Lightweight Requirements Engineering for Exascale Co-design

Alexandru Calotoiu, Alexander Graf
TU Darmstadt, Germany
{calotoiu, graf}@cs.tu-darmstadt.de

Torsten Hoefler
ETH Zurich, Switzerland
htor@inf.ethz.ch

Daniel Lorenz, Sebastian Rinke, Felix Wolf
TU Darmstadt, Germany
{lorenz, rinke, wolf}@cs.tu-darmstadt.de

***Abstract*—Given the tremendous cost of an exascale system, its architecture must match the requirements of the applications it is supposed to run as precisely as possible. Conversely, applications must be designed such that building an appropriate system becomes feasible, motivating the idea of co-design. In this process, a fundamental aspect of the application requirements are the rates at which the demands for different resources grow as a code is scaled to a larger machine. However, if the anticipated scale exceeds the size of available platforms this demand can no longer be measured. This is clearly the case when designing an exascale system. Moreover, creating analytical models to predict these requirements is often too laborious—especially when the number and complexity of target applications is high. In this paper, we show how automated performance modeling can be used to quickly predict application requirements for varying scales and problem sizes. Following this approach, we determine the exascale requirements of five scientific codes and use them to illustrate system design tradeoffs.**

I. Introduction

Ever-growing computational demands from domains such as climate science, theoretical physics, and neuroscience require large-scale machines in the near future. Planning the design and detailed configuration of such systems is a daunting task since they are often major investments, up to half a billion dollar over their lifetime. Thus, it is extremely important that the machine efficiently supports the execution of all target applications. Designing it is a multi-year planning effort while it stays "top of the line" only for three to five years after installation. Thus, while the machine must be tailored to its workload, it must also be productive from day one on.

Co-designing applications and the system is a powerful technique to ensure early and sustained productivity as well as good system design. In their early phases, such co-designs often rest on *back-of-the-envelope* (BOE) calculations. For example, BOE calculations have been famously used to determine the well-known "bytes-to-flop ratio" for the network and memory in early Cray machines. They continue to gain popularity with requirements-balance models such as the roofline model [1]. In general, such calculations allow problems in applications to be detected early on and their severity to be determined years before the machine is installed or the first prototype becomes available. This is increasingly important since mitigating such problems can often take several person-years. On the system side, BOE calculations allow designers to adjust system parameters to target applications, for example, they can be used to determine the required bytes-to-flop ratio of memory, network, or even the file system. In addition, they can be used to determine required memory sizes, usability of accelerators and co-processors, and even the number of sockets and size of shared-memory domains in the target system.

We automate these BOE calculations in a lightweight *requirements analysis* for scalable parallel applications. We introduce a minimal set of hardware-independent application-centric requirements that cover the most significant aspects of application behavior. Combining standard performance profiling [2], [3] and stack-distance sampling [4] with a lightweight automatic performance-modeling method [5], [6], we generate empirical models of these requirements that allow projections for different numbers of processes and problem sizes.

Once empirical models are established for an interesting set of requirements, the designer can use them to "play" with configurations such as (1) the amount of memory per node, (2) the speed of memory per node, (3) the network injection speed (how many adapters per node), or (4) the number of cores per socket or node (e.g., many- vs. multicore) etc.. Given the degree of automation we provide, the number of applications and system design choices to be included in the co-design process can be much higher than in a manual study, substantially expanding its breadth.

In this work, we demonstrate the power of our technique by analyzing both large-scale scientific applications and exascale proxy applications and the appropriateness of various system designs. The major contributions of this paper are:

- The introduction of a set of application-centric requirements that cover performance critical application aspects without considering hardware-specific behavior
- The integration and extension of existing performance tools [2], [3], [4], [5], [6] that allows the requirements of parallel applications to be modeled, including memory consumption and access locality
- A technique for practical co-design that extrapolates application requirements to an envisioned system and points out possible bottlenecks on both sides
- A case study with five applications that shows how they would respond to relative system upgrades and how well they would match three different exascale candidate systems

TABLE I: Requirement metrics.

Resource	Metric
Memory footprint	# Bytes used (resident memory size)
Computation	# Floating-point operations (#FLOP)
Network communication	# Bytes sent / received
Memory access	# Loads / stores; stack distance

II. Approach

As the foundation of our approach, we define a very simple notion of requirements that supports their quantification in terms of the amount of data to be stored, processed, or transferred by an application. Knowing these numbers alone does not target a precise prediction of application runtime but can serve as an indicator of the relative importance of certain system resources and how this ratio changes as we scale a program to a larger system, as the analysis in Section III will show. Ultimately, our requirements are expressed in the form of empirical models that allow projections for different numbers of processes and problem sizes.

A. Application-centric requirements

We choose requirements to be purely application centric, that is, we do not make any assumption about the hardware other than the ability to run the code as is. Hence, all our requirement metrics refer to data flow at the interface between hard- and software – not between lower layers of the hardware. While specific hardware features could improve the rate at which the requirements are fulfilled, the classes of behavior our requirement models capture will not change. For example, even if revolutionary hardware features double the speed at which floating-point computations are performed, if the number of floating-point computations that need to be performed grows quadratically with the number or processes, while all other requirements remain constant, the floating-point requirement will remain the bottleneck for that particular application as it scales up.

Since it is currently the predominant programming model and also expected to be highly influential in the future, we stipulate that of each target application an MPI version exists. Application requirements are then expressed as a set of functions $r(p, n)$ that predict the demand for resource r depending on the number of processes p and the problem size per process n. Because we regard thread-level concurrency merely as a way to satisfy the requirements, we consider requirements not below the granularity of processes, which may nevertheless be multithreaded—either locally or by launching GPU kernels.

Currently, we consider the requirement metrics listed in Table I, classified by the resource they refer to. I/O would be handled analogously to the network communication requirement. None of our analyzed applications includes significant I/O traffic, we therefore refrain from including I/O metrics in this analysis. Our metrics characterize application requirements in terms of space (i.e., memory consumption) and "data metabolism" (i.e., bytes processed in floating-point units or exchanged via memory and network). Because the amount of data moved between processor and memory subsystem alone is barely a reliable indicator of the pressure an application exerts on the memory subsystem, we also consider memory access locality. Specifically, we capture the stack distance [4] of memory accesses, which is the number of accesses to unique locations that occur between two accesses to the same location. The stack distance is related to the more commonly known reuse distance, a metric which counts the number of accesses between two accesses to the same location without trying to discern whether or not these accesses are unique. Figure 1 provides a small example for the difference between reuse distance and stack distance.

