FLAME: Graph-based Hardware Representations for Rapid and Precise Performance Modeling

Mehmet E Belviranli
Oak Ridge National Laboratory
belviranlime@ornl.gov

Jeffrey S Vetter
Oak Ridge National Laboratory
vetter@ornl.gov

***Abstract*—The slowdown of Moore's law has caused an escalation in architectural diversity over the last decade, and agile development of domain-specific heterogeneous chips is becoming a high priority. However, this agile development must also consider portable programming environments and other architectural constraints in the system design. More importantly, understanding the role of each component in an end-to-end system design is important to both architects and application developers and must include metrics like power, performance, space, cost, and reliability. Being able to quickly and precisely characterize the needs of an application in the early stages of hardware design is an essential step toward converging on the primary components of these increasingly heterogeneous platforms. In this paper, we introduce *FLAME*, a graph-based machine representation to flexibly model a given hardware design at any desired resolution while providing the ability to refine specific components along the hierarchy. *FLAME* allows each processing unit in the system to declare its specific capabilities and enables higher level elements to reuse and leverage these declarations to form more complex system topologies. Applications are characterized with the Aspen application model; each component has the ability to report its characteristic behavior for a given application model against a supported metric. We demonstrate the feasibility of *FLAME* for several workloads via multi-core machine representations on different levels abstraction.**

I. Introduction

The slowdown of Moore's law has caused an escalation in architectural diversity over the last decade, and agile development of domain-specific heterogeneous chips tailored for specific application workloads is becoming a high priority. This process will require the close coordination of applications, software, and hardware so that architects can understand the role of each component in an end-to-end system design in terms of metrics like power, performance, space, cost, and reliability. Moreover, this agile design must also consider portable programming environments and other architectural constraints in the system design. Being able to quickly and precisely characterize the needs of an application in the early stages of hardware design is an essential step toward converging on the primary components of these increasingly heterogeneous platforms.

As the processing units become highly specialized and the architectural diversity within the system increases, the design space increases exponentially, and traditional performance prediction techniques fall short in various qualities. For example, even before simulation, an architect may want to quickly estimate the power, performance, and cost of a range of components in domain-specific system on a chip (DSSoC) against a broad range of workloads. Once the architect has narrowed the design space, they can then represent the prototype design with intricate specification in cycle-accurate simulators. Similarly, functional emulation can proceed when there are abstracted or unavailable software modules in the application. Most simulation and emulation-based approaches are also known to have limited scale, falling far short of the scale of contemporary salable systems in high performance computing (HPC) and enterprise facilities. Furthermore, simulation and emulation require detailed representations of each application that may need to be optimized for each specific architectural option. For example, ideally, the compiler toolchain must be able to lower specific application operations to new hardware features being emulated like a new *sqrt* instruction or a deeper write queue.

Analytical models, on the other hand, give application developers more flexibility and speed in performance prediction, including varying levels of abstraction and sensitivity analysis, at the expense of accuracy. These models reduce the machine-specific performance factors into a set of parameters that can be embedded into equations representing the complexity of computation and other important resources, such as memory. For scientific applications, these equations are tailored to fit the specifics of the algorithm and the hardware target. Traditionally, to generalize and reuse the computational and communication hardware characteristics that are common across commodity systems, the community has used *structured analytical model*s like logP [1], BSP [2], and roofline [3]. Domain-specific languages (DSL) for performance modeling [4] further reduce the complexity of dealing directly with analytical equations by allowing the creation of composable representations that are interchangeable across different applications and architectures. Among the most notable, Aspen [5] (Abstract Scalable Performance Engineering Notation) defines a formal grammar to describe program behavior (i.e., *application model*) and hardware organization (i.e., *machine model*). Using these two models, Aspen lets developers quickly query and analyze the performance against various metrics and scenarios.

While DSL-based modeling frameworks like Aspen allow the creation of reusable and portable analytical models that make performance analysis and prediction easier for many scientific applications, these tools are falling behind on the rising complexity of the architectures. Similar to simulators,

hardware models in these approaches are usually restricted to following specific architectural patterns [6] and lack the extensibility and adaptability required to support rapid and experimental design of emerging architectures.

