Analytically Modeling Application Execution for Software-Hardware Co-Design

Jichi Guo†, Jiayuan Meng‡, Qing Yi†, Vitali Morozov‡, Kalyan Kumaran‡
University of Colorado Colorado Springs†
Colorado Springs, CO, USA
{jguo2, qyi}@uccs.edu
Argonne National Laboratory‡
Lemont, IL, USA
{jmeng, morozov, kumaran}@anl.gov

***Abstract*—Software-hardware co-design has become increasingly important as the scale and complexity of both are reaching an unprecedented level. To predict and understand application behavior on emerging or conceptual systems, existing research has mostly relied on cycle-accurate micro-architecture simulators, which are known to be time-consuming and are oblivious to workloads' control flow structure. As a result, simulations are often limited to small kernels, and the first step in the co-design process is often to extract important kernels, construct mini-applications, and identify potential hardware limitations. This requires a high level understanding about the full applications' *potential* behavior on a future system, e.g. the most time-consuming regions, the performance bottlenecks for these regions, etc. Unfortunately, such application knowledge gained from one system may not hold true on a future system. One solution is to instrument the full application with timers and simulate it with a reasonable input size, which can be a daunting task in itself. We propose an alternative approach to gain first-order insights into hardware-dependent application behavior by trading off the accuracy of analysis for improved efficiency. By modeling the execution flows of user applications and analyzing it using target hardware's performance models, our technique requires no cycle-accurate simulation on a prospective system. In fact, our technique's analysis time does not increase with the input data size.**

I. Introduction

Scientific applications are one of the main driving forces for the design of the next generation large scale systems. Their code can easily reach tens of thousands of lines, with a large number of dynamically interacting functions and modules. On the other hand, computer architectures embrace more parallelism and heterogeneity, facing an exploding design space. With the scale and complexity in both software and hardware reaching an unprecedented level, their co-design has become crucial.

A key challenge in software-hardware co-design is to understand *potential* application behavior on conceptual systems at a high level, gain insights about performance bottlenecks, and discover optimization opportunities for both applications and hardware design. Two key questions about application behavior are: (1) what are the *hot regions* (i.e., the most time-consuming regions of the application), and (2) what are their performance bottlenecks. The hot regions of a workload are often determined by its execution flow, defined as the input-dependent run time traversal of the application code. The concept includes two aspects: hot spots and hot paths. A hot spot is a single code block (e.g., a loop or a function) that consumes a large portion of the application run time; a hot path is the subset of the application execution flow that connects all the hot spots.

Answers to the above questions help system designers extract mini-applications and build benchmarks for future systems. They may also indicate potential hardware limitations and guide system design. Answers to these questions are hardware-dependent; depending on the application's characteristics and resource demands, e.g., memory bandwidth requirement, computation intensity, and degrees of parallelism, hot regions found on one machine often do not remain "hot" on another. As we will show in Section VII, the top 10 hot spots we have identified from profiling a single application on two different machines, an Intel Xeon-based node and an IBM BG/Q node,come in very different orders. In addition, the two lists of hot spots have only 4 in common. As a result, empirical knowledge about application behavior on one system may not be portable to a future system.

Profilers such as gprof [15] are often used to understand application behavior; however, they are only available on existing, accessible systems. Micro-architecture simulators, however, often take an enormous amount of time to simulate even a single kernel with a reasonable sized data set. Moreover, simulators treat applications as black boxes and are oblivious to workloads' control flow structure. Implementing a profiling tool over a simulator, however, is a daunting task and may lengthen the simulation time even further.