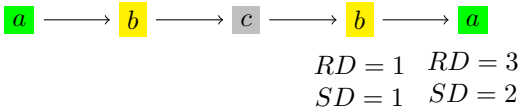

Fig. 1: Example of reuse distances and stack distances. a, b, c represent different memory locations. Accesses to these locations happen in the order indicated by the arrows. Reuse distances and stack distances are denoted by RD and SD respectively.

Thus, we define the *requirements model* of an application as a set of functions $r_i(p, n)$ with each r_i representing one of the requirement metrics listed in Table I. We deliberately refrain from modeling execution time and energy consumption because we consider them as manifestation of requirement fulfillment and not as their expression. This is an important difference to the classic approach of trying to create architecture-specific performance models that can predict actual performance (e.g., execution time) as a function of various hardware parameters. Because hardware parameters, such as a processing algorithm or network topology, may easily reach beyond a simple scalar variable such as the number of processors, architecture-specific models usually differ qualitatively across architectures. Thus, the modeling effort grows with the product of applications and architectures, as illustrated in Figure 2, while a requirements model needs to be created only once for each application. In addition, we provide a lightweight automatic tool chain to produce them empirically, further accelerating the analysis. By contrast, practically determining

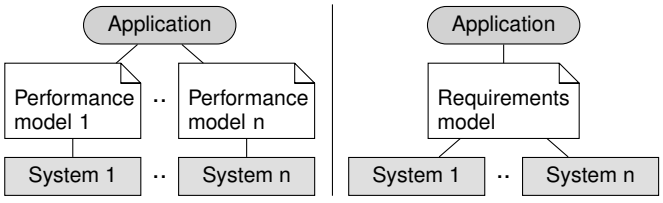

Fig. 2: Co-design with architecture-specific performance models (left) and purely application-centric requirements models (right).

the actual execution time of an application on a hypothetical system requires at least a simulator.

B. Data acquisition

The individual tools and methods used to gather data on the chosen requirements are established in the performance analysis community. To ensure our approach remains reproducible and easy to follow, we detail both how we employ each of the tools and how we generate our performance measurements. What is needed is an MPI implementation of the code, the profiling tools Score-P [2] and PAPI [3] to count floating-point operations, MPI data transfers, and loads and stores, Threadspotter [4] to measure memory locality, and the performance-model generator Extra-P [5], [6] to finally turn the collected requirement data into models. Threadspotter was modified to extract the data it collects for post-processing. Disregarding slight variations in the platform-dependent semantics of certain hardware counters, the requirements of an application can basically be obtained on any system. The metrics we acquire can be narrowed down to—in most cases—highly reproducible hardware and software counters such as floating-point operations or bytes injected into the network. A significant advantage of this approach is the simplicity and low effort with which requirements models can be instantiated for a given execution configuration. All metrics refer to a single process, since the matching hardware resources such as CPUs, memory, and network links grow roughly proportionally with the the number of processes expected on a machine.

Memory footprint: We invoke getrusage() to determine the resident memory occupied by each process across its entire lifetime. The memory footprint is one of the—if not the most important scalability inhibitor—simply because it can prevent an application from running at all.

Computation and communication: We acquire the metrics related to computation and communication using the Score-P profiler, which captures them at the granularity of individual function call paths. This allows bottlenecks to be precisely attributed to individual program locations. Given that the number of floating-point operations required per process is roughly independent of the number of threads used to compute them, we profile the application single threaded, which simplifies our workflow and makes it more robust.

Memory accesses and locality: The number of memory accesses and their stack distance is measured using a combination of Threadspotter and PAPI. Originally designed as an interactive locality optimizer for non-expert users, Threadspotter collects memory access distances only internally to derive optimization suggestions. However, we have modified it such that we can access these metrics directly. Threadspotter identifies loops in an application and instruments groups of instructions that access the same memory location within those loops. Therefore *instruction groups* represent the granularity at which distance metrics are provided. To keep the runtime dilation within practical limits (roughly a factor of eight), Threadspotter samples the execution in short bursts where all memory accesses are documented, followed by periods during which no measurements are gathered. Since Threadspotter does not count memory accesses, we let PAPI measure the number of load and store instructions for the entire program. Then, we estimate the number of memory accesses per instruction group based on the ratio of samples collected for different instruction groups. We address the known inaccuracy and non-determinism of load and store operations [7] by counting them for the entire run instead of individual functions with all fine-grained instrumentation removed.

To offset the non-determinism and possible variance of this process, we propose the following methodology to analyze memory locality: First, any instruction group with less than 100 samples gathered for each measurement configuration is ignored, because the risk of outliers adversely affecting the resulting model is too high. We have observed that both the number and magnitude of outliers is much greater when examining memory locality than it is for other metrics. This becomes obvious when considering a loop which is executed multiple times during the runtime of a program. In the loop itself, stack distance is low if it shows good locality. However, many memory accesses can happen between different executions of the loop, leading to higher stack distance when returning to the loop later on. To capture the most common behavior, namely the memory access pattern within loops, we model the median over all gathered samples.

C. Model generation

Originally developed to uncover scalability bugs in applications with a large code base, the model generator Extra-P [5], [6] uses empirical measurements to generate performance models in an attempt to help developers better understand their applications and determine performance bottlenecks of any kind. The goal is to offer the type of insight analytical models of codes bring developers without the significant manual effort involved in obtaining these models. Extra-P requires a set of performance measurements as input, representing runs with different numbers of processes and problem sizes. The input sources used in this study encompass performance data obtained with PAPI, Score-P, and Threadspotter.

As a rule of thumb, we need to run measurements for at least five different configurations of each parameter we consider, requiring 25 measurements in the case of process count and problem size variation. Because we rely on highly reproducible hardware- and software counters, we need only one run per configuration and measurement tool–unless they can be applied simultaneously. The output of the generator is a set of human-readable functions, one for each instrumented program location and metric. Each function describes the evolution of the metrics as the number of processes and the problem size per process are changed.