To address this problem, we believe that a new hardware representation is necessary. It should (a) allow building scalable models by *abstract*ing and reusing existing components to reduce the overall complexity while providing more precise predictions for larger systems and (b) describe the architecture in a *flexible* and *expressive* way without decreasing the modeling simplicity, especially during the early application and hardware design process.

In this paper, we present *FLAME* (*FL*exible *A*bstract *M*achine *E*xpression), a graph-based machine representation methodology to support a wide range of architectural organizations in analytical models while allowing developers to define specific components as abstractions or refine the level of detail for a desired hardware part as necessary. *FLAME* relies on a connected multi-layer graph topology [7] to describe the hierarchical relation between multiple levels of abstraction for a component and provides a formal definition to allow automated tools to create performance predictions for important metrics. The ultimate goal of *FLAME* is to be able to "precisely" identify the effects of proposed components at various levels of an end-to-end system analysis.

Specifically, we make the following contributions.

- We present the design and implementation of *FLAME*: a graph-based machine representation technique to increase the expressiveness, precision, and flexibility of structured analytical modeling schemes.
- We provide a description for the multi-layer graph representation and design an accompanying class hierarchy to represent the hardware components as nodes and edges of the graph; and
- We demonstrate the capabilities of *FLAME* by predicting the performance of scientific kernels on a multi-core CPU modeled by *FLAME*.

II. Background: Performance Modeling

Application developers and architects rely on various prediction techniques and tools to perform design decisions, and each of them has specific advantages over another. Cycle-accurate simulators, such as GEM5, SimpleScalar, and RSIM, have been widely used to mimic the projected hardware behavior with the ability to represent fine-granular architectural characteristics at the expense of very slow execution times. Functional simulators, on the other hand, allow faster emulation of instruction set architectures (ISAs) on non-native devices; however, they fail to provide timing information. Hardware emulators rely on reconfigurable devices such as FPGAs to speed up the simulation process and provide cycle-accurate predictions but come at the cost of a tedious synthesis process. Both simulation- and emulation-based approaches are usually focused on modeling a single node and require extensive knowledge of every component being modeled.

```
param n = 8192 // input size
param word = 16 // double complex
param a = 6.3 // constant for cache miss calculation
param P = 16 // number of threads
param Z = 2.1 * mega
data fftVolume [n * word]

kernel fft {
 execute [P] "myloop"{
   flops [(5 * (n/P) * log2(n/P))] as dp, simd, fmad
   loads [((n/P*word)) * a * max(1, log((n/P)*word)/
    log(Z))] from fftVolume
 }
}
```

Listing 1: Aspen application model example for 1D FFT.

Analytical modeling is situated on the other extreme end of the scale and targets to provide performance insight via application and algorithm characterization. In this type of performance prediction, the constituent parts of the application are parameterized, and the critical path is represented with equations that are dependent on algorithm-specific inputs and hardware metrics. While many scientific applications build their own custom performance models [8], [9], [10], structural analytical models have been commonly used to develop reusable techniques and abstractions. Numerous studies have been proposed to synthesize modeling expressions [11], describe hardware characteristics [12], and create DSL-based application and machine representations [4] and design tools to automate the process [13]. In this study we will be utilizing the roofline model [3] and ASPEN [5] to build upon and evaluate our work.

A. Roofline Model

In contrast to creating complex analytical representations, the roofline model [3] helps developers identify the bounds for compute and memory bottlenecks on a given architecture. Instead of predicting the performance, roofline describes the maximum throughput (i.e., GFlops/sec) of an application as a function of its arithmetic intensity (i.e., Flops/byte).

$$\text{GFlops/sec} = min \begin{cases} \text{Peak FLOPS} \\ \text{Mem. Bandwidth} \times \text{Intensity} \end{cases} \quad (1)$$

The roofline model, as shown in Equation 1, breaks the architectural behavior into two regions: (a) the linear curve where the performance is bounded by memory and (b) the flat part where the computational capacity limits the attainable throughput. Depending on application's arithmetic intensity, the effective performance falls under one of these regions. The (a) region will also shift toward or away from the (b) region, depending on how well the application optimizes memory accesses.

