our models into more concise units:

- **Ignore it.** If ignoring the design choice has no significant consequences for the consumption of power or the provision of computational performance we choose to eliminate the feature from the model. We include architecture-specific instantiations of hardware features, such as specific single-instruction, multiple-data (SIMD)-vector widths, in this category since it is the presence of the functional unit that is important for reasoning with the model – not the specific capabilities of the functional unit itself. Such details are provided in the *proxy architecture* annotation of the model.
- **Abstract it.** If the specific details of the hardware design choice are well enough understood to provide an automated mechanism to optimize a layout or schedule, an abstracted form of the choice is made available (for example, register allocation has been successfully virtualized by modern compilers).
- **Expose it.** If there is no clear mechanism to automate decisions but there is a compelling need to include a hardware choice, we explicitly expose it in our abstract machine model deferring the decision of how best to utilize the hardware to the application programmer. For instance, the inclusion of multiple types of memory will require specific data structure placement by the programmer, which in turn implies a need for the programming model to also support data placement.

For the models that follow in this section, it is important to note that we do not fully describe the coherency aspects of the various memory subsystems. These memory systems will likely differ from current coherency schemes and may be non-coherent software-based coherent, or hardware-supported coherent. Similarly, we describe the basic node compute infrastructure with a view that the interconnect associated with the node is an orthogonal aspect. To be clear, each of these design dimensions – memory and network interface – are orthogonal in the design of an exascale machine, which will be addressed in further additions to our basic in-node model. We expect network interfaces to be integrated into the processor, but we leave the specifics of the network topology to further increments of our models since, in current practice, very few algorithms are designed or optimized to a specific network topology.

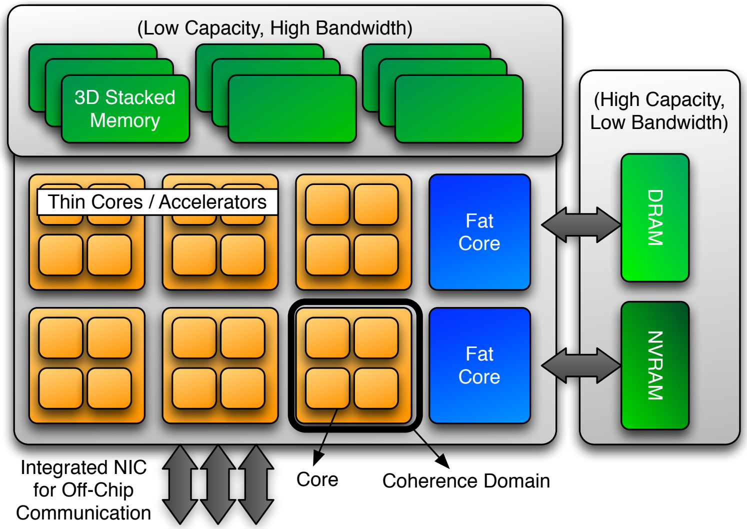


Figure 2.1: Abstract Machine Model of an exascale Node Architecture

2.1 Overarching Abstract Machine Model

We begin with a single model that highlights the anticipated key hardware architectural features that may support exascale computing. Figure 2.1 pictorially presents this as a single model, while the next subsections describe several emerging technology themes that characterize more specific hardware design choices by commercial vendors. In Section 2.2, we describe the most plausible set of realizations of the single model that are viable candidates for future supercomputing architectures.

2.1.1 Processor

It is likely that future exascale machines will feature heterogeneous nodes composed of a collection of more than a single type of processing element. The so-called *fat* cores that are found in many contemporary desktop and server processors characterized by deep pipelines, multiple levels of the memory hierarchy, instruction-level parallelism and other architectural features that prioritize serial performance and tolerate expensive memory accesses. This class of core is often optimized to run a small number of hardware threads with an emphasis on efficient execution of system services, system runtime, or an operating system.

The alternative type of core that we expect to see in future processors is a *thin* core that features a less complex design in order to use less power and physical die space. By utilizing a much higher count of the *thinner* cores a processor will be able to provide high performance if a greater degree of parallelism is available in the algorithm being executed.

Application programmers will therefore need to consider the uses of each class of core; a *fat* core will provide the highest performance and energy efficiency for algorithms where little parallelism is available or the code features complex branching schemes leading to thread divergence, while a *thin* core will provide the highest aggregate processor performance and energy efficiency where parallelism can be exploited, branching is minimized and memory access patterns are coalesced.

2.1.2 On-Chip Memory

The need for more memory capacity and bandwidth is pushing node architectures to provide larger memories on or integrated into CPU packages. This memory can be formulated as a cache if it is fast enough or, alternatively, can be a new level of the memory system architecture. Additionally, scratchpad memories (SPMs) are an alternate way for cache to ensure a low latency access to data. SPMs have been shown to be more energy-efficient, have faster access time, and take up less area than traditional hardware cache [14]. Going forward, on-chip SPMs will be more prevalent and programmers will be able to configure the on-chip memory as cache and/or scratchpad memory, allowing initial legacy runs of an application to utilize a cache-only configuration while application variants using scratchpad-memory are developed.

2.1.3 Cache Locality/Topology

A fundamental difference from today's processor/node architecture will be the loss of conventional approaches to processor-wide hardware cache coherence. This will be driven heavily by the higher power consumption required with increased parallelism and the greater expense in time required to check the increased function unit count for cached copies of data. A number of existing studies provide a description of the challenges and costs associated with this exercise: Schuchhardt et al. [14] and Kaxiras and Keramidas [8] provide quantitative evidence that cache coherency creates substantial additional on-chip traffic and suggest forms of hierarchical or dynamic directories to reduce traffic, but these approaches have limited scalability. Furthermore, Xu [20] finds that hierarchical caching doesn't improve the probability of finding a cache line locally as much as one would hope – a conclusion also supported by a completely independent study by Ros et al. [12] that found conventional hardware coherence created too much long-distance communication (easily a problem for the scalability of future chips). We find that considerable recent work has followed along the lines of Choi et al.'s 2011 DeNovo approach [3], which argues that hybrid protocols, including self-invalidation of cache and flexible cache partitions, are better ideas and show some large improvements compared to hardware cache coherency with 64 cores and above.

Fast Forward is DOE's advanced technology development program (http://www.exascaleinitiative.org). Due to the strong evidence in the literature and numerous independent architectural studies, we and a number of the Fast Forward vendors believe there is ample evidence that continuing to scale current hardware coherence protocols to manycore chips will come at a severe cost of power, performance and complexity. Many of the Fast Forward vendors have therefore adopted various forms of hierarchical coherence models (termed as *islands*, *coherence domains* or *coherence regions* depending on the vendor implementation), which are reflected in our AMM diagram under the unified moniker of "*coherence domains*". It is likely that the fat cores will retain the familiar automatically managed memories now familiar to developers, but scaling up coherence across hundreds or thousands of thin cores now seems unlikely. In the best case, the thin cores may be grouped into several *coherence domains* (as shown in Figure 2.1) that allow for automatic management, but the programmer will be responsible for explicitly moving data between incoherent domains. It is also just as likely that there may be almost no automatic management of memory for these thin cores leaving the full burden on the developer. Besides programming difficulties, regional coherence leads to varying access latencies. Thus it is performance-critical for application software to be aware of the topology somewhere in the software stack.

Current multicore processors are connected in a relatively simple all-to-all or ring network. As core counts surpass the dozen or so cores we see on current processors these networks will cease to scale and will give rise to more sophisticated network topologies. Unlike the current on-chip networks, these networks will stress the importance of locality and force the programmer to be aware of where data is located on-chip to achieve optimal performance.