Models are identified in an iterative process. For each parameter, we instantiate a number of model hypotheses of a certain size (up to a maximum number of terms n) according to the performance model normal form given in Equation 1 and select the winner through cross-validation. We start with

just one term, and increase the size of the hypotheses until we see no significant improvement in the quality of the result.

$$f(x) = \sum_{k=1}^{n} c_k \cdot x^{i_k} \cdot log_2^{j_k}(x) \tag{1}$$

We continue the process using the models representing individual parameters such as number of processes or problem size, which we successively combine after each step into a form corresponding to the expanded performance model normal form, as seen in Equation 2, substituting x_i for x.

$$f(x_1, \ldots, x_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l) \tag{2}$$

For example, the individual model of a function f for number of processes $f(p) = \log p$ and the one for problem size per process for the same function $f(n) = n^2$ could be combined either as $f(p, g) = \log p \cdot n^2$ or as $f(p, g) = \log p + n^2$. The details of model generation including references to the precise statistical methods can be found in [5], [6]. Different from those studies, however, we refrain from modeling execution time because execution-time models cannot be derived empirically for a platform that only exists as a design.

D. Locality modeling

To explain how requirements modeling can help understand the scaling behavior of memory locality, we provide a small example using the naïve matrix-matrix multiplication shown in Listing 1. For the sake of simplicity, we consider models as a function of the matrix size n alone.

In this example, we have three instruction groups, A, B, and C, representing the instructions that access the arrays with the same names, respectively. Both the reuse and stack distance of A is $2n$, as every time we iterate over k again we reuse the addresses of A. This means that there are only $2n$ accesses, n to addresses in A and n to addresses in B, between consecutive accesses to the same address in A. As each of these accesses is to a different memory location, the reuse and stack distances are identical. For C, neither stack nor reuse distance can be computed, as the memory locations are never touched again after their first access. For B, the reuse distance is $2n^2+n-1$, while the stack distance is n^2+2n-1. This difference stems from whether only unique accesses to A are considered or not.

The conclusion is that, as the size of the matrix increases, the reuse and stack distances of both A and B increase. This, in turn, means that it becomes increasingly likely that any given access will be to an address no longer residing in the cache. As long as the problem size is small enough that all matrices fit in the cache, performance will remain at a constant high. As the problem size grows, eventually, the matrices will no longer fit completely into the cache. As the stack distance of B is much higher than that of A, accesses to B will be the first to fail to find the data in the cache and the performance will degrade. Eventually, all accesses to B will be cache misses, and the performance will no longer degrade. Should the problem size increase enough that even some accesses to A turn into cache misses, the performance will degrade again, until virtually all memory accesses will be cache misses. The exact matrix size when performance starts to degrade depends on the size of the cache and the protocol used. However, the models of the stack distance, which we capture to characterize memory locality, on the one hand, combined with models of the total number of memory accesses on the other, are capable of discovering whether the pressure on the memory subsystem will ultimately begin to increase or not without knowledge of the hardware.

Listing 1: Naïve matrix-matrix multiplication $C = AB$.

```
void mmm(float *A, float *B, float *C, int n){
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            float v = 0.0f;
            for (int k = 0; k < n; k++)
                v += A[i*n + k] * B[k*n + j];
            C[i*n + j] = v;}}}
```

Listing 2: Blocked matrix-matrix multiplication $C = AB$. Matrix C is expected to be initialized with 0.

```
void mmm2(float *A,float *B, float *C,
     int n, int b){
 for (int jj=0; jj< n; jj += b){
  for (int kk=0; kk< n; kk += b){
   for (int i=0; i< n; i++) {
    for (int j=jj; j< min(jj+b, n); j++){
     float v=0.0f;
     for (int k=kk; k< min(kk+b, n); k++){
       v+= A[i*n + k] * B[k*n + j];}
     C[i*n + j]+=v;}}}}}
```

Now let us consider a blocked matrix-matrix multiplication, as implemented in Listing 2. Unlike the naïve implementation, we improve locality by multiplying and adding blocks of size b instead of individual matrix elements.

In this new example, we have the same three instruction groups, A, B, and C. Using this algorithm, the behavior is different depending on whether a new loop is starting or the code is iterating within a loop. The common case for A should be computing within the innermost loop if a sensible value for b is chosen. In this case, the stack distance of A is $2b+1$ and the reuse distance of A is $3b$. This means that there are only $3b$ accesses, b to each of the arrays A, B, and C, between consecutive accesses to the same address in A. Each of these accesses to A and B is to a different memory location, but all accesses to C are to the same address, hence the difference between stack and reuse distance. For B, we have no reuse within an iteration of the j-loop but b addresses are reused whenever we start such an iteration anew. Therefore, the reuse distance is $3b^2$. The stack distance is only $2b^2+b$ due to the reuse of addresses in C. For C, the most common stack and reuse distance is 2, as the same address is reused in every iteration of the innermost loop. This means that in the most common case for each of the instruction groups, the new

algorithm ensures that the locality does not depend on the size of the matrix. This will allow larger matrices to be computed without placing higher demands on the memory subsystem. As both implementations require the same number of floating-point operations and the same number of memory accesses, the blocked implementation is preferable.

As demonstrated using this simple example, our analysis discovers whether a given implementation is locality-preserving or not. If yes, we can assume that the number of main memory accesses scales with the number of retired load and store instructions, which we can easily measure. If not, changing the underlying algorithm is the most sensible option. To the best of our knowledge, this is the very first time an automatic method to model the scalability of memory locality has been proposed.

E. Co-design methodology

The key point of our method is to *guide the programmer to find application bottlenecks relative to an architecture as well as to guide the architect to find system bottlenecks that a given application would experience.* Our requirements models are functions of the number of MPI processes p and the input problem size n. To compare the requirements of an application on two different architectures, all we need to do is to calculate the application requirements using the values for p and n the application would use on these two systems. Below, we explain how to obtain these values.

First, we choose an appropriate number of processes. Although we concede that exceptions may exist, often encountered difficulties of exploiting more thread-level concurrency than a single socket provides lead us to the following rule of thumb: Each socket should run a separate and potentially multithreaded MPI process. In this way, we can simply set p to the number of available sockets. Nonetheless, deviating from this rule is absolutely possible. For example, systems with monolithic nodes like Blue Gene/Q can still be accurately treated by adapting p to the most common usage scenario.