B. Aspen

Aspen [5] is a popular DSL for structured analytical performance modeling. The Aspen application model (AAM) supports expressions to represent compute and memory characteristics, input parameters, data structures, parallel regions, kernel declarations, repetitive sections, and communicational

```
machine Keeneland {
  node [120] sl390_node
  interconnect qdrInfiniband
}
node sl390_node {
  socket [2] intel_xeon_x5660
  socket [3] nvidia_m2090
}
socket intel_xeon_e5_2698_v3 {
  core [16] haswellCore
  memory ddr4
  cache haswellCache
}
core haswellCore {
  param coreClock = 2.3 * giga
  param issueRate = coreClock * 2
  resource cycles(number) [number/coreClock]
  resource flops(number) [number/issueRate]
    with dp [base * 2],
      simd [base / 8],
      fmad [base / 2]
}
cache haswellCache {
  property capacity [40 * mega]
}
```

Listing 2: Aspen machine model for Keeneland supercomputer.

requirements of an application. The Aspen machine model, on the other hand, is used to declare the relation between cores, memory, and interconnect and specify their throughput and bandwidth values as well as supported features such as SIMD capabilities and floating point precision.

Listings 1 and 2 give an example application model for 1D-FFT and a machine model description for a supercomputing cluster, respectively. In the application model the `fft` kernel is represented in terms of floating point operations (`flops`) and memory `loads`. The total counts of these two operations are determined by the statements that use the application `params` declared in the top portion of the model. The `flops` expression declares three traits, `dp`, `simd`, and `fmad`, which can hint the machine model to use accelerated performance metrics. The model hierarchically lists all the main components of the cluster top to bottom, and the parent components indicate the count of each sub-component via either explicit numbers or hardware `params`. The description for the `core` declares what type of computational resources (e.g., `cycles` or `flops`) the processing unit provides, along with all supported traits.

Aspen also provides analysis tools to produce performance estimates and algorithm characteristics for a given application and machine model pair. Several frameworks based on Aspen extend its functionality to automatically create application models [13], generate synthetic workloads [14], characterize complex memory hierarchies [15] , perform hardware-software optimization [16], and model extreme-scale workflows [17].

C. Motivation

This study focuses on increasing the expressiveness and flexibility of machine representations in Aspen and other analytical performance modeling frameworks. The hardware description given in Listing 2 has the following limitations.

- *A formally valid topology allows only specific patterns of hardware organization.* This type of restriction is common across both simulators and structured analytical models and limits the scope of hardware (e.g., experimental, nonconventional) that can be described.
- *The level of detail achievable with the model is fixed.* The representation does not give the developers the ability to dive deep into the specifics of a component for better prediction accuracy or abstract some parts of the hardware to reuse them in higher levels of the machine hierarchy.
- *Complex interconnect infrastructure across components cannot be properly modeled.* Modern systems embed various type of interconnects (e.g., NVLink, UPI, PCIe, OPA, DMI) with non-uniform bandwidths, and current models do not provide the flexibility to express such advanced topologies.

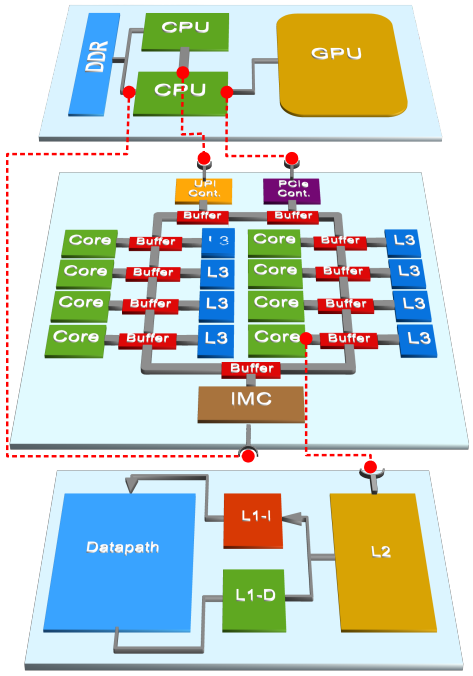

Fig. 1: Multi-layer representation of a CPU socket in a CPU-GPU system.

In the rest of this paper, we present *FLAME* to address the limitations imposed by the hardware models used in existing performance prediction frameworks.