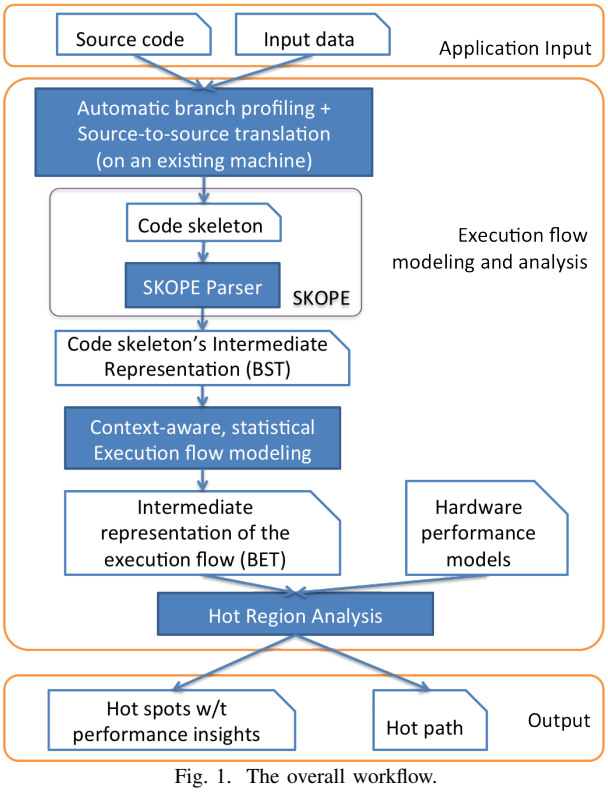

Fig. 1. The overall workflow.

To quickly gain first-order insights about hot regions and reasons behind their performance bottlenecks on future systems,

we present a framework that analytically models the holistic execution flow of an application and then employs parameterized hardware performance models to project application performance for varying architecture design configurations. As shown in Figure 1, the overall workflow of our approach includes three steps. First, a source-to-source application analysis engine analyzes the input code to generate a structural description of the application in the form of the SKOPE workload modeling language [22]. During this step the analysis engine also automatically profiles the application on a local machine to obtain frequencies of the control-flow branches and incorporates the branch outcome distributions into the code skeleton. Second, based on the resulting code skeleton, our performance analysis engine automatically constructs a static model for the application execution flow, which is then characterized using a hardware performance model parameterized with the appropriate architecture configuration. Finally, the performance projection for potential hot regions and their performance bottlenecks are projected and reported.

Our approach essentially uses statistical execution flow models of applications and coarse-grained performance models of hardware to project application performance on a prospective architecture without requiring any simulation or instrumented execution of the application on the target hardware. Local profiling is needed only once to gather the branch frequency statistics, and the profiling results can be reused to analyze and project performance across different architectures. By mathematically modeling the repetitive control flow caused by looping, the analysis and projection time does not scale with the input data size or the actual execution time of the application. By modeling the application execution flow, we are able to gain "contextual" insights such as the control flow leading to individual hot spots, the relations among hot spots, and sizes of input data associated with each invocation of a hot spot. Moreover, we can project whether computation or memory access is the bottleneck.

Our key technical contributions include the following.

1) We designed a new intermediate representation (the BET representation) to incorporate dynamic knowledge about an application (e.g. control flow statistics) and analytically model its execution flow.
2) We developed methods to statistically estimate the hot spots of applications and their performance bottlenecks according to their modeled execution flow on the target hardware.
3) We formulated how to extract the hot paths of an application based on the hierarchical control flow of the application and the already identified hot spots.

The remainder of the paper is organized as follows. Section II discusses related work. Section III presents an overview of the framework and a summary of existing work used in this paper. Section IV presents our analytical model of the statistical behaviors of the execution flow. Section V introduces our algorithms for identifying hot spots and hot paths. Section VI summarizes our experimental methodology. Section VII evaluates the accuracy of the hot spots found by our technique. Conclusions are drawn in Section VIII.

II. Related Work

a) Architecture Simulation: Microarchitecture simulators have been widely used to study prospective hardware [3], [5], [17]. However, developing detailed simulators for new emerging architectures is an extremely time-consuming task. By processing an application in its binary form, it often takes days or even weeks to simulate a large application, and it is difficult to trace the behavior to the source level to gain additional insights. Our research seeks to overcome these difficulties by analyzing applications at the source level and by modeling the architecture configurations at a much higher level.

b) Application Profiling and Tuning: A variety of software tools, e.g., TAU [28], HPCToolkit [2], among others [8], [14], [15], [27], monitor application performance through instrumentation, sampling, and analytical interpretation of runtime statistics to help developers identify potential performance bottlenecks. These tools rely on actual execution of the workload. As a result, they require a fully functional software stack on the target hardware, which may not be available when studying newly adopted or future hardware. Most existing profiling tools focus on individual functions and statements and lack a high level picture of how they are connected at runtime. Our work additionally provides an informative overview of the runtime execution paths, leading to potentially more informed decisions.