Since a bigger input problem usually yields better parallel efficiency for a given number of processes, we strive to fully exploit the main memory available to a process. This means, after determining the number of processes p, we "inflate" the input problem until it completely occupies the available memory. Because memory is usually allocated per process, we find it more natural to consider the process-local memory for this purpose, also because process-local memory can be readily translated into process-local problem size using our process-local memory footprint model. We henceforth use the letter n to always denote the process-local problem size.

Once we have calculated the requirements of our application on two different systems A and B using the tuples (p_A, n_A) and (p_B, n_B) that we determine as recommended above, we can compare how the ratio of requirements changes as the application is ported from one system to the other. For example, let us assume the ratio between the number of floating-point operations and the number of bytes sent across the network on system A is r, while it is r/k on system B. This means that communication requirements will grow by a factor of k as the application is ported from A to B. This can be interpreted in two different ways: Either the network on system B should provide bandwidth that is a factor of k stronger relative to its floating-point performance or the application should be optimized to restore the original ratio on system B. A more elaborate example is presented in Section III-A.

Essentially, our co-design approach characterizes a system initially only in terms of the problem size and process count it can accommodate. We call this a *system skeleton*. The skeleton is then used to derive more specific requirements an application would expose the system to. Based on these requirements, the system designer can subsequently select further system properties such as the processor, the number of cores per processor, the integration of accelerators, the network, and the memory hierarchy or ask the application developer to optimize certain aspects of the application if the requirements can not be reasonably satisfied within the available technology or budget envelope.

In principle, our approach can map more than one application on a given system simultaneously. For example, we could assume that a system is shared between two applications in space according to a certain ratio as long as we can derive our model parameters p and n for each of them. However, since sharing is ultimately a matter of scientific priority, whose definition falls outside the scope of this paper, we narrow the discussion to the scenario where one application is granted exclusive access to the full machine to calculate the largest possible input problem. This objective corresponds to the idea of *heroic* runs [8].

III. Co-design Studies

We conducted two hypothetical co-design studies with five real applications. The first study, which is presented in Section III-A compares the benefits of three different system upgrades, while the second, which is presented in Section III-A, compares the advantages of three different exascale candidate systems. Before we delve into the details of these two studies, however, we first survey our test applications and their requirements below.

We chose five applications to represent a spectrum of different behaviors that are encountered on today's supercomputers: Kripke [9] and LULESH [10], two proxy apps created specifically towards exascale co-design, MILC [11], a QCD code, Relearn [12], a brain simulation, and icoFoam, a CFD solver from the widely used OpenFOAM package [13].

We gathered our measurements on two test systems: The first one is JUQUEEN, an IBM BlueGene/Q system at Jülich Supercomputing Centre with almost 500,000 cores. Each node features one PowerPC A2 processor with 16 cores running at 1.6 GHz. The second one is Lichtenberg, a Linux cluster at Technische Universität Darmstadt that consists of 706 nodes with two 8-core Intel Xeon E5-2670 processors on each node, running at 2.6 GHz. We obtained the measurements for LULESH, MILC, and Relearn on JUQUEEN and those for icoFOAM and Kripke on Lichtenberg. Because Threadspotter

TABLE II: Per-process requirements models. p denotes the number of processes and $n = N/p$ the problem size per process obtained by dividing the overall problem size N by the number of processes p, under the assumption that the overall problem size can be divided equally among all processes. For each metric, we show the terms with the largest impact on performance for both problem size per process and number of processes. The coefficient is the sum across the entire program, rounded to the nearest power of ten. We mark potential performance bottlenecks with a warning sign.

	Metric	Model	
Kripke	#Bytes used	$10^5 \cdot n$	
	#FLOP	$10^7 \cdot n$	
	#Bytes sent & received	$10^4 \cdot n$	
	#Loads & stores	$10^8 \cdot n + 10^5 \cdot n \cdot p$	⚠
	Stack distance	Constant	
LULESH	#Bytes used	$10^5 \cdot n \log n$	
	#FLOP	$10^5 \cdot n \log n \cdot p^{0.25} \log p$	⚠
	#Bytes sent & received	$10^3 \cdot n \cdot p^{0.25} \log p$	⚠
	#Loads & stores	$10^5 \cdot n \log n \cdot \log p$	
	Stack distance	Constant	
MILC	#Bytes used	$10^6 \cdot n$	
	#FLOP	$10^{10} \cdot n + 10^7 \cdot n \log p$	
	#Bytes sent & received	$10^4 \cdot Allreduce(p)$	
		$10^4 \cdot Bcast(p)$	
		$10^9 \cdot n$	
	#Loads & stores	$10^{11} + 10^8 \cdot n \log n + 10^5 \cdot p^{1.5}$	
	Stack distance	$10^5 \cdot n$	
Relearn	#Bytes used	$10^6 \cdot n^{0.5}$	
	#FLOP	$10^3 \cdot n \log n \cdot \log p + p$	
	#Bytes sent & received	$10^5 \cdot Allreduce(p)$	
		$10 \cdot Alltoall(p)$	
		$10 \cdot n$	
	#Loads & stores	$10^6 \cdot n \log n + 10^5 \cdot p \log p$	
	Stack distance	Constant	
icoFoam	#Bytes used	$10^3 \cdot n + 10^2 \cdot p \log p$	⚠
	#FLOP	$10^8 \cdot n^{1.5} \cdot p^{0.5}$	⚠
	#Bytes sent & received	$n^{0.5} \cdot Allreduce(p)$	⚠
		$p^{0.5} \log p$	⚠
		$n \cdot p^{0.375}$	⚠
	#Loads & stores	$10^8 \cdot n \log n \cdot p^{0.5} \log p$	⚠
	Stack distance	Constant	

does not support the processor of JUQUEEN, we measured stack distance for all applications on Lichtenberg. Already this showcases one advantage of our approach. Because the metrics we collect are architecture independent, we can easily overcome the deficiencies of the measurement infrastructure on one system by choosing another.