III. Graph-based machine representation

Graph-based machine representation (*FLAME*) uses a connected multi-layer graph to model the hardware. Wires (i.e., interconnects) are represented by edges, and any component connected to an interconnect is represented by a node. Layers of the graph are used for separating different levels of abstractions.

A. Hierarchical Component Topology

Figure 1 illustrates a CPU+GPU system described by our proposed multi-layer graph-based machine representation. The level of detail increases by each layer, and the connections (i.e., edges) across layers maintain the relation between abstracted and detailed versions of models. For example, in the top-most layer, the lower CPU socket has three different

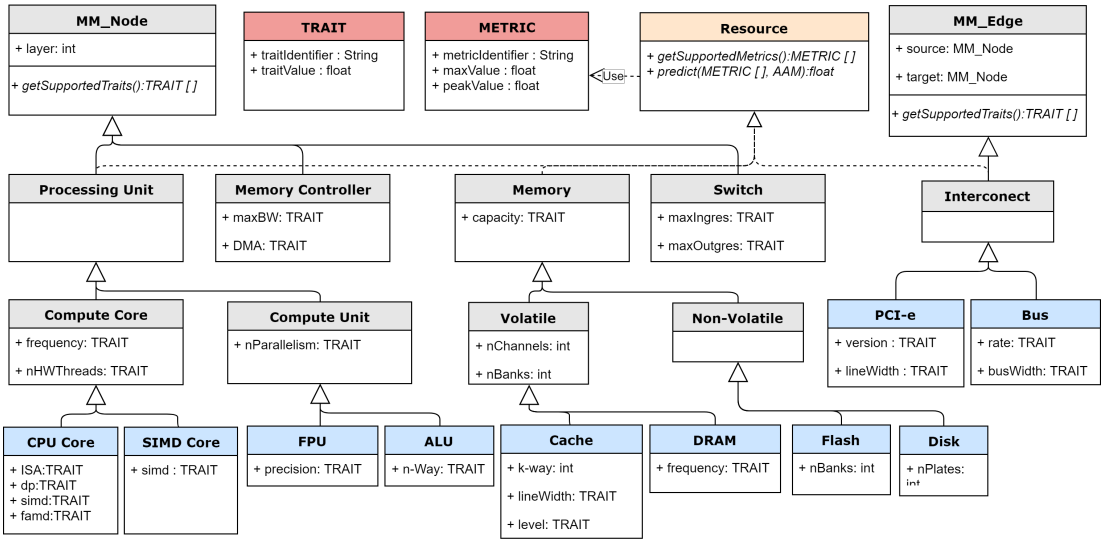

Fig. 2: Partial UML class diagram for graph-based representation.

connections: SMI (Scalable Memory Interconnect) channel for DDR accesses, UPI (Ultrapath Interconnect) to communicate with the other CPU socket, and PCIe bus to reach to the GPU. However, because the CPU socket itself is abstracted in the top layer, the exact point of connection with the chip is not clear.

Inter-layer edges provide an alternate and detailed path for modeling tools to follow, and they attach to the outlets specified by the components (i.e., nodes) in the more detailed layer. For example, in the middle layer, the integrated memory controller (IMC), the UPI controller, and the PCIe controller declare interfaces to route the connections attached from the outside of the chip to the inside. The bottom-most layer further details the sub-components of a core by rerouting the connection coming from the ring buffer to the L2 cache inside the core.

FLAME allows multiple layers of abstractions for any unique type of component in the system, and a fully detailed model will look like a tree of layers. The modeling complexity of a component is determined by the depth of the specific sub-tree associated with that component.

B. Implementation

FLAME follows an object-oriented approach to express a topological and semantical relationship between different hardware components. Figure 2 depicts a partial UML diagram that shows the node and edge hierarchy where the components are derived from. *FLAME* comes with a predefined set of abstract and leaf-level (i.e., non-abstract) class implementations for common hardware. The model developers can extend the abstract classes to build their own components and reuse the leaf-level implementations as the sub-components of their custom hardware.

Each non-abstract component (i.e., a node or an edge) needs to declare a set of `TRAIT`s that will later match with the operations specified in the application model. `TRAIT`s identify the capabilities or features of a component and can also be used to configure the common components that are already implemented by *FLAME*.