It is well recognized application performance cannot be fully modeled with precision analytically, due to the many unknowns in application behavior and architectural intricacies. Consequently, automated empirical tuning has been adopted by researchers to guide the optimization of both domain-specific libraries [11], [34] and general-purpose compilers [12], [16], [37]. Our work targets identifying hot spots in large-scale applications instead of evaluating the impact of different optimizations for isolated computational kernels. Specifically, we trade off the accuracy of performance modeling for higher-level insights into potential application-architecture interactions. For example, by steering away from the cache effects and assuming a constant cache hit/miss ratio, we focus on high-level software-hardware co-design issues such as the resource requirement balances among different architectural components. We demonstrate this approach is sufficient in identifying hot spots and performance bottlenecks in large applications. The identified hot spots can then be used by developers, architecture designers, or existing auto-tuning systems to fine-tune their optimizations.

c) Performance Models: Application- or hardware-specific models have been used in many scenarios to study workload performance and to guide application optimizations [7], [13], where the applications are usually run at a small scale to obtain knowledge about their runtime overhead and performance scaling. Snavely et al. developed a general modeling framework [29] that combines hardware signatures and application characteristics to determine the latency and overlapping of computation and data movement. An alternative approach uses black-box regression, where the workload is executed or simulated over systems with different settings, to establish connections between system parameters and runtime performance [4], [19], [20], [31]. Our work is complementary to these approaches in that we model the structure of the execution flow and identify hot regions.

d) Other Performance Analysis Techniques: Existing research on constructing scientific mini-applications or kernels to represent the behavior of much larger full-scale applications [30] is related to our work in that they utilize domain knowledge to manually identify hot regions along with the associated execution contexts. Static analysis tools, including compiler frameworks such as ROSE [26], are able to automatically recognize the static control flow within programs. Our research essentially augments static control flow of programs with input-dependent runtime statistics to effectively model the runtime behavior of applications. Similar to our work, Narayanan et al. [23] modeled performance characteristics as an expression of input variables but did not analytically model the control flow structure and did not focus on identifying hot regions. Ertvelde and Eckout proposed the idea of using generic algorithms to synthesize shorter and more representative workload [32]. Their machine learning approach has to be trained by running various configurations of the workload on a target hardware.

III. Overview and Background

The overall workflow of our framework is shown in Figure 1. We have extended the SKOPE framework [22] to automatically analyze the source code and abstract its performance behavior according to user-supplied input flags and data sets. The output is a *code skeleton* that summarizes the high level semantics that relate to performance behaviors of the underlying instructions. We then build a model of the application execution flow to synthesize performance characteristics of each code block within the application. Finally, we use the roofline model [35] to estimate the performance of every code region, and the resulting execution flow model with performance estimations is used as input to identify hot spots and hot paths. In the following, we briefly introduce SKOPE's code skeleton language and the roofline performance model.

A. Code Skeletons

The code skeleton explicitly expresses all the control flows of the original code but replaces the actual instruction sequence with performance characteristics including iteration count, degree of parallelism, computation intensity, and data access patterns. A pedagogical example of the code skeleton language is shown in Figure 2(a), which is structured similarly to its original source code in terms of functions, loops, branches, and data types. Two functions, *main* and *foo*, are declared in this example.

A code skeleton is parsed by SKOPE into a data structure named the *block skeleton tree* (BST). Figure 2(b) shows the BST corresponding to the code skeleton in Figure 2(a). Each node of the BST corresponds to a statement in the code skeleton. Statements such as function definitions, loops, or branches may encapsulate other statements, which in turn become the children nodes. The BST does not contain information about the input. Consequently, it alone does not reflect the control flow and data flow, which are often input dependent.