In our experiments, we varied the number of MPI processes and the problem size per process. Some of the metrics we analyze can be gathered at different levels of granularity. A fine granularity is useful to pinpoint performance bottlenecks in applications. For the current analysis however, we are interested in the performance of the application as a whole and we therefore wish to summarize the models obtained. The memory footprint, the number of floating-point operations, and the number of loads and stores are gathered by examining the

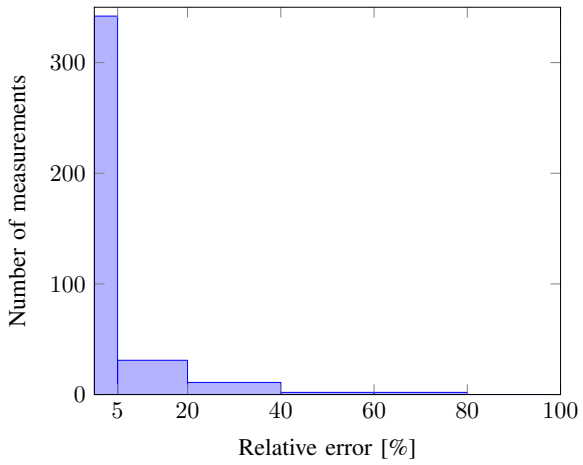

Fig. 3: Measurements classified by percentile relative error over all generated models.

entire application monolithically. Requirements for communication and memory locality are obtained at the granularity of function calls and instruction groups, respectively. For each application, we selected all models with the fastest growing requirements for each of the two model parameters p and n, added all coefficients for these models, and rounded them to the nearest power of ten. We generated models considering polynomial and logarithmic exponents. The polynomial exponents take values between 0 and 3, including all fractions of the types $\frac{i}{8}$ and $\frac{i}{3}$. For logarithms, we used the exponents $\{0; 0.5; 1; 1.5; 2\}$.

The resulting requirements models of our five applications are presented in Table II. To assess the model quality, the histogram shown in Figure 3 classifies each measurement that was used to generate a model according to the relative error of the generated model. The overwhelming majority (88%) of measurements points are well explained by our models and have relative errors smaller than 5%, and most of the remaining ones (8%) still have relative errors smaller than 20%. We therefore claim that the models we generate are overall adequate to serve as a basis for the co-design process. Below, we briefly discuss the requirements of each application individually.

Kripke is a 3D Sn particle transport code and implements an asynchronous MPI-based parallel sweep algorithm. A major goal of Kripke is the evaluation of programming models, data layouts, and sweep algorithms in terms of their performance impact. The problem size per process is defined as the simulated volume per process. As expected from a exascale proxy app, Kripke should scale reasonably well to any number of processes and the problem size per process will remain configurable without incurring significant performance losses. Only the number of loads and stores shows a multiplicative effect of problem size and process count and might lead to a slowdown.

LULESH is also a a widely studied proxy application in DOE co-design efforts for exascale which calculates simplified

3D Lagrangian hydrodynamics on an unstructured mesh. The problem size per process is defined as the simulated volume per process. The growth rates of all requirements with respect to both problem size and process count are very close to ideal. With the current implementation, the multiplicative effect process count and problem size per process have on computation and communication for LULESH is a small obstacle in tailoring and scaling the application to run on different systems. The growth rates are slow enough to limit these issues at anything except the most extreme scales.

MILC – MIMD Lattice Computation – is a set of codes for studying quantum chromodynamics (QCD) via parallel simulations of the SU(3) lattice gauge theory on a four-dimensional lattice. MILC is a highly scalable application, which consumes a major fraction of the CPU cycles in US DOE and NSF computing centers. We analyzed the application MILC/su3_rmd. Its runtime was modeled in [11], [5]. However, no requirement models exist for MILC. For MILC, the problem size per process is defined as the size of the lattice. MILC should be able to fit most target systems without significant performance loss. The only requirement than can be optimized is memory access. If the memory locality was improved, increasing the problem size per process would be possible without losing performance.

Relearn simulates the dynamics of the connectome in the brain, that is, how connections between individual neurons are formed and deleted. This is also called structural plasticity. The problem size parameter in our tests defines the number of neurons per process to be simulated. According to our empirical findings, memory consumption increases with the square root of the problem size. The theoretical expectation is a linear function of n plus a much weaker and possibly negligible linear function of p, although it only reflects the programmer's view and does not necessarily capture implementation details at lower levels of the software stack. A linear model was among the best model candidates, but the chosen model fits the measured data slightly better. For reasons of consistency with the chosen approach we will therefore use the empirically determined model throughout this work. Either way, Relearn has no serious issues in scaling to any number of processes. It will be able to vary the domain size per process to fit multiple target systems without significant performance loss and can accommodate systems where memory consumption is at a premium.

IcoFoam is a solver in the widely used open-source computational-fluid-dynamics package OpenFOAM. OpenFOAM, developed by ESI/OpenCFD, is a non-monolithic library encompassing over 80 flow solvers that supports numerical simulations of a broad variety of continuum models. We analyzed the unmodified icoFoam executable from the demonstration instance of OpenFOAM (development version from April 2017 from openfoam.org). This flow solver implements a method suitable for the incompressible flow of a Newtonian fluid under isothermal conditions. We applied the solver to a two-dimensional test case, namely the well-known lid-driven cavity [14]. There, the problem size is defined as the number of computational cells per process. Almost everything including memory consumption and memory access, communication, and computation limit the scalability of icoFoam. The severity and multitude of performance issues suggest that a different approach is required as a whole.

TABLE III: Process count and memory per process available to applications for three different system upgrade scenarios.

System upgrade	Process count	Memory per process
A: Double the racks	$p' = 2 \cdot p$	$m' = m$
B: Double the sockets	$p' = 2 \cdot p$	$m' = 0.5 \cdot m$
C: Double the memory	$p' = p$	$m' = 2 \cdot m$

A. System upgrade

Now, we use the requirements models listed in Table II to answer the first co-design question that we consider in this study: *"Given a large system defined such that the application equally exhausts all available resources, which of the possible upgrades would benefit the application most?"* Possible upgrades we consider are (A) doubling the entire system, (B) doubling the number of processor sockets per node and leaving everything else constant, and (C) doubling the memory and leaving everything else constant. These upgrades and how the resources that are available to the applications change on the new systems are summarized in Table III. The combination of possible upgrades can be expanded to encompass further realistic scenarios that are considered by system architects.

Before we summarize our results for all applications, however, we illustrate our workflow, and in particular our co-design methodology presented in Section II-E, step by step using LULESH as a walk-through example. After that, we summarize the results of this process for all applications and discuss our findings. To exemplify the process of determining how an application would respond to a system upgrade, we choose system upgrade A, that is, doubling the racks of the system. We enumerate and describe the different steps of our scaling method in Table IV.