2.1.4 Integrated Components

The network interface controller (NIC) is the gateway from the node to the system level network, and the NIC architecture can have a large impact on the efficiency with which communication models can be implemented. For large parallel systems, the inter-node network is the dominant factor in determining how well an application will scale. Even at small scale, applications can spend a large portion of their time waiting for messages to arrive and reductions in bandwidth or failure to substantially improve latency over conventional methods can greatly exacerbate this problem. A custom NIC that integrates the network controller, and in some cases the messaging protocol, onto the chip to reduce power, is expected to also increase messaging throughput and communication performance [2, 17]. Although there is a risk of choosing a network interface that is not compatible with all the underlying data communication layers, applications that send small and frequent messages are likely to benefit from such integration.

2.1.5 Hardware Performance Heterogeneity

One important aspect of the AMM that is not directly reflected in the schematic of the AMM in Figure 2.1 is the potential for non-uniform execution rates across the many billions of computing elements in an exascale system.

This *performance heterogeneity* will be manifested from chip level all the way up to system-level. This aspect of the AMM is important because the HPC community has evolved a parallel computing infrastructure that is largely optimized for bulk-synchronous execution models. It implicitly assumes that every processing element is identical and operates at the same performance. However, a number of sources of performance heterogeneity may break the assumptions of uniformity that underpin our current bulk-synchronous models. Since the most energy-efficient floating point operations (FLOPs) is the one you do not perform, there is increased interest in using adaptive and irregular algorithms to apply computation only where it is required, and also to reduce memory requirements. Even for systems with homogeneous computation on homogeneous cores, new fine-grained power management makes homogeneous cores look heterogeneous [15]. For example thermal throttling on Intel Sandy Bridge enables the core to opportunistically sprint to a higher clock frequency until it gets too hot, but the implementation cannot guarantee deterministic clock rate because chips heat up at different rates. Options for active (software mediated) power management might also create sources of performance non-uniformity [4]. In the future, non-uniformities in process technology and near-threshold-voltage for ultra-low-power logic will create non-uniform operating characteristics for cores on a chip multiprocessor [7]. Fault resilience will also introduce inhomogeneity in execution rates as even hardware error correction is not instantaneous, and software-based resilience will introduce even larger performance heterogeneity [19].

Therefore even homogeneous hardware will look increasingly heterogeneous in future technology generations. Consequently, we can no longer depend on homogeneity, which presents an existential challenge to bulk-synchronous execution models.

2.2 Abstract Model Instantiations

Given the overarching model, we now highlight and expand upon key elements that will make a difference in application performance.

2.2.1 Homogeneous Many-core Processor Model

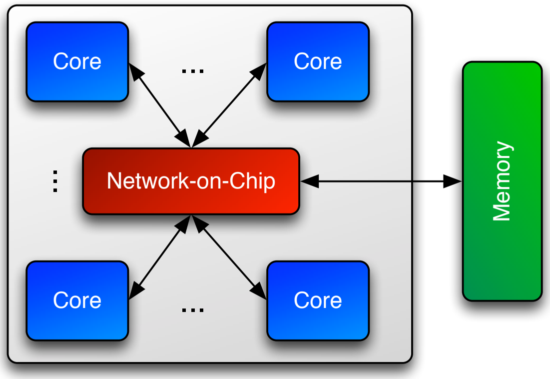


Figure 2.2: Homogeneous Manycore Model

In a homogeneous manycore node (Figure 2.2) a series of processor cores are connected via an on-chip network. Each core is symmetric in its performance capabilities and has an identical instruction set (ISA). The cores share a single address memory space and may have small, fast, local caches that operate with full coherency. We expect that the trend of creating individual clock and voltage domains on a per-core basis will continue allowing an application developer or system runtime to individually set performance or energy consumption limits on a per-core basis meaning that variability in runtime will be present on a per-core basis, not because of differences in the capabilities of each core but because of dynamic configuration. Optionally, the cores may implement several additional features depending on the performance targets including simultaneous multithreading (SMT), instruction level parallelism (ILP), out-of-order instruction execution or SIMD short-vector units.

Like a system-area interconnect, the on-chip network may "taper" and vary depending on the core pair and network topology. Similarly, the programmer will have to contend with network congestion and latency.

Depending on the programming model, communication may be explicit or largely implicit (e.g. coherency traffic).

2.2.2 Multicore CPU with Discrete Accelerators Model

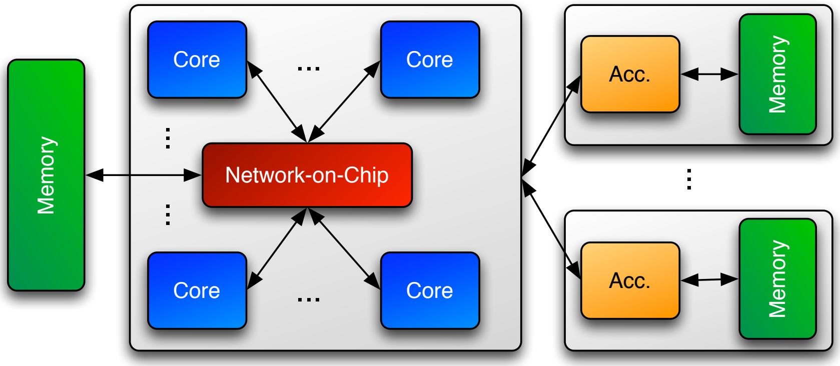


Figure 2.3: Multicore CPU + Discrete Accelerators Model (Acc: Accelerator)

In this model a homogeneous multi-core processor (Figure 2.3) is coupled with a series of discrete accelerators. The processor contains a set of homogeneous cores with symmetric processor capabilities that are connected with an on-chip network. Each core may optionally utilize multi-threading capabilities, on-core caches and per-core based power/frequency scaling. Each discrete accelerator is located in a separate device and features an accelerator processor that may be thought of as a throughput oriented core with vector processing capabilities. The accelerator has a local, high performance memory, which is physically separate from the main processor memory subsystem. To take advantage of the entire compute capability of the processor, the programmer has to utilize the accelerator cores, and the programming model may have to be accelerator-aware. This model is seen in existing DOE systems such as the OLCF Titan and LANL Roadrunner systems, but it represents what we consider an obsolete approach to acceleration that has been superseded by the integrated multicore model described in subsequent sections. We have not seen competitive bids for systems based on this approach for recent DOE procurements, nor is such an architecture desirable for future DOE system acquisitions.

2.2.3 Integrated CPU and Accelerators Model

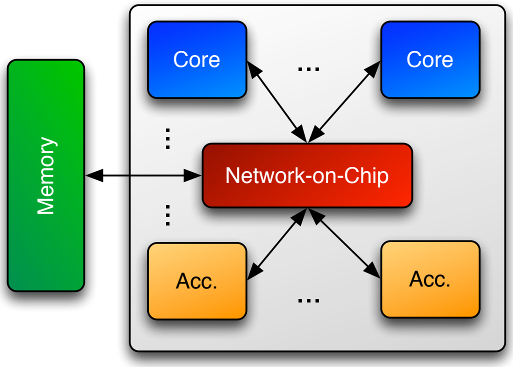


Figure 2.4: Integrated CPU + Accelerators Model (Acc: Accelerator)

An integrated processor and accelerator model (Figure 2.4) combines potentially many latency-optimized processor CPU cores with many accelerators in a single physical die, allowing for potential optimization to be added to the architecture for accelerator offloading. The important differentiating aspect of this model is a shared, single coherent memory address space is accessed through shared on-chip memory controllers. While this integration will greatly simplify the programming, latency optimized processors and accelerators will compete for the memory bandwidth.