B. Generating Code Skeletons

Within the original SKOPE framework [22], code skeletons were manually generated from an existing application. Using the ROSE compiler framework [26], we have developed an application analysis engine to automatically convert Fortran or C code into code skeletons. It is composed of two components: a source-to-source translator and a branch profiler. The source-to-source translator statically characterizes the instruction mix, array accesses, and control flow structures. The branch profiler profiles a program on a local machine using gcov [1] to obtain branch outcome statistics such as the iteration counts of loops with uncertain boundaries (e.g., `while` loops) and fall-through probabilities for data-dependent branches. Such information is hardware independent and is encoded as expressions of the input data, specifically the input data sizes and distribution of values, which are summarized in a *hint file* provided by the developers. Currently, the code skeleton analysis engine only supports programs with regular data structures such as arrays, which are the most widely used in scientific application. Accommodating more complex pointer-based structures remains future work. In the rest of the paper, we focus on execution flow modeling and performance analysis using the generated code skeleton.

Our current skeleton generation process is domain specific, with a focus on scientific workload and array operations. As a first-order approximation, our static analysis does not take into account issues associated with the limited number of registers, instruction level dependences, and potential compiler optimizations to manage these issues.

C. The Roofline Performance Model

The roofline model [35] estimates the application's performance potential based on its operational intensity, e.g., the ratio between the number of floating point operations and the number of bytes moved to and from the memory. Given the maximum peak flop rate and the maximum memory bandwidth of the target platform, the operational intensity can be used to derive whether the workload is computation-bound or memory bandwidth-bound and then estimate performance projection accordingly. The roofline model is typically used for estimating macro-level, first-order performance. Due to its simplicity, it is being increasingly adopted in application performance modeling.

IV. Execution Flow Modeling

As shown in Figure 1, after obtaining the code skeleton of an input application, our framework invokes an *execution flow modeling* component to reason about its expected run time control flow behavior, e.g., probabilistic of branching outcomes, iteration counts of loops, and dependence constraints among the operations. The key challenge is to meet the following requirements simultaneously.

1) Critical information needs to be preserved so that the user can gain an understanding of the high level control flow semantics.
2) Statistics of computation and memory operations need to be recorded to offer performance insights.
3) Constructing and analyzing the model need to take a minimal amount of time that is asymptotically independent of the input data size of the application.

Conventional representation of the program execution flow, e.g., using a program trace obtained from simulation [36], cannot meet the above requirements due to the difficulty of extracting structural or semantic information from the trace and due to its undesirable property of scaling at least linearly with the size of data being operated. Similarly, call graphs [21] are not a good fit as they only capture function-level interactions without performance details within the individual functions.

We present a new data structure, *Bayesian Execution Tree* (BET), to statically model the execution flow of programs based on their input data combined with various run time statistics. Our key insights are:

1) The execution flow is often repetitive in nature due to loops. By keeping track of the loop iteration counts without repetitively evaluating the loop, the modeling and analysis overhead can be made independent of the data size.
2) While branches may affect subsequent control flow, their effects can be statistically captured based on branch outcome specifications.
3) Performance characteristics are often dependent on only a small collection of variables (e.g., branch conditions, loop boundaries), which can often be analyzed statically given the input data. Even for branches that depend on dynamically generated values, their fall-through probability can be obtained from the knowledge provided by the user or a local profiler when constructing the code skeleton.

The following first presents our *Bayesian Execution Tree* (BET) representation of the program execution flow and then describes our algorithm for automatically constructing the BST from an input code skeleton.

A. Bayesian Execution Tree

Figure 2(c) illustrates an example BET corresponding to the workload in Figure 2(a). A *BET node* refers to the dynamic execution of a code block with a given context. Each BET node is associated with a conditional probability describing the chance