The requirements of LULESH are listed at the top of the table as part of Step I. Following this process, we can now draw conclusions regarding system utilization, requirements balance, and usefulness of a particular upgrade. The ratios between new and old problem sizes indicate how the largest problem size that can be solved changes, both per process and overall. The ratios between new and old requirements indicate which system components will experience an increased load relative to other components.

The requirements of LULESH can be expressed as the product of single-parameter functions that either depend the problem size per process or the number of processes. When doubling the racks, only the value of p changes, and in this particular case, all terms depending on n can be reduced when determining the ratios of the changing requirements. This means that the these ratios are valid regardless of the problem size per process. This will not be generally true as it depends on the specific relative upgrade. That the number of

processes affects computation and communication means that these requirements increase slightly. Luckily, computation and communication only increase by 20% and will therefore allow LULESH to solve an overall problem twice as large with only a small performance degradation.

Using the five applications above, we now apply the previously illustrated workflow to analyze the benefits and drawbacks of different system upgrades. We use the requirement models determined for each application individually, as well as the upgrades listed in Table III to determine the new problem sizes per process for each application, assuming that all processors a system provides are used. We then determine the new requirements for computation, communication, and memory access. Our comparative analysis is numerically summarized in Table V. We consider an optimistic linear relation between problem size per process and requirements as a baseline for scalability. For example, if we double the racks we wish that the total problem size that can be solved should double, too, but that the requirements per process remain the same. This simplifying assumption will not be generally true, but provides a notion of desirable behavior for our discussion. Apart from MILC, no other application has shown any change in memory locality with respect to process count and problem size per process. We therefore focus on the total number of load and store instructions as the primary memory-access metric in these cases.

When analyzing Table V, it becomes obvious that the most important parameter is the problem size per process, as it determines all other requirements and how well the stated goal of trying to perform heroic runs is met. When considering the computation, communication, and memory-access requirements per process, these should ideally follow the same behavior as the problem size per process. None of the analyzed applications reach this ideal, although Kripke and MILC come close with only one and two deviations, respectively, which one can see by tracing their columns in Table V.

Finally, we summarize how the applications benefit from the proposed upgrades: Kripke benefits equally from doubling the memory or doubling the sockets, and slightly less from

TABLE IV: Workflow for determining the requirements of LULESH after doubling the number of racks (upgrade A). The upgrade is relative, therefore we can omit model coefficients.

I: Create requirement models for memory footprint, communication, computation, and memory access.

Metric	Process scaling and problem scaling
#FLOP	$n \log n \cdot p^{0.25} \log p$
#Bytes sent & recv.	$n \cdot p^{0.25} \log p$
#Loads & stores	$n \log n \cdot \log p$
#Bytes used	$n \log n$

II: Determine the new maximum number of processes and new memory available per process that the upgraded system supports.

Configuration parameter	Old	New
Processes count	p	$p' = 2p$
Memory	m	$m' = m$

III: Determine the new memory footprint requirement per process if all processors are used.

Metric	Old	New
#Bytes used	$n \log n$	$n' \log n'$

IV: Determine the new problem size per process such that the memory footprint equals the memory available to each process and compute the new overall problem size.

Metric	Old	New	Ratio
Problem size per proc.	$n \log n = n' \log n' = m$		1
Overall problem size	$p \cdot n$	$p' \cdot n'$	2

V: Determine the new requirements for computation, communication, and memory access.

Metric	Old	New	Ratio
#FLOP	$p^{0.25} \log p$	$(2p)^{0.25} \log 2p$	≈ 1.2
#Bytes sent & recv.	$p^{0.25} \log p$	$(2p)^{0.25} \log 2p$	≈ 1.2
#Loads & stores	$\log p$	$\log 2p$	≈ 1

TABLE V: System upgrade comparison. We show how problem size and the corresponding requirements of an application change in response to each upgrade. While the ability to solve large overall problems is desirable, the per-process requirements for computation, communication and memory access should be as low as possible. The base-line expectation, which assumes a linear relation between requirements and problem size per process, is provided in the rightmost column for each metric.

Ratios \ Apps.	Kripke	LULESH	MILC	Relearn	icoFoam	**Baseline**
System upgrade A: Double the racks						
Problem size per process	1	1	1	1	0.5	**1**
Overall problem size	2	2	2	2	1	**2**
Computation	1	1.2	1	1	0.5	**1**
Communication	1	1.2	1	1	0.7	**1**
Memory access	2	1.2	2.8	2	0.7	**1**
System upgrade B: Double the sockets						
Problem size per process	0.5	0.5	0.5	0.3	0.3	**0.5**
Overall problem size	1	1	1	0.5	0.6	**1**
Computation	0.5	0.6	0.5	0.3	0.2	**0.5**
Communication	0.5	0.6	0.5	0.3	0.3	**0.5**
Memory access	0.5	1	1.4	1	0.5	**0.5**
System upgrade C: Double the memory						
Problem size per process	2	1.4	2	4	1.4	**2**
Overall problem size	2	1.4	2	4	1.4	**2**
Computation	2	1.4	2	4	1.7	**2**
Communication	2	1.4	2	4	1.4	**2**
Memory access	2	1.4	2	4	1.4	**2**

doubling the racks. LULESH draws the biggest advantage from doubling the racks and the least from doubling the memory. MILC and Relearn profit most from doubling the memory and least from doubling the sockets. The last application studied, icoFoam, would benefit only from doubling the memory. Consequently, there is no upgrade which is best for all applications, but overall doubling the memory or the racks would help most applications the most.