A component should declare itself as a `RESOURCE` if it is going to provide a quantifiable functionality (i.e., `METRIC`) to the model. For example, if a CPU core supports floating point operations, the corresponding class should implement `getSupportedMetrics()` function to include *FLOPS* in the list and also return a proper value when the `predict()` function is called with *FLOPS* as the queried metric.

C. Performance Prediction

The `predict()` function is key to implementing flexible and precise performance models with *FLAME*, and any component extending the `RESOURCE` class needs to implement it. The `predict()` function, as shown in Listing 3, takes three parameters, a `METRIC`, a pointer to the Aspen Application Model (`AAM`), and the `METHOD` and returns the value calculated by the function.

FLAME provisions a hierarchical implementation of *predict()* function: A component returns a predicted performance value by using the measurements returned by predict functions of its subcomponents. This structure allows a given architectural organization at a specific level of abstraction to reuse existing prediction implementations for more detailed layers while considering the effects of the cross-component interaction at the higher level on the overall performance.

The valid set of values for the `METRIC` parameter is declared by the `getSupportedMetrics()` function, as explained above, and *FLAME* allows components to implement `predict()` for a wide range of metrics, such as performance, power, cost, and latency. As the depth of a component's layer increases, the number of supported metrics will decrease, since the hardware components will likely have a dedicated function

```
float predict(METRIC, AMM, METHOD);
```

Listing 3: The `predict()` function.

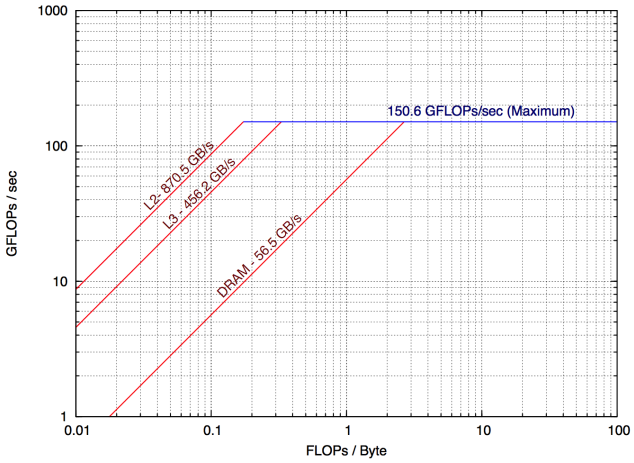

Fig. 3: Roofline model created for the experimental system.

to perform. For example, a CPU core will support both FLOPS and IOPS, whereas an ALU will only support the latter.

The Aspen application model (AMM), the second parameter, is a pointer to the root node of the abstract syntax tree (AST) of the modeled application. AMM can correspond to a single expression, a basic block, a kernel, or an entire application. The supported granularity depends on the level of the hardware component the `predict()` function is being implemented on. For example, a CPU core can take an entire application model, whereas an ALU will only accept expressions with integer arithmetics in it. The `predict()` function in higher level components is also responsible for breaking down the parts of the supplied application model into sub-components.

The `METHOD`s that the `predict()` function can use vary by the characteristics of the underlying architecture, the targeted accuracy of the predicted performance, and also the level of effort that developer wants to spend in building the model. The initial implementation of *FLAME* uses the popular roofline model as the primary method of performance prediction. The roofline model helps understanding computational and memory bounds of a given system by characterizing the relationship between arithmetic intensity and throughput.

IV. Evaluation

To demonstrate the benefits of multi-layer graph-based machine representation, we have performed experiments with a matrix multiplication (matmul) kernel on a multi-core CPU and compared them with the performance predictions for the three different levels of hardware previously explained in Figure 1. We used Aspen to describe an application model for matmul, and we created machine representations for *DRAM-only* (*top* layer), DRAM+L2 (*middle* layer), and DRAM+L2+L3 (*bottom* layer) using the graph-based methodology formerly presented in this study. Our goal is to show the effects of having different levels of hardware details in the model on the prediction accuracy.