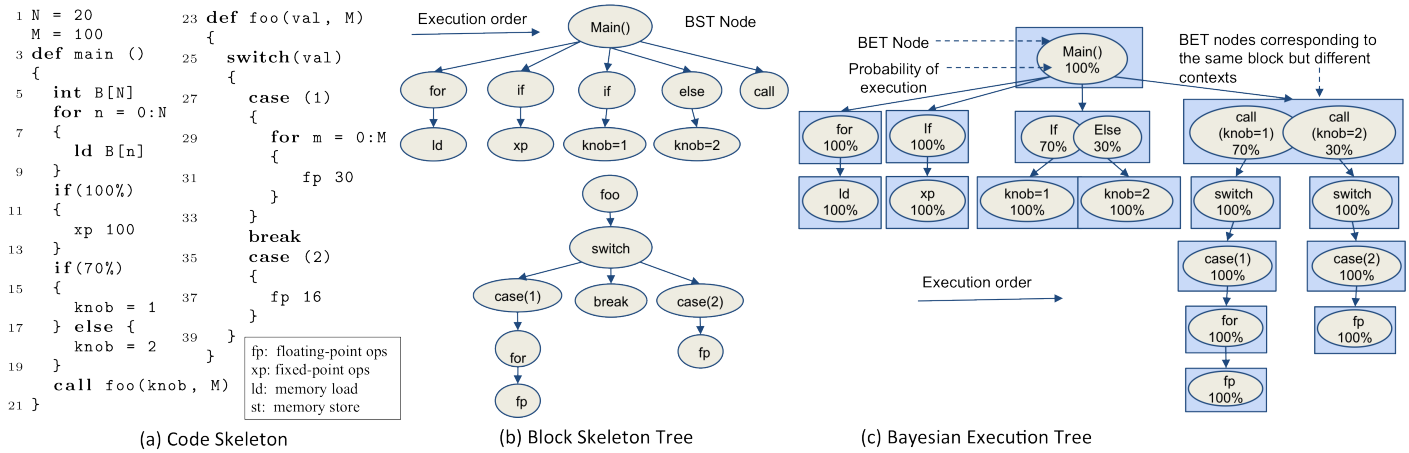

Fig. 2. An example code skeleton, and its corresponding block skeleton tree and bayesian execution tree.

of reaching it given the execution of its parent node. For example, Line 14 in Figure 2(a) is a branch whose outcome would affect the branch at Line 25. This branch leads to two contexts with different values of `knob`, each is associated with a different probability and each creates a BET node corresponding to the function call for `foo` (rightmost nodes in Figure 2(c)).

In contrast to the BST (Block Skeleton Tree) in Figure 2(b), the BET contains the knowledge of the run time values of a number of input variables and the sizes of varying data being operated. It conceptually traverses the BST to "mimic" the run time execution starting from the BST node that corresponds to the main function. The BSTs of different functions are then connected based on the input-dependent execution, where the BST of the callee is mounted as a child within the caller's BST upon each functional invocation.

The BET essentially models the dynamic execution flow by recording the sequence of branches, loops, and functional calls, as well as their hierarchical structure. Since the control flow often depends on the branch outcome, we compute which branch paths may take place and what are their probabilities according to the input data. To do so, we track the probability distribution of *context values*, defined as the set of variables that would affect branch outcomes, loop boundaries, and data accesses. As a result, the BET models each invocation of a code block along with the invocation's probability and context.

B. Constructing the Bayesian Execution Tree

To construct the BET, we first build the initial context with the values of input variables of array dimensions. This context accounts for 100% probability. We then start from the main function and traverse its BST. A BET node is created upon every encountered BST node for each current context. The algorithm traverses the BST in pre-order and recursively creates BET nodes.

At each function call, the entire BST of the callee is copied and mounted in place, with the value of arguments set according to the current context. BST copies of the same function can be mounted to different places wherever it is being called; each invocation would have a different context. After being mounted, the traversal continues into the mounted BST.

Upon a loop statement, a single BET node is created. It merely populates the values of the loop boundaries using variables in the current context without iterating over the loop body. Iterations with different control flows are represented as different child nodes with their corresponding probabilities.

Upon conditional branches and `switch` statements, the conditional probability of a BET is set to the probability distribution of its context multiplied by the probability of the associated branch outcome. Branching statements may also spawn more contexts if variables are assigned different values according to the branch outcome.

`Return` statements are modeled by setting the probability of the associated context to zero, so that the following statements are no longer executed with this context. The zero-probability context is also promoted to ancestor BET nodes, until it reaches a BET node that corresponds to a function call.