B. System design

The second question we address is: *"How would the performance change when an application is ported between different proposed exascale systems?"* This question sheds light on how differences in system design affect the studied applications by looking at absolute numbers rather than relative differences. For this, we investigate how the applications studied would map to potential exascale straw-man systems. Rather than using relative upgrades as in the previous section, where we assumed that certain characteristics of an existing system are doubled, we now focus on how our method works when applied to absolute values for system characteristics such as flop/s per processor. There are a number of hardware architecture directions suggested to reach exascale. Major differences between the approaches lie in their ratio of nodes to processors to flop/s per processor, which combined are supposed to reach 1 exaflop/s. By processor, we define a computational unit designed to run a process (potentially multi-threaded). Possible design options for such systems are presented in Table VI and summarized below:

- A **massively parallel system** that consists of nodes with many but weak processors
- A **vectorized system** that consists of nodes with few but very powerful processors
- A **hybrid system** with a large number of moderately powerful processors per node

For this study, we assume a total memory per system of 10 PB, divided equally among all processors. This value is consistent with the ratios of flop/s to memory of current top supercomputing systems in the world. The total memory, I/O, and network resources can also vary, but more likely as a function of the available funds and not to satisfy a certain ratio to other resources. A more detailed analysis where more system characteristics would vary is certainly possible, but would not be qualitatively different. We therefore focus only on the computational requirement and memory footprint relative to the problem size per process and the number of processes are taken into consideration. We determine the number of processes for each of the systems by multiplying the node and processor counts, as we want to have access to the full exaflop/s. For each application we can then determine the problem size per process that would consume all the memory available to a process. Knowing the problem size per process and the number or processes, we determine the overall problem size for each application. The results of this workflow, which is similar to the one presented in Table IV, are presented in Table VII.

TABLE VI: Characteristics of three exascale straw-man systems.

Metric	Massively parallel	Vector	Hybrid
Nodes	$2 \cdot 10^4$	$5 \cdot 10^4$	10^4
Processors	$2 \cdot 10^9$	$5 \cdot 10^7$	10^8
Processors per node	10^5	10^3	10^4
Memory per processor	$5 \cdot 10^6$	$2 \cdot 10^8$	10^8
Flop/s per processor	$5 \cdot 10^8$	$2 \cdot 10^{10}$	10^{10}

TABLE VII: Maximum overall problem size for selected applications and time each application needs to solve the same benchmark problem on different exascale straw-man systems described in Table VI, assuming perfect parallelization. Following a workflow similar to Table IV, we determine the values using the requirement models from Table II.

	Metric	Massively parallel	Vector	Hybrid
Kripke	Maximum overall problem size	10^{10}	10^{10}	10^{10}
	Minimum wall time for benchmark problem [s]	0.1	0.1	0.1
LULESH	Maximum overall problem size	$3.9 \cdot 10^{10}$	$1.7 \cdot 10^{10}$	$1.9 \cdot 10^{10}$
	Minimum wall time for benchmark problem [s]	40	21.5	33
MILC	Maximum overall problem size	10^{10}	10^{10}	10^{10}
	Minimum wall time for benchmark problem [s]	10^2	10^2	10^2
Relearn	Maximum overall problem size	$5 \cdot 10^{10}$	$4 \cdot 10^{12}$	10^{12}
	Minimum wall time for benchmark problem [s]	4	0.02	0.2

icoFoam is notably absent in Table VII, as the number of processes adversely affects the memory required per process. This unfortunately means that the studied instance of the code cannot fully utilize any of the three systems, as the memory requirement regardless of problem size per process is larger than what is available if all processors are used. While it would be possible to run this code on a smaller subset of processors, that is not the focus of our study.

We discover that for Kripke and MILC the different system types do not affect the largest overall problem size that can be solved. That is because any difference in the ratio between process count and problem size per process can be offset by configuring the application appropriately. The situation is different for the other applications, as the ratio p/n between process count and problem size per process is more relevant to the required memory. Relearn can solve much larger overall problems on a system with fewer but stronger processors, while LULESH can solve the largest problem on the massively parallel system.

We further analyze on which system a given problem can be solved faster, by taking the biggest overall problem size

of each application that can be solved on all systems, and change the problem size per process such that each system solves the same problem. We still use all available processors to have access to all computational resources each system provides. We then use the problem size per process and the number of processes to determine the number of floating-point operations required per process. After dividing this requirement by the floating-point rate offered by the processor, we can estimate a lower bound of the runtime this computation takes. The lower bound is based on the simplifying assumption that parallelization is perfect and no communication overhead exists. This lower bound can be useful in comparing how the architectures will affect application performance as the difference of multiple orders of magnitude for some applications is unlikely to be offset by the communication overhead. Kripke and MILC take the same time to solve the problem on each system. However, both Relearn and LULESH benefit more from a high ratio and would perform better on the vectorized system. To shift the lower bound closer to more realistic runtimes, we need to take other requirements such as communication into account, which is feasible as long as the system designer can specify the rates at which the hardware can satisfy them.

A possible optimization for LULESH could be changing the algorithms such that the effects of problem size per process and process count are additive rather than multiplicative with respect to the different requirements: $\#FLOP = 10^5 \cdot n \cdot \log n \quad + \quad p^{0.25} \cdot \log p$. In the example from Table VII, this would improve the overall time to solution by approximately three orders of magnitude on each system, to less than 0.1 compared to between 20 and 40 seconds. It would also change how LULESH performs on the different systems, obtaining the best results on the massively parallel system as opposed to the currently favored vectorized system.

The recommended course of action beyond improving the applications is to experiment with a small number of prototype nodes of the kind to be employed in the system, and determine the actual rates at which the requirements can be satisfied. Requirements other than computation such as memory access and network communication must be considered. For example, similarly to our analysis of potential exascale system candidates with different nodes to processors to flop/s ratios, an analysis of the network requirements taking network bandwidth, latency, and topology into account can be performed. However, when considering a small number of nodes as opposed to an entire rack or multiple racks, network communication may differ qualitatively.

Nevertheless, already before any hardware prototype becomes available, will our co-design methodology provide hints as to which basic system design will be most or least profitable for a given workload. In our study, the vectorized system provides the best performance potential for the benchmark problem across all applications. On the other hand, when looking at the possible maximum problem size, LULESH would favor the massively parallel system, while Relearn would prefer the vectorized system, although a problem size of $4 \cdot 10^{12}$ is already more than needed. After all, the human brain has only about 10^{11} neurons. The decision which alternative to choose, that is which application(s) to prioritize, however, is beyond the scope of this paper.

IV. Related Work

Performance models have already been used to predict code performance on different architectures, leveraging tool chains with varying degrees of automation [15], [16], [17]. In contrast, our work focuses on the extrapolation of requirements rather than exact performance to guide higher-level decisions in exascale application and system design. Carrington et al. project node-level requirements using simple linear, logarithmic, exponential, or constant regression [18]. Our method goes beyond that by combining more complex functions to characterize system-wide requirements. In the widest sense, the roofline model [1], [19] can also be considered as an interpretable requirements model. It is, however, mostly designed as an application optimization tool and thus not easily applicable to co-design.