2.2.4 Heterogeneous Multicore Model

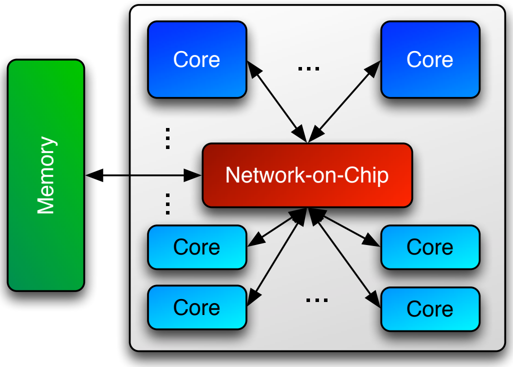


Figure 2.5: Heterogeneous Multicore Model

A heterogeneous multi-core architecture features potentially many different classes of processor cores integrated into a single die. All processor cores are connected via an on-chip network and share a single, coherent address space operated by a set of shared memory controllers. We envision that the cores may differ in ISA, performance capabilities, and design, with the core designers selecting a blend of multi-threading, on-chip cache structures, short SIMD vector operations, instruction-level parallelism and out-of-order/in-order execution. Thus, application performance on this architecture model will require exploiting different types and levels of parallelism. Figure 2.5 provides an overview image of this design is shown for two classes of processor cores.

The main difference between the heterogeneous multi-core model and the previously discussed integrated multi-core CPU and accelerator model (Section 2.2.3) is one of programming concerns: in the heterogeneous multi-core model each processing element is an independent processor core that can support complex branching and independent threaded, process or task-based execution. In the integrated multi-core with accelerator model, the accelerators will pay a higher performance cost for heavily divergent branching conditions and will require algorithms to be written for them using data-parallel techniques. While the distinction may appear subtle, these differences in basic hardware design will lead to significant variation in application performance and energy consumption profiles depending on the types of algorithm being executed, motivating the construction of two separate AMMs.

2.3 Abstract Models for Concept Exascale Architectures

The machine models presented in Section 2.2 represent relatively conservative predictions based on known vendor roadmaps and industry trends. However, with the advent of system on chip (SoC) design, future machines can be optimized to support our specific requirements and offer new methods of implementing on-chip memories, change how coherency and data sharing are done and implement new ways to support system latencies. This creates a much wider design space. In this section we present one possible concept for a customized system node.

2.3.1 Performance-Flexible Multicore-Accelerator-Memory Model

The homogeneous multicore-accelerator-memory (MAM) model is an aggressive design for a future processor based around new approaches to make general computation more efficient. The focus in this design is on achieving higher percentages of peak performance by supporting execution through a greater variety of specialized function units and multi-threaded parallel execution within each core. The processor features many heterogeneous cores, a hierarchical internal network and an internal NIC with multiple network connections, allowing

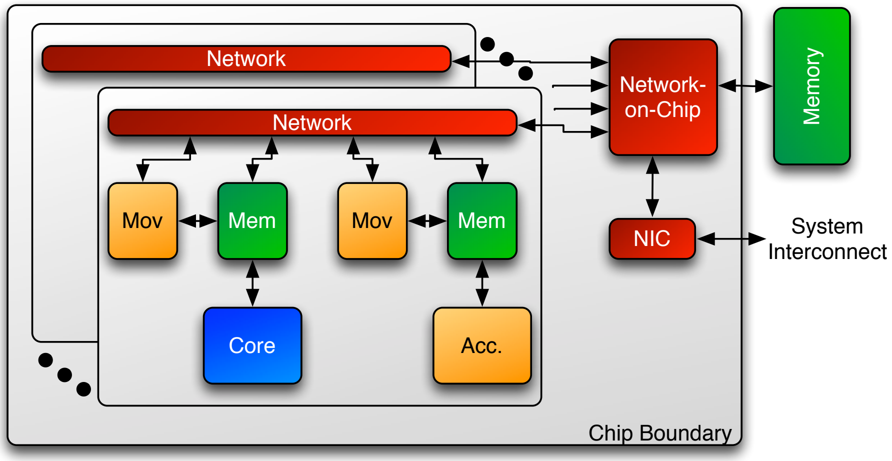


Figure 2.6: Homogeneous Multicore-Accelerator-Memory Model (Mem: Memory, Acc: Accelerator, Mov: Data Movement Engine)

multiple memory channels and extremely high internal and external access to local and remote processor memories throughout the system. There are multiple components in the memory system: many internal memory blocks that are integrated into the CPU cores as well as main memory that is directly connected to nodes and is available to other nodes through a system's network.

Each core implements a high order number of threads to hide latency and to enable threads to be of variable performance. Multiple threads in a core run at the same time and the implementation is such that the core is kept busy; for example if a thread is held waiting for a main-memory item, another thread is put into execution.

As stated above, each thread in a core is given a portion of local on-chip memory. That memory is originally private to each thread, though each core can choose to share portions of its space with other threads. Each portion has multiple pieces such that some pieces can be for caching and others for "*scratch space*" at the same time. Scratch space is memory that is used to store intermediate results and data placed there is not stored in main memory. This saves energy and reduces memory traffic. (Scratch data is saved if a job is rolled out.)

When a portion of its local, cached memory is shared with other threads, the sharing entities see only a single cache for that specific portion of the address space. This greatly reduces coherency issues. If the shared portion of local memory is scratch space, there is only a single copy of the data. In this latter case, coherency must be managed by software, or it can be done with atomic operations if appropriate.

There are also at least two different kinds of high-performance accelerators: vector units and move units (data movement engines) that are integral parts of each core. The vector acceleration units can execute arbitrary length vector instructions, unlike conventional cores which execute short SIMD instructions. The vector units are done such that multiple vector instructions can be running at the same time and execution performance is largely independent of the length of the vectors.

Multiple data movement engines are also added to the processor and vector units to provide general data movement, such as transposing multidimensional matrices. Both the vector and move accelerators can perform their functions largely independent of the thread processes or can be directly controlled by and interact directly with executing threads. Local scratch pads are included in the core and accelerators (shown in the block diagram as Mem-blocks) where applications can store data items at very low access latency. These local memories also provide a direct core-to-remote-core messaging capability where messages can be placed ready for processing. By providing separate memory blocks, vector units and processor cores can run independently.

Thread execution in each core is organized into blocks of time. An executing thread can have a single clock in an execution block or it can have multiple clocks. This enables the number of threads in execution and the execution power of threads to vary depending on application requirement. There can also be execution threads that are inactive but ready to execute when an executing thread would be idle (waiting on some action to complete), or can be loaded or unloaded. Any thread seeing more than a one clock-block wait time is replaced with an inactive thread that is ready but is waiting for active execution time.

Chapter 3

Memory System

3.1 Memory Drivers

The next generation of main memory –DDR-4– will be available in the near future and is expected to be the basis of main memory for at least the next three to four years. But that memory cannot be used –at least not by itself– as the basis for the high-end and exascale systems envisioned here. As the DDR-4 standard pushes engineering to the limit, JEDEC (the standards body that supported the development and implementation of the DDR memory standards) will not provide a DDR-5 standard, forcing system vendors to explore alternative technologies.

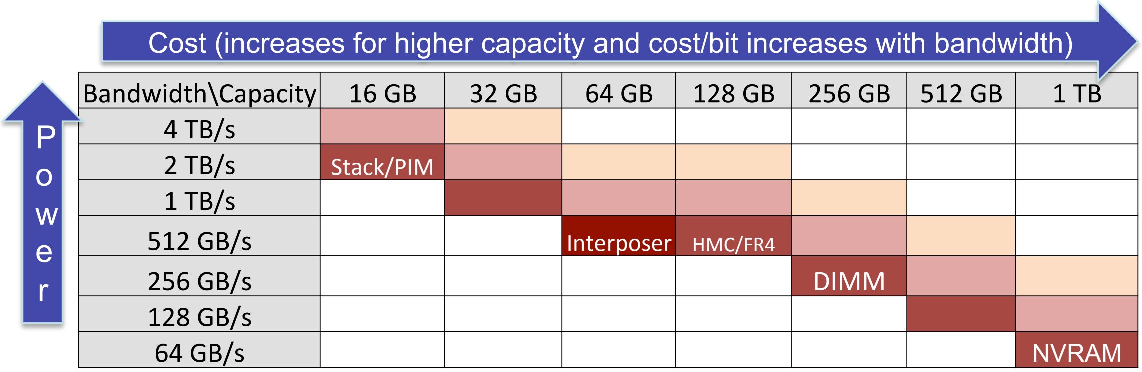
A promising alternative to DDR-5 is to provide a hybrid memory system that will integrates multiple types of different memory components with different sizes, bandwidths, and access methods. There are also an efforts underway to use some very different DRAM parts to build an integrated memory subsystem; this memory has characteristics that are very different than DDR-4 technology such that power and energy would be reduced with respect to current memory. In addition, because the power is reduced, the size of memory can be greatly increased.