We ran the baseline measurement on an Intel Xeon E5-2683 16-core CPU with 40 MB of shared L3 and 512KB per-core L2 cache and 256GB of DDR4-2400 ram. We used a naive

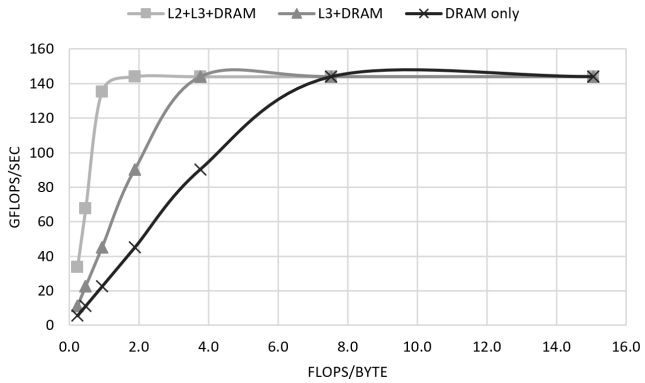

Fig. 4: Attainable throughput values predicted by *FLAME* for three different detail levels of component representations.

OpenMP implementation of dense matrix multiplication kernel for baseline execution. To model the three different levels of detail, we followed the steps below.

1) We run the Empirical Roofline Modeling (ERM) tool [18] to measure practical boundaries for compute and memory bandwidths on our experimental system. The tool uses a synthetic benchmark to identify boundaries for two higher levels of cache and the DRAM. The specific throughput limits for our experimental system are depicted in Figure 3.
2) We then profile the matrix multiplication kernel to extract L2 and L3 hit rates. We use these rates to obtain a combined bandwidth value via a weighted average of separately measured throughputs (which are obtained in the previous step). The top layer directly uses the DRAM bandwidth by assuming all memory accesses are misses, and the middle and bottom layers use DDR+L2 and DDR+L2+L3 bandwidths, respectively.
3) We integrate the throughput values into the `predict()` functions of hardware components, *machine*, *socket*, and *core*, to reflect the memory bandwidth properly at each level of detail.

Figure 4 shows the attainable throughput predictions for the matmul kernel using the three different levels of hardware abstractions. The cache-specific throughput behavior observed by the ERM tool for synthetic kernels is also reflected by the

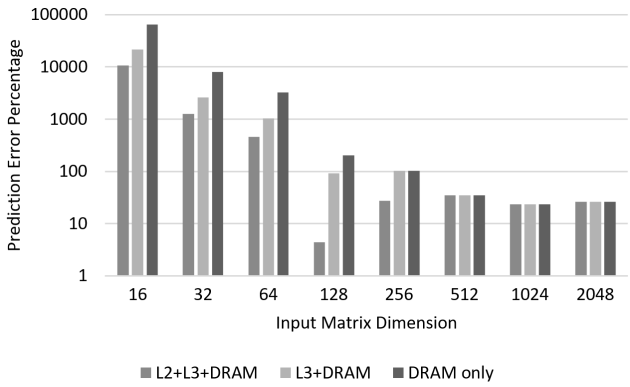

Fig. 5: Prediction error percentages against actual execution.

estimations for the matmul kernel using models created with *FLAME*. As the arithmetic intensity increases with larger input sizes, the throughput is bottlenecked by the computational limits.

The results given in Figure 5 show the prediction error rates against the actual execution of the OpenMP-based matmul kernel. Although the prediction accuracy for smaller input sizes is low, the difference between error percentages for a given input matrix dimension is consistent with the level of detail provided by each model.

Our experiments demonstrate the feasibility of using different abstraction levels by trading modeling complexity for increased accuracy. It is essential to note that *FLAME* targets for precision, not accuracy, and therefore able to represent the relative difference between multiple detail levels correctly while resulting in less accurate predictions, especially for small input.

V. Additional Related Work

Many scientific high performance computing (HPC) applications [8], [9], [10] develop analytical models to estimate scalability of the workload across large supercomputers. A few functional [19] and communication-oriented HPC simulators [20], [21] have been developed to provide more organized and reusable solutions.

Several studies [4], [22], [13], [23], [11], [24] propose tools and techniques based on DSLs for performance modeling. They provide compiler support for annotated model description, generating representations directly from source code, and auto-instrumentation for runtime performance model creation. In general, these tools deal with application modeling only and do not provide support for machine representation.

VI. Conclusion

In this study we have introduced *FLAME*, a flexible machine model representation for structured analytical performance modeling. Different from prior studies, *FLAME* provides a multi-layer graph-based representation to express a wide range of computing hardware using an extensible component class hierarchy.