Gahvari and Gropp [20] as well as Bhatele et al. [21] studied the theoretical feasibility of several computational algorithms on hypothetical exascale machines introduced in a previous study [22]. They show bounds on network requirements in terms of latency and bandwidth that would have to be satisfied in order to solve these problems. While extremely valuable, their studies are purely theoretical and not based on real applications. With our method, we enable similar studies for actual code bases.

Many co-design approaches rely on application and architecture simulation. Such simulations exist at numerous granularities, ranging from cycle-accurate [23], [24] to coarse and model-driven [25]. Direct execution approaches often show severe memory limitations [26], especially in the exascale range. The main drawback of such simulations are the enormous resources required to run them. Furthermore, without additional human interpretation simulation results on their own provide little insight into application scaling. Our requirements models need no resources beyond the small scale measurements required to produce them, enabling extreme-scale predictions at very low cost. Moreover, they are intuitive in that they allow direct statements such as "the required network bandwidth grows logarithmically with the system size".

V. Conclusion

In this work, we introduce a quick and simple automatic back-of-the-envelope technique to generate application-centric requirements models for parallel applications. The workflow we propose leverages these models to enable system designers and application developers to ponder various upgrade and design options. We characterize performance in terms of relative requirement changes - from one system or one application to another. This pattern indeed matches the common case, where an initial version of an application running on an initial system already exists. And even if no such system exists, our

approach can successfully help compare design options. The main advantage of our approach in relation to architecture-specific performance models, which are traditionally hard and laborious to produce with high accuracy, however, is the small effort on the one hand and the low complexity of the models on the other, facilitating quick insights at low cost—easily at the scale of an entire compute-center workload.

ACKNOWLEDGMENT

This work has been supported by the German Research Foundation (DFG) through the Program *Performance Engineering for Scientific Software* and the ExtraPeak project, by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH16008D, and by the US Department of Energy under Grant No. DE-SC0015524. The authors gratefully acknowledge the computing time granted through JARA-HPC on the supercomputer JUQUEEN at Forschungszentrum Jülich, as well as the opportunity to conduct part of this study on the Lichtenberg high-performance computer of TU Darmstadt.

REFERENCES

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[2] D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. S. Shende, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A unified performance measurement system for petascale applications," in *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, Gauß-Allianz. Springer, 2012, pp. 85–97.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.

[4] E. Berg, "Efficient and flexible characterization of data locality through native execution sampling," Ph.D. dissertation, Department of Information Technology, Uppsala University, Nov. 2005.

[5] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proc. of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC13),Denver, Colorado, USA*, Nov. 2013.

[6] A. Calotoiu, D. Beckingsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf, "Fast multi-parameter performance modeling," in *Proc. of the 2016 IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan*. IEEE Computer Society, Sep. 2016, pp. 1–10.

[7] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 215 – 224.

[8] S. Byna, A. Uselton, Prabhat, D. Knaak, , and Y. He, "Trillion Particles, 120,000 cores, and 350 TBs: Lessons Learned from a Hero I/O Run on Hopper," 2013. [Online]. Available: http://sdav-scidac.org/images/publications/Byn2013a/cug13.pdf

[9] A. J. Kunen, "Kripke - user manual v1.0," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-SM-658558, August 2014.

[10] I. Karlin, J. Keasler, and J. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2013.

[11] G. Bauer, S. Gottlieb, and T. Hoefler, "Performance modeling and comparative analysis of the MILC lattice QCD application su3_rmd," in *Proc. of CCGrid*, May 2012.

[12] S. Rinke, M. Butz-Ostendorf, M.-A. Hermanns, M. Naveau, and F. Wolf, "A scalable algorithm for simulating the structural plasticity of the brain," *Journal of Parallel and Distributed Computing*, 2018.

[13] H. Jasak and A. Jemcov, "OpenFOAM: A C++ library for complex physics simulations," in *International Workshop on Coupled Methods in Numerical Dynamics, IUC*, 2007, pp. 1–20.

[14] U. Ghia, K. N. Ghia, and C. T. Shin, "High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method," *Journal of Computational Physics*, vol. 48, pp. 387–411, 1982.

[15] L. Carrington, A. Snavely, and N. Wolter, "A performance prediction framework for scientific applications." *Future Generation Comp. Syst.*, vol. 22, no. 3, pp. 336–346, 2006.

[16] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," *SIGMETRICS Performance Eval. Review*, vol. 32, no. 1, pp. 2–13, June 2004.

[17] L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Proc. of the ACM/IEEE Conference on Supercomputing*, ser. (SC '05). IEEE Computer Society, 2005, p. 40.

[18] L. Carrington, M. Laurenzano, and A. Tiwari, "Characterizing large-scale hpc applications through trace extrapolation," *Parallel Processing Letters*, vol. 23, no. 4, 2013.

[19] B. Norris, W. Spear, and A. Malony, "Performance analysis of applications in the context of architectural rooflines," in *Proc. of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA: ACM, 2017, pp. 345–348.

[20] H. Gahvari and W. Gropp, "An introductory exascale feasibility study for FFTs and multigrid," in *Proc. of the IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–9.

[21] A. Bhatele, P. Jetley, H. Gahvari, L. Wesolowski, W. D. Gropp, and L. V. Kalé, "Architectural constraints to attain 1 exaflop/s for three scientific application classes," in *Proc. of the IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. IEEE, 2011, pp. 80–91.

[22] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J. Andre, D. Barkai, J. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, C. Xuebin, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. J., Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011.

[23] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang, "Mambo: a full system simulator for the PowerPC architecture," *SIGMETRICS Performance Eval. Review*, vol. 31, pp. 8–12, March 2004.

[24] A. F. Rodrigues, R. C. Murphy, P. Kogge, and K. D. Underwood, "The structural simulation toolkit: exploring novel architectures," in *Proc. of the ACM/IEEE Conference on Supercomputing*, ser. (SC '06). ACM, 2006.

[25] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim: simulating large-scale applications in the LogGOPS model," in *Proc. of the 19th ACM Intl. Symposium on High Performance Distributed Computing*, ser. (HPDC). ACM, 2010, pp. 597–604.

[26] C. Mei, "A preliminary investigation of emulating applications that use petabytes of memory on petascale machines," Master's thesis, University of Illinois at Urbana-Champaign, 2007.