As an example, consider a system that has two types of components in its memory system. This hypothetical system would contain a fairly small number of parts that are mounted on top of or are in the same carrier as the CPU chip (e.g. one to four memory parts with each part being a 3D stack of redesigned memory die). The bandwidth of these integrated memory parts will likely be in the low 100's of gigabytes-per-second each – much higher than any current memory parts or memory modules. But this increased bandwidth comes at a cost of lower capacity; therefore, this high-bandwidth memory alone will be unable to support any realistic applications and must be paired with a higher capacity, lower bandwidth memory. This higher capacity, lower bandwidth memory will likely be something like DDR-4, and will provide the majority of the system's memory capacity. These trade-offs are diagrammed in Figure 3.1. Of course such a two-level structure raises issues with respect to system software: compilers, libraries, and OS support, and other elements of the system software stack.

Finally, these new high-bandwidth components will come at significantly higher procurement cost. We envision analysis being performed to not only to justify the increased cost of these components with respect to application performance, but also to determine the optimum ratio of high-bandwidth, low-capacity memory to low-bandwidth, high-capacity memory. Current market data shows the high-speed low-capacity stacked memory is consistently 3x-4x more expensive per bit of capacity than the higher-capacity lower-bandwidth DDR memories. This ratio is governed more by market dynamics than costs, and the trend is expected to continue.

This performance vs. cost analysis will also need to include relative energy consumption for each memory type. As an example, if DDR-4 is used as the basis for the majority of a system's memory capacity, then the total capacity in such a system will be significantly reduced when compared to a system utilizing DDR-4 in combination with technologies which are more energy efficient, such as NVRAM, and/or higher performance, such as HMC. This performance vs. cost vs. energy trade-off analysis does not have an immediate impact on the study here, but will affect choices that must be made at a system level such as: how much is additional memory worth with respect to system performance, total size on the computer room floor, and other facility considerations.

Finally, a system need not be restricted to two levels of memory and may have three or more levels with



Bandwidth\Capacity	16 GB	32 GB	64 GB	128 GB	256 GB	512 GB	1 TB
4 TB/s							
2 TB/s	Stack/PIM						
1 TB/s							
512 GB/s			Interposer	HMC/FR4			
256 GB/s					DIMM		
128 GB/s							
64 GB/s							NVRAM

Figure 3.1: Known memory technologies enable us to get memory capacity or memory bandwidth, but not both in the same device technology. This motivates the move towards a new organization for external memories.

each level composed of a different memory technology, such as NVRAM, DRAM, 3D Stacked, or other memory technologies. As a result, future application analysis must account for complexities created by these multi-level memory systems. Despite the increased complexity, however, the performance benefits of such a system should greatly outweigh the additional burden in programming brought by multi-level memory; for instance, the amount of data movement will be reduced both for cache memory and scratch space resulting in reduced energy consumption and greater performance.

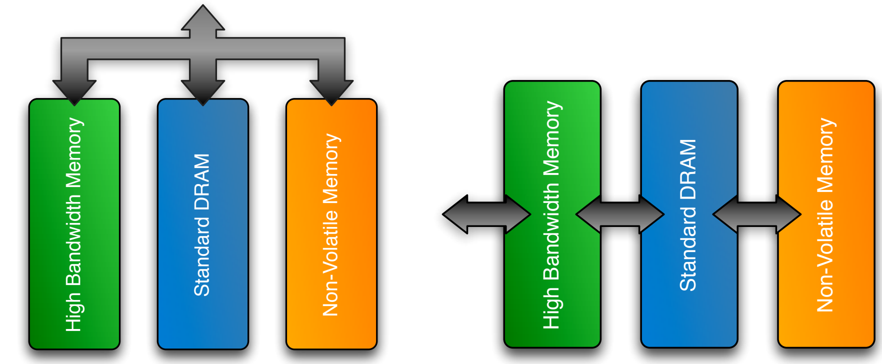
Further advantages can be demonstrated if, for example, NVRAM – which boasts a lower energy cost per bit accessed compared to DRAM – is used as part of main memory, allowing for an increase in total memory capacity while decreasing total energy consumption. Additional research is needed to find the best ways to allow the application programmer to make use of these changed and expanded capabilities.

3.2 Future Memory Abstractions

For each of the node architectures presented in Section 2.2, the layout of memory within the node can be regarded as an orthogonal choice – that, it is possible for architectures to mix and match arrangements for compute and selection of memory components. Due to the explosive growth in thread count expected to be present in the exascale machine the total amount of memory per socket must increase accordingly, potentially in the range of 1 terabyte or more per node. As explained in Section 3.1, it is expected that the memory system will be made up of multiple levels of memory that will trade capacity for bandwidth. In many ways this concept is not so unfamiliar to developers who may optimize problem sizes to fit in local caches, etc. These new memory levels will present additional challenges to developers as they select correct working set sizes for their applications. As an initial first attempt to characterize likely memory sub-systems, we propose three pools of memory:

1. High-bandwidth-memory: A fast, but relatively small-capacity, high bandwidth memory technology based on new memory standards such as JEDEC's high bandwidth memory (HBM) standard [6], Micron's hybrid memory cube (HMC) technology [16], or a technology like WideIO [5].
2. Standard DRAM: A larger capacity pool of slower DDR DRAM memory.
3. Non-volatile-memory: A very large but slower pool of non-volatile based memory.

As shown in Figure 3.2(a), w propose two principle approaches to architect these memory pools: (a) a physical address partitioning scheme in which the entire physical space is split into blocks allowing each memory pool to be individually addressed, and (b) a system in which faster memory pools are used to cache slower levels in the memory system. A third possible approach to constructing a memory system is to provide a blending of these two models with either user-defined or boot-time defined partitioning of the memory systems into partial cache and partial address space partitioned modes.



(a) Physical Address Partitioned Memory Subsystem

(b) Multi-Level Cached Memory System

Figure 3.2: Memory Subsystem Layouts

3.2.1 Physical Address Partitioned Memory System

In a physical address partitioned memory system, the entire physical memory address space is split into discrete ranges of addresses for each pool of memory (Figure 3.2(a)). This allows an operating system or runtime to decide on the location of a memory allocation by mapping the request to a specific physical address, either through a virtual memory map or through the generation of a pointer to a physical location. This system therefore allows for a series of specialized memory allocation routines to be provided to applications. An application developer can specifically request the class of memory at allocation time. We envision that an application developer will be able to request a specific policy should an allocation fail due to memory pool become exhaustion. Possible policies include allocation failure, resulting in an exception, or a dynamic shift in allocation target to the next slowest memory pool. While the processor cores in this system may possess inclusive caches, there will be no hardware support for utilizing the faster memory pools for caching slower pools. This lack of hardware support may be overcome if an application developer or system runtime explicitly implements this caching behavior.

3.2.2 Multi-Level Cached Memory System

An alternative memory model is that multiple pools are present in the node, but they are arranged to behave as large caches for slower levels of the memory hierarchy (Figure 3.2(b)). For instance, a high-bandwidth memory pool is used as a caching mechanism for slower DDR or slower non-volatile memory. This would require hardware caching mechanisms to be added to the memory system and, in some cases, may permit an application developer or system runtime to select the cache replacement policy employed in the system. It is expected that this system will possess hardware support for the caching behavior between memory levels; however, a system lacking this hardware support could implement a system runtime that monitors memory accesses to implement an equivalent behavior in software.