FLAME uniquely allows a hardware sub-component at any level in the component hierarchy to be abstracted so that the underlying prediction methodologies can be utilized by higher level components. We demonstrate the abstraction capabilities of *FLAME* on a multi-core system for a matrix multiplication kernel on three machine models with different detail levels.

We believe *FLAME* is a powerful technique and will play a major role in structured analytical modeling as it becomes more commonly implemented for different application and machine models.

Acknowledgments

This work was supported by DARPA MTO DSSOC project 1868-Z203-18, and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

[1] D. e. a. Culler, "Logp: Towards a realistic model of parallel computation," ser. PPOPP '93.

[2] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[3] 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, Apr.

[4] N. R. Tallent and A. Hoisie, "Palm: easing the burden of analytical performance modeling," ser. ICS '14.

[5] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," ser. SC '12.

[6] S. e. Chunduri, "Analytical performance modeling and validation of intel's xeon phi architecture," in *Proceedings of the Computing Frontiers Conference*, 2017.

[7] M. Kivelä, A. Arenas, M. Barthelemy, J. P. Gleeson, Y. Moreno, and M. A. Porter, "Multilayer networks," *Journal of complex networks*, vol. 2, no. 3, pp. 203–271, 2014.

[8] M. Larsen, C. Harrison, J. Kress, D. Pugmire, J. S. Meredith, and H. Childs, "Performance modeling of in situ rendering," ser. SC' 16.

[9] G. Herschlag, S. Lee, J. S. Vetter, and A. Randles, "Gpu data access on complex geometries for d3q19 lattice boltzmann method," ser. IPDPS '18.

[10] M. e. Halappanavar, "Towards efficient scheduling of data intensive high energy physics workflows," in *Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science*, 2015, p. 3.

[11] J. Meng, X. Wu, V. Morozov, V. Vishwanath, K. Kumaran, and V. Taylor, "Skope: A framework for modeling and exploring workload behavior," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, p. 6.

[12] A. Chureau, A. Bouchhima, A. A. Jerraya, P. Gerin, and H. Shen, "Flexible and executable hardware/software interface modeling for multiprocessor soc design using systemc," ser. ASP-DAC'07.

[13] S. Lee, J. S. Meredith, and J. S. Vetter, "Compass: A framework for automated performance modeling and prediction," ser. ICS '15.

[14] C. D. Carothers, J. S. Meredith, M. P. Blanco, J. S. Vetter, M. Mubarak, J. LaPre, and S. Moore, "Durango: Scalable synthetic workload generation for extreme-scale application performance modeling and simulation," ser. SIGSIM-PADS '17.

[15] I. B. Peng, J. S. Vetter, S. V. Moore, and S. Lee, "Tuyere: Enabling scalable memory workloads for system exploration," ser. HPDC '18.

[16] M. Umar, S. V. Moore, J. S. Vetter, and K. W. Cameron, "Prometheus: Coherent exploration of hardware and software optimizations using aspen."

[17] E. D. et.al, "Panorama: An approach to performance modeling and diagnosis of extreme-scale workflows," *The International Journal of High Performance Computing Applications*, 2017.

[18] Y. J. e. Lo, "Roofline model toolkit: A practical tool for architectural and program analysis," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, 2015, pp. 129–148.

[19] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale, "Evaluating hpc networks via simulation of parallel workloads," ser. SC '16.

[20] A. Ramaswamy, N. Kumar, A. Neelakantan, H. Lam, and G. Stitt, "Scalable behavioral emulation of extreme-scale systems using structural simulation toolkit," ser. ICPP '18.

[21] B. Van Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal, "Performance models for cpu-gpu data transfers," ser. CCGrid '14.

[22] C. Chan, D. Unat, M. Lijewski, W. Zhang, and J. Bell, "Software design space exploration for exascale combustion co-design," ser. ISC '13.

[23] A. Bhattacharyya and T. Hoefler, "Pemogen: Automatic adaptive performance modeling during program runtime," ser. PACT'14.

[24] J. Chen and R. M. Clapp, "Astro: Auto-generation of synthetic traces using scaling pattern recognition for mpi workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2159–2171, 2017.