In order to guarantee high-performance, productive computing on extreme-scale supercomputers, a few guidelines should be adhered to. First, we should agree on a common, yet parameterizable, abstract machine model. Today, we see two camps of abstract machine models as exemplified by the shared memory extension of the Von Neumann CPU architectures, and the explicitly hierarchical/heterogeneous machine model seen in various GPU-like accelerators. The former has allowed us to easily port code to a myriad of CPU architectures including x86 CPUs (Xeon or Opteron), three generations of Blue Gene, SPARC, MIPS, various vector processors, and even Intel’s latest Xeon Phi manycore processor. The explicitly hierarchical/heterogeneous machine model is not a new phenomenon, but rather an approach that appears periodically when the performance, energy, or design cost of virtualizing the hierarchy or heterogeneity becomes prohibitive given the current process technology or market constraints. Unfortunately, the presence of a second (or third) abstract machine model demands we contemplate two dramatically different implementations. We are thus presented with the unpalatable choice of either destroying productivity (writing a second implementation from scratch) or ignoring one class of machine (most likely the less productive and less portable).

Second, we must have common, standards-based programming models (or hierarchy of programming models). The presence of at least one common programming model across all architectures ensures we can write one implementation of a program and have it run everywhere. Despite the evolution of architectures to include heterogeneous processing or hierarchical memories, it is imperative architects, compiler writers, and runtime software engineers work together to preserve a common abstract machine model and programming model that scientists may target.

Although the presence of a common abstract machine model and programming model may provide portability and possibly (weak) scalability by nodes, it is no guarantee of performance portability (i.e. efficient use of a processor). In fact, determining optimality is an immense challenge. As part of the SciDAC Institute for Sustained Performance, Energy, and Resilience, we are developing a Roofline Toolkit that will allow programmers to easily analyze their code on a routine-by-routine basis in order to understand its performance potential. The toolkit is based on a generalization of the Roofline model which uses bound and bottleneck analysis to provide a performance bound for a loop nest by examining its (DRAM) communication and computational requirements. The model has been refined to reflect the complexity of in-core performance was well as the cache hierarchy and their individual bandwidths. We believe the model can be extended and automated so that it can deal with arbitrary code and explicitly programmed memory architectures (i.e. local stores or device memories on accelerators).

One expects the combination of a common abstract machine model and programming model will lead to suboptimal performance for some routines on some machines. If used in conjunction with performance monitoring tools, we believe the toolkit will allow one to identify which routines are underperforming. Knowing which thousand lines of code in a multi-million-line application should be written in order to improve performance is a key to performance portability. That is, it is made clear where programmers should focus their efforts.