3.3 3-D Stacked Memory Systems, Processing in Memory (PIM), and Processing Near Memory PNM

A new technology that will emerge in the memory hierarchy is 3D-stacked memory (Table 3.1). These stacks of memory will have a logic layer at the base to handle read and write requests to the stack. Not only will there be multiple memory dies in a single memory component, but in some versions these memory dies will be mounted directly on CPU chips resulting in greater density with a reduced energy footprint. Additionally, processor-in-memory (PIM) functionality may emerge in conjunction with the stacked memory architectures that include logic layers at the base of the memory stacks. These PIM capabilities offer acceleration to many memory operations, such as atomics, gather-scatter, pointer chasing, search, and other memory bandwidth intensive operations. These accelerators can execute faster and more efficiently than general-purpose hardware.

Configuration	Bandwidth	Capacity / Node
Single-Level HMC		
HMC (4 HMC “modules”)	∽960 GB/s	∽2TB†
Multi-Level DRAM		
HBM (4 stacks @ 200GB/s)	∽800 GB/s	∽64 GB
DDR (4 channels (8 DIMMs) @ 20GB/s)	∽80 GB/s	∽512 GB
NVRAM		
NVRAM	10–20 GB/s	4–8× DRAM

Table 3.1: Approximate Bandwidths and Capacities of Memory Subsystem
† See notes in Section 3.3

An SoC design flow creates an opportunity for many types of acceleration functions to improve application performance. These accelerators can make data movement more efficient by avoiding unnecessary copies, or by hiding or eliminating overhead in the memory system or a systems interconnect network. However, how best to expose these operations to the programmer is still an active area of research.

Some 3D memory technologies, such as the HMC, allow memory parts or modules to be “chained” in different topologies. In contrast, DDR connects a small number of memory parts to a processor. Similarly, there are other standards for high-performance memory on the horizon, such as HBM that also only connect a single memory part to a single processor. “Chained” memory systems differ from DDR and HBM in their ability to support a very high memory capacity per-node. The limitations of per-node memory capacity when using chained systems would be dominated by dollar cost and total allowable power consumption. While the relative dollar cost of stacked memory is still unknown, it is expected to be more ($ per bit) than DDR. In contrast, the power cost – Joules per accessed memory bit – is expected to be significantly less for 3D stacked memory when compared to DDR.

Chapter 4

Programming Considerations

Emerging architectures are bringing with them a shift in constraints that will require careful consideration for development of future exascale-class applications, particularly for those demanding an evolutionary approach to porting [1]. In the future we expect that new optimization goals will become commonplace, specifically that application developers will target the minimizing of data movement and the maximization of computational intensity for each piece of data loaded from memory, rather than a focusing on increasing the raw compute performance (FLOP/s) used by each processor. The optimization of data movement is becoming more complex, as future architectures may lack the global cache coherence found on today's systems and we anticipate an explosion of on-chip parallelism. These architectural shifts must be accompanied by changes in programming models that are more adept at preserving data locality, minimizing data movement, and expressing massive parallelism.

4.1 Data Movement/Coherence Model

Programmers have relied on a large shared cache to virtualize data movement. However, shared memory and cache coherence across all threads in a single socket will no longer be the norm. As the number of cores per socket increase programmers will be faced with the need to explicitly manage data movement. In the best case, regional coherence domains will automatically manage memory between a subset of cores. Between these domains there may be a relaxed consistency model, but the burden will still fall on developers to efficiently and manually share data on-chip between these domains.

Non-uniform memory access issues are already prevalent in today's machines. With the high core counts (in the 1000s), the issue will be more detrimental to performance because programmers can no longer assume execution units or the various memory components are equidistant. The importance of locality in these new architectures that utilize explicitly managed memory systems will drive the development of a more data-centric programming model allowing programmers to more naturally express the data layout of their programs in memory.

Moreover, configuring local memory as cache and scratchpad memory will become an important tuning parameter. Current GPUs allow a split of 1/4, 1/2 or 3/4 of the on-chip storage between hardware-managed and software-managed memory. We expect that future systems will allow for even more flexible configuration in the split between scratchpad and cache.

4.2 Hardware Performance Heterogeneity

Current parallel programming paradigms such as the bulk-synchronous parallel (BSP) [18] model implicitly assume that the hardware is homogeneous in performance. During each phase of computation, each worker thread is assigned an equal amount of work, after which they wait in a barrier for everyone to complete. If all threads complete their work at the same time, this computational paradigm is extremely efficient. On the other hand, if some threads fail to complete their portion of the work, then large amounts of time could be wasted by many threads waiting at the barrier. As on-chip parallelism increases, the performance of the cores on a chip will become less and less homogeneous, which will require adaptation by the programmer, programming model, domain decomposition and partitioning tools, and/or system runtime to maintain a high level of performance.

One source of increasing heterogeneity is the use of multiple types of cores on a chip such as thin and fat cores. As described in section 2.1.5, largely parallel tasks will run more efficiently on throughput-optimized *thin cores*, while mostly serial tasks will run more efficiently on latency-optimized *fat cores*. The programming model and runtime must be aware of this distinction and help the programmer utilize the right set of resources for the particular set of tasks at hand so the hardware may be utilized in an efficient manner. A potential type of heterogeneity similar to this is near-threshold-voltage (NTV) operation [7], which individually scales the frequency and voltage of cores to their most energy-efficient operational range. The compiler may utilize the frequency scaling information to make static scheduling decisions, or the runtime may adapt based on how quickly it observes tasks run on the different cores.

Another source of irregularity is imbalanced workloads, which may create thermal hotspots on the chip that result in frequency throttling. Such throttling will temporarily impact the execution rate of subsets of cores on a chip. Furthermore, if exascale machines are subject to increased rates of hardware faults, then the associated mitigation techniques such as error correction or recovery could cause large delays to subsets of the threads or processes in an application.

In each of these cases, a strict BSP program formulation would result in all of the worker threads waiting for the affected threads to complete before moving on to the next phase of computation or communication. Future machines with higher degrees of performance heterogeneity are therefore likely to rely on runtime systems to dynamically adapt to changes in hardware performance or reliability. Current research strategies for programming to these performance variations include extending existing languages and parallel APIs, such as C++11, future C++ standards-based language level parallelism, and traditional runtimes including OpenMP, and developing alternative languages and task-parallel runtimes, such as Chapel and the Open Community Runtime (OCR) [13]. For each approach, the solution will need to provide an efficient mapping with a low burden to the application programmer.

4.3 Increased Parallelism

With increasing core counts, more parallelism will need to be exposed by the applications to keep all of the cores on the chip busy. This could prove challenging if application developers have structured their algorithms in a way that limits the level of concurrency expressed to the programming model.

There are four broad sources of parallelism: (1) Instruction-level parallelism between independent instructions, (2) Vectorization of an instruction over multiple data elements, (3) Thread-level parallelism between independent execution contexts/threads, and finally (4) Domain decomposition-level parallelism, which is typical of scientific applications designed to run over massively parallel nodes.

Increased concurrency will need to be exposed in the software stack to utilize large numbers of functional units and to hide higher latencies through hyper-threading. There is potential for future programming languages, compilers, and runtimes to help developers expose greater parallelism through data-centric programming models that allow automatic task decomposition and pipelining. These same systems that help reason about task and data dependencies for scheduling could also be used to manage data movement across the chip to increase access locality for energy and performance benefits described earlier.

4.4 Fault Tolerance and Recovery

Checkpoint-restart systems commonly used with BSP systems will not scale well on massively parallel architectures since the number of threads that would require a roll back and restart would be much higher. Tolerating faults efficiently will require a more localized method for fault recovery, replacing the global coordination and synchronization of today's methods.

Chapter 5

Proxy Architectures for Exascale Computing

Proxy architecture models (PAMs) were introduced as a codesign counterpart to proxy applications in the DOE ASCAC report on the Top Ten Exascale Research Challenges [11]. This Computer Architecture Laboratory (CAL) AMM document separates the PAMl concept into AMM and proxy architectures, but the intent is still to facilitate codesign and communication.

In this section we identify approximate estimates for key parameters of interest to application developers. Many of these parameters can be used in conjunction with the AMM models described previously to obtain rough estimates of full node performance. These parameters are intended to support design-space exploration and should not be used for parameter- or hardware- specific optimization as, at this point in the development of Exascale architectures, the estimates may have considerable error. In particular, hardware vendors might not implement every entry in the tables provided in future systems; for example, some future processors may not include a Level-3 cache.

5.1 Design Parameters

The following list of parameters allows application developers and hardware architects to tune any AMMs to their desire. The list is not exhaustive and will continue to grow as needed. Since this list is for all AMMs presented in this document, not all parameters are expected to be applicable to every AMM. In fact, we expect that for each AMM only a subset of this list of parameters will be used for architecture tuning. Likewise, not all parameters are useful for application developers, such as bandwidth of each level of the cache structure.

	Processor Cores	Gflop/s per Proc Core	NoC BW per Proc Core (GB/s)	Processor SIMD Vectors (Units x Width)	Accelerator Cores	Acc Memory BW (GB/s)	Acc Count per Node	TFLOP/s per Node[1]	Node Count
Homogeneous M.C. Opt1	256	64	8	8x16	None	None	None	16	62,500
Homogeneous M.C. Opt2	64	250	64	2x16	None	None	None	16	62,500
Discrete Acc. Opt1	32	250	64	2x16	O(1000)	O(1000)	4	16C + 2A	55,000
Discrete Acc. Opt2	128	64	8	8x16	O(1000)	O(1000)	16	8C + 16A	41,000
Integrated Acc. Opt1	32	64	64	2x16	O(1000)	O(1000)	Integrated	30	33,000
Integrated Acc. Opt2	128	16	8	8x16	O(1000)	O(1000)	Integrated	30	33,000
Heterogeneous M.C. Opt1	16 / 192	250	64 / 8	8x16 / 2x8	None	None	None	16	62,500
Heterogeneous M.C. Opt2	32 / 128	64	64 / 8	8x16 / 2x8	None	None	None	16	62,500
Concept Opt1	128	50	8	12x1	128	O(1000)	Integrated	6	125,000
Concept Opt2	128	64	8	12x1	128	O(1000)	Integrated	8	125,000

Table 5.1: $Opt1$ and $Opt1$ represent possible proxy options for the abstract machine model. $M.C$: multi-core, Acc: Accelerator, BW: bandwidth, $Proc$: processor, For models with accelerators and cores, C denotes to FLOP/s from the CPU cores and A denotes to FLOP/s from Accelerators.

5.1.1 Processor

Parameter	Expected Range	Notes
Bandwidths (GB/s)		
Chip ↔ Memory	60–100	To off-chip DDR DRAM
Chip ↔ Memory	600–1200	To on-chip DRAM
Chip ↔ Memory	600–1200	To off-chip HMC-like
Core ↔ L1 Cache	O(100) – O(1000)	
Core ↔ L2 Cache	O(100)	
Core ↔ L3 Cache	Varies by NoC Bandwidth	
Capacities		
L1 Cache	8KB–128KB	Per Core
L2 Cache	256KB–2MB	Per Core
L3 Cache	64MB–128MB	Likely to be shared amongst groups of cores/accelerators
L4 Cache	2GB–4GB	Not on all systems, likely to be off-package embedded-DRAM
Memory System Parameters		
Cache block size	64-128B	
Number of cache levels	2–4	
Coherency domains	1–8	per chip
Network-On-Chip		
Topology	Options include: Mesh, fat-tree, hierarchical ring, cross-bar, etc. Still active research area.	
Bandwidth	8-64 GB/s	per core
Latency	1–10 ns	neighbor cores
Latency	10–50 ns	cross chip
Core Parameters		
Number of cores	64-256	per chip
Threads per core	2–64	
Thread Switching	Possible policies include: Each cycle, on long-latency event, time quanta	
SIMD/vector width	4–8 DP FP	
Dispatch, issue, execution widths	2–8	Simple processors may have limited dual dispatch (i.e. 1 general purpose instruction and 1 memory)
Max references outstanding	8–128	per core
Atomic operations		Possible implementations include: Simple or transactional

5.1.2 Memory

Parameter	Range	Notes
Capacities		
In-package DRAM Memory	32–64 GB	
Off-Chip DRAM Memory	∽512GB–2TB	Memory capacity may be much larger, at the cost of fewer nodes. See notes in Section 3.3
Off-Chip NVRAM Memory	∽2TB–16TB	
Capabilities		
Extended Memory Semantics	None, Full/Empty Bits, Transactional, data movement, specialized compute, general compute.	
In-Package Memory Technologies/Levels		
HBM/Wide-IO	∽16GB per stack, ∽ 200GB/s per stack. Can be stacked directly on processor. Cannot be "chained."	
Off-Package Memory Technologies/Levels		
HMC	∽16GB per stack, ∽240GB/sec per stack. Can be "chained" to increase capacity. Multiple access sizes.	
DDR-4	∽64GB per DIMM, 20GB/s per channel, up to 2 DIMMs per channel. Optimized for cacheline sized access.	
NVRAM	∽4-8× capacity of DRAM, 10–20GB/s. Requires KB sized access. Highly assymetric access latency.	

5.1.3 System Network

Parameter	Range	Notes
Topology	Possible implementations include: Torus, fat-tree, hierarchical ring, or Dragonfly	
Bisection Bandwidth		1/8 to 1/2 of injection bandwidth
Injection BW	100GB/s - 400GB/s	per node
Messaging Rate	250MMsg/s	Two-sided communications
Message Processing Rate	1BMsg/s	One-side communications
Latency	500-1500ns	Two-side communications nearest neighbor
Latency	400-600ns	One-sided communications nearest neighbor
Latency	3-5μs	Cross-machine

5.2 Reference Proxy Architecture Instantiations

Some expanded AMMs in Section 2.2 have their roots in existing advanced technology processors. One of these could be a harbinger of what an exascale processor may look like. We provide their proxy architecture information here for reference.

5.2.1 Homogeneous Manycore Model: Intel Sandy Bridge

An example of the homogeneous manycore processor model (Section 2.2.1) is the Intel Sandy Bridge (shown in Figure 5.1), a 64-bit, multi-core, dual-threaded, four issue, out-of-order microprocessor. Each core has 32KB of L1 data cache, 32KB of instruction cache and 256KB of L2 cache. The processor consists of up to eight cores, a large shared L3 cache, four DDR3 memory controllers, and 32 lanes of PCI-Express (PCIe). There is a 32-Byte ring-based interconnect between cores, graphics, L3 cache, and system agent domain.

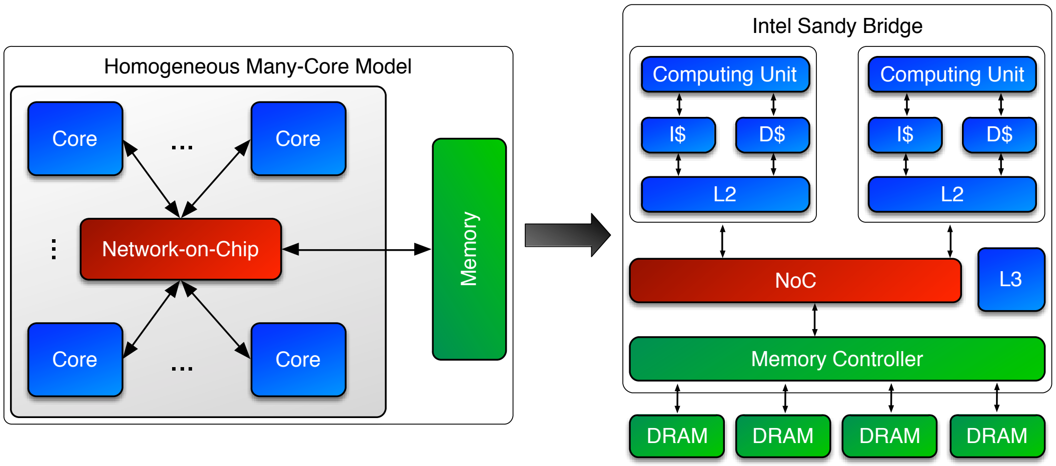


Figure 5.1: Reference proxy architecture instantiation: Intel Sandy Bridge

5.2.2 Multicore CPU + Discrete Accelerators Model: Sandy Bridge with Discrete NVIDIA GPU Accelerators

An example of the multicore CPU + discrete accelerators model (Section 2.2.4) is a system utilizing Intel's Sandy Bridge multi-core processor with NVIDIA GPU-based accelerators (shown in Figure 5.2). Each node is comprised of a dual-socket Xeon host-processor (Sandy Bridge E5-2660 2.66GHz) compute nodes with two NVIDIA K20X Kepler GPU cards per node. These GPU cards connect to the main system via a PCIe Gen2x16 interface. Each GPU card comprises of $2,688$ processor cores and 6GB of 384-bit GDDR-5 memory with a peak bandwidth of 250 GB/s.

5.2.3 Integrated CPU + Accelerators Model: AMD Fusion APU Llano

An example of an integrated CPU and accelerators (Section 2.2.3) type of architecture is the AMD Fusion APU Llano shown in Figure 5.3. The Llano architecture consists of three main modules: four CPU x86 cores for general-purpose activities; an integrated graphic core; and I/O, memory, and display controllers. Each CPU core has its own L1 and L2 caches. The unified memory architecture (UMA) implemented in Llano allows processors and graphics to share a common memory space. The GPUs also have a dedicated non-coherent interface to the memory controller (shown with a dotted line) for commands and data. Therefore, the memory controller has to arbitrate between coherent (ordered) and non-coherent accesses to memory. Note that the CPU is not intended to read from GPU memory, hence the performance is singularly poor because of the necessary synchronization.

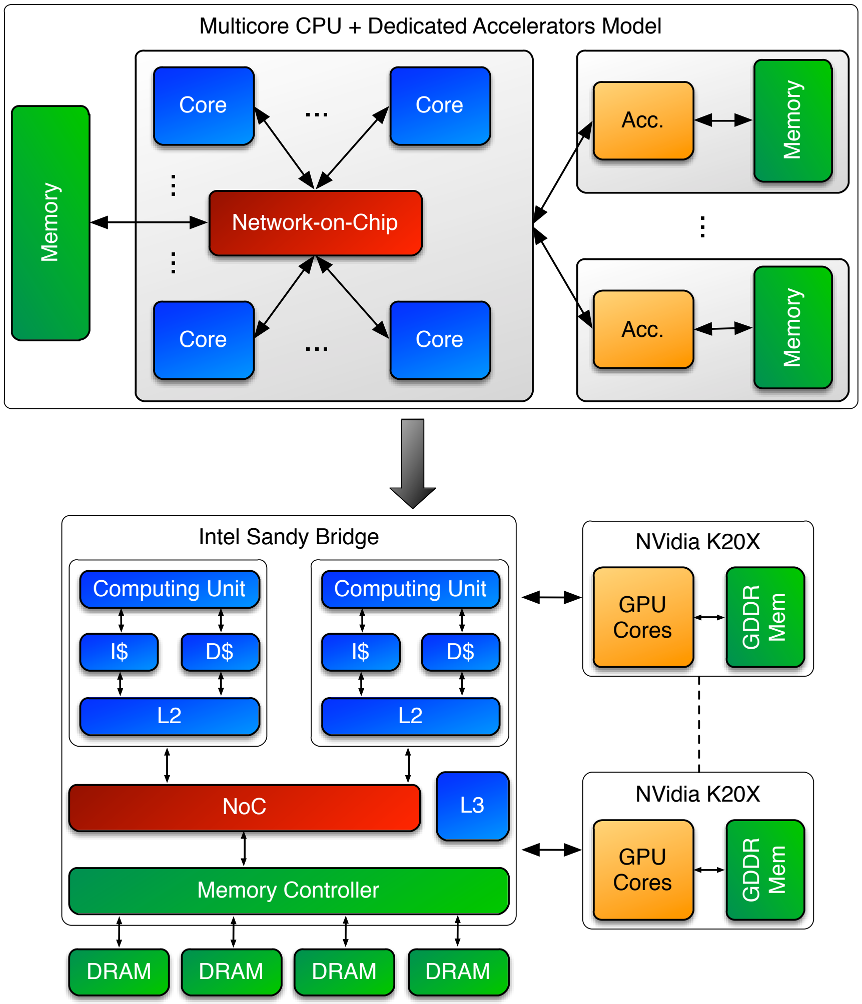


Figure 5.2: Reference proxy architecture instantiation: Multicore CPU with Discrete GPU Accelerators

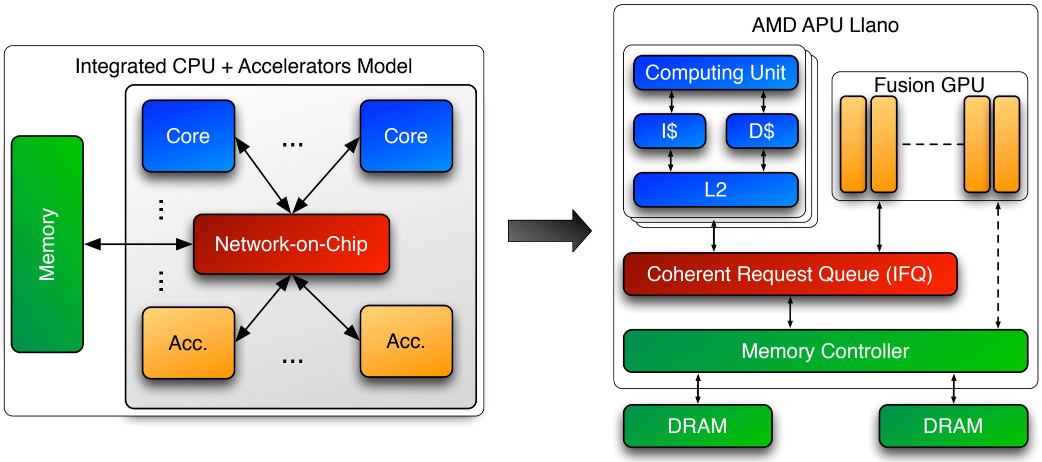


Figure 5.3: Reference proxy architecture instantiation: AMD APU Llano

Chapter 6

Conclusion

Advanced memory systems, complex on-chip networks, heterogeneous systems and new programming models are just some of the challenges facing users of an exascale machine [11]. The AMMs presented in this document are meant to capture the design trends in exascale systems and provide application software developers with a simplified yet sufficiently detailed view of the computer architecture. This document also introduces and describes proxy architectures, a parameterized instantiation of a machine model. Proxy architectures are meant to be a communication vehicle between hardware architects and system software developers or application performance analysts.

Communication and coordination The Computer Architecture Laboratory (CAL) has an important role in facilitating communication between laboratory open research projects and hardware architecture research and development by DOE's Fast Forward and Design Forward companies that is usually proprietary. CAL supports this communication by developing AMMs and non-proprietary, open proxy architectures. These proxy architectures and AMMs can be used to provide applications, algorithms and co-design projects with a target model of the hardware, node, and system architecture to guide their software development efforts. An important characteristic of these proxy architecture models is their ability to also capture lower level architectural parameters to provide a non-proprietary description of advanced architecture concepts for quantitative analysis and design space exploration with our architectural simulation tools. The development of proxy architectures and AMMs is intended to allow open exchange of advanced hardware architecture concepts without disclosing intellectual property.

Metrics for success The key metric for success will be the use of AMMs at the abstract high level for guiding application development, and proxy architectures at the more detailed lower level for guiding system software and architectural simulation efforts. The successful outcome is co-designed hardware architectures and associated software.

AMM and proxy architecture evolution Co-design is a multi-disciplinary endeavor. It is an activity that bridges many different technical communities, and as we have seen with proxy applications, having common terminology and touch points is extremely valuable in fostering multi-disciplinary collaborations. In this spirit, the CAL project is contributing this initial set of common definitions for AMMs and their associated proxy architectures to the DOE exascale program. As with proxy applications, our AMMs and Proxy Architectures are expected to evolve through the co-design process.

Bibliography

[1] R.F. Barrett, S.D. Hammond, C.T. Vaughan, D.W. Doerfler, M.A. Heroux, J.P. Luitjens, and D. Roweth. Navigating an Evolutionary Fast Path to Exascale. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 355–365. IEEE, 2012.

[2] R. Brightwell and K.D. Underwood. An Analysis of NIC Resource Usage for Offloading MPI. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 183–, April 2004.

[3] Byn Choi et al. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 155–166, Washington, DC, USA, 2011. IEEE Computer Society.

[4] Jack Dongarra et al. The International Exascale Software Project Roadmap. *IJHPCA*, 25(1):3–60, 2011.

[5] JEDEC. Wide I/O Single Data Rate. JESC 229, JEDEC, December 2011. `http://www.jedec.org/standards-documents/docs/jesd229`.

[6] JEDEC. High Bandwidth Memory (HBM) DRAM. JESC 235, JEDEC, October 2013. `http://www.jedec.org/standards-documents/docs/jesd235`.

[7] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (ntv) design: opportunities and challenges. In *DAC*, pages 1153–1158, 2012.

[8] S. Kaxiras and G. Keramidas. Sarc coherence: Scaling directory cache coherence in performance and power. *Micro, IEEE*, 30(5):54–65, Sept 2010.

[9] Martha Kim. Scaling Theory and Machine Abstractions. `http://www.cs.columbia.edu/~martha/courses/4130/au13/pdfs/scaling-theory.pdf`, Sept 2013.

[10] Peter M. Kogge and John Shalf. Exascale computing trends: Adjusting to the "new normal"' for computer architecture. *Computing in Science and Engineering*, 15(6):16–26, 2013.

[11] Robert Lucas AND et. al. Top Ten Exascale Research Challenges, DOE ASCAC Subcommittee Report, February 2014.

[12] A. Ros, B. Cuesta, M.E. Gomez, A. Robles, and J. Duato. Cache miss characterization in hierarchical large-scale cache-coherent systems. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 691–696, July 2012.

[13] V. Sarkar, B. Chapman, W. Gropp, and R. Knauserhase. Building an Open Community Runtime (OCR) framework for Exascale Systems, 2012.

[14] M. Schuchhardt, A. Das, N. Hardavellas, G. Memik, and A. Choudhary. The impact of dynamic directories on multicore interconnects. *Computer*, 46(10):32–39, October 2013.

[15] John Shalf, Sudip S. Dosanjh, and John Morrison. Exascale computing technology challenges. In *VECPAR*, pages 1–25, 2010.

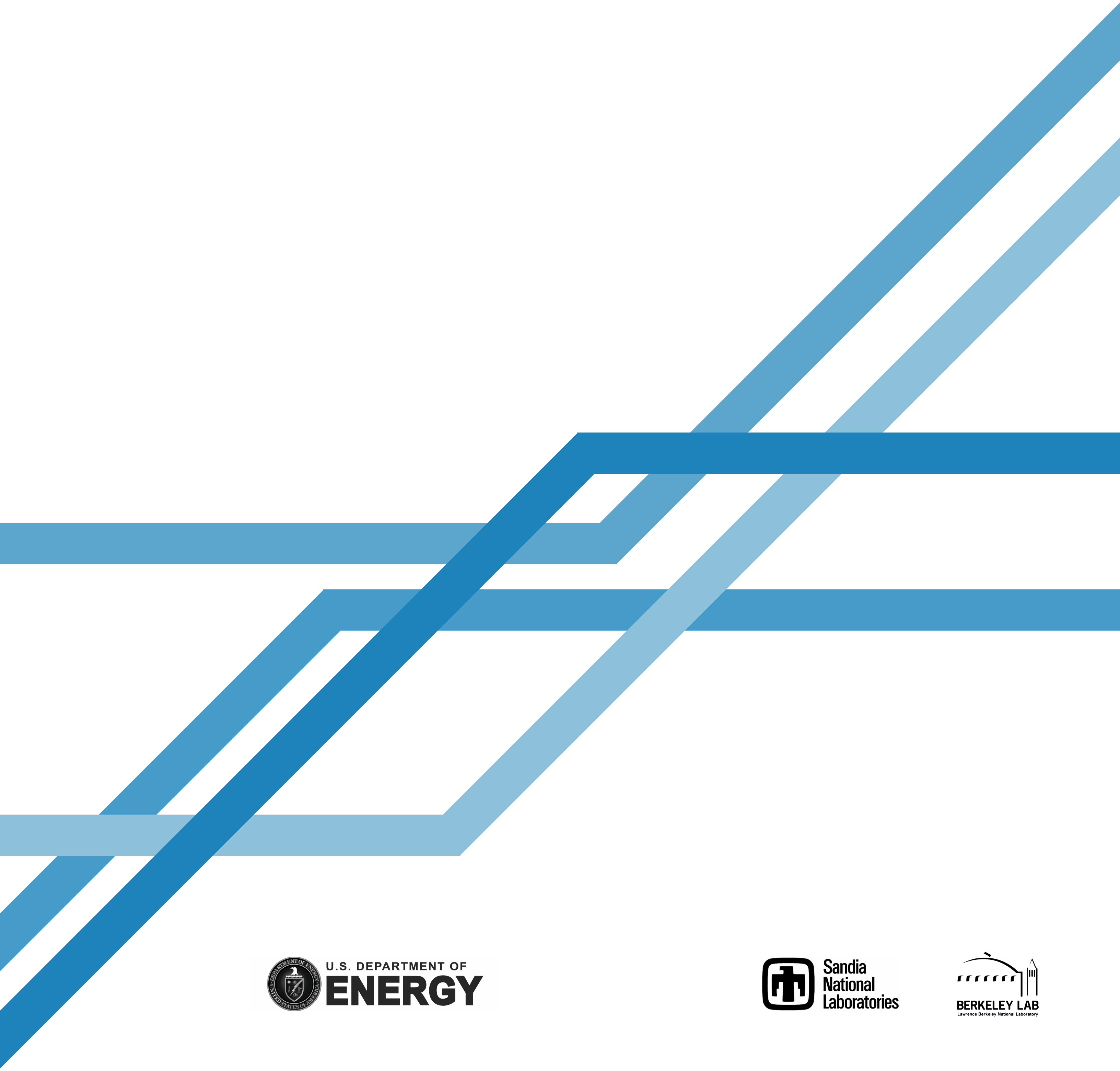
[16] Micron Technology. Hybrid Memory Cube Specification, 2014.

[17] K.D. Underwood, K.S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. A Hardware Acceleration Unit for MPI Queue Processing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 96b–96b, April 2005.

[18] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[19] Brian van Straalen, John Shalf, Terry J. Ligocki, Noel Keen, and Woo-Sun Yang. Scalability challenges for massively parallel amr applications. In *IPDPS*, pages 1–12, 2009.

[20] Yi Xu, Yu Du, Youtao Zhang, and Jun Yang. A composite and scalable cache coherence protocol for large scale cmps. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 285–294, New York, NY, USA, 2011. ACM.



U.S. DEPARTMENT OF ENERGY

Sandia National Laboratories

BERKELEY LAB
Lawrence Berkeley National Laboratory