`Continue` statements are modeled in a similar way as a branch, except that their probabilities are promoted to ancestors until a BET node corresponding to a loop statement is encountered. `Break` statements is similar to `continue` statements and it further modifies the number of loop iterations. To model its effect, the conditional probability of executing the `break` statement is promoted to the closest ancestor BET node that corresponds to a loop. The expected number of loop iterations is then calculated by $\frac{1-(1-p)^n}{p}$, where p is the conditional probability of break and n is the size of the loop's range. When $p = 0$ (the loop never breaks), the expected iteration count is n.

The resulting BET corresponds to the execution flow with all functions inlined and all performance-sensitive variables evaluated according to the input. Note that this traversal is extremely light weight compared to even functional emulation of the original program: loops are not iterated since it is treated merely as a single node in the BST; no computation is performed to produce data values except for those variables preserved in the code skeleton, which usually have to do with data-dependent control flow.

Due to the branching of dynamic contexts, the number of nodes in the BET can be significantly larger than that of BST. In the worst case scenario, every branch creates two different contexts, each leading to a different subsequent execution flow. In this case, the BET may be 2^B larger than the BST, where B is the number of static branch instructions in the code skeleton. However, this only occurs when the workload is a chain of branches with independent branch conditions; according to our experience, workloads often exhibits nested structures, and the branch outcomes are often correlated. In any case, the size of the BET does not grow with the input size. For all our benchmarks, the size of the BET averages at 88% of that of the source code statements, and it never exceeds a factor of two.

C. Modeling Library Functions

Library functions present a challenge for model-based performance analysis. Although their source code may not be available, they may take a significant amount of time to compute and thus should be considered as hot spots. It is difficult to analytically

model the performance of library functions without knowing their source code, as their control flows can be input-dependent, and their underlying instruction latencies can be hardware-dependent.

We model library functions in a semi-analytical manner. In particular, we assume that for the same input, the numbers of dynamic instructions stay mostly the same across different hardware. We use hardware counters to empirically obtain the mixture of dynamic instructions through profiling on a local architecture and then use the information as input to our analytical roofline model to project their performance on the target architecture. For situations where the number of dynamic instructions vary when operating on different inputs, we randomly generate a sufficient number of input instances, profile dynamic instructions for each instance, and average the statistics to obtain the average mixtures of dynamic instructions.

V. Hot Region Analysis

Our hot region analysis takes the BET as the input and produces two outputs: the hot spots and the hot paths. Hot spots are small code blocks that consume a significant amount of time and thus are potential performance bottlenecks. Hot paths depict *where* the hot spots are invoked during the execution flow and *how* they connect to each other. In particular, the same hot spot may be reachable from several control flow paths, with each invocation operating within a different run time context and taking a different amount of time. A hot path is conceptually a stripped down version of the workload with only hot spots and the control flows that lead to them. Hot paths can also be used for constructing miniapplications.

There are three steps in the hot region analysis: per-block performance estimation, hot spot identification, and hot path construction. The following describes each step in detail.

A. Modeling Code Block Performance

Given a BET as input, we first characterize the performance metrics of each code block in a bottom-up fashion. Each BET node that corresponds to a loop, a branch, or a function defines a code block, and its immediate children nodes form statements within the block. For each code block, we scan its statements and aggregate metrics including floating point operation count, fixed point operation count, number of loads and stores, and size of data types. Note that the number of loads and stores are not the actual number of DRAM accesses; they only roughly tell the number of data elements needed for computation without taking into account caching effects.

Once performance characteristics are collected, they are used as inputs to a roofline model to project the performance of the corresponding code blocks. To reflect the target hardware, the roofline model is parameterized with key hardware parameters such as the peak flop rate, frequency, instruction latency, issue width, vector width, shared cache access latency, memory latency, and peak memory bandwidth. The model computes (1) the time needed to process the given number of operations (T_c) and (2) the time needed to transfer the required amount of data (T_m) and then outputs the maximum of the two assuming there is perfect overlapping between computation and memory accesses. As a first-order estimation, we assume a constant cache miss rate.[1]

To estimate the actual run time instead of the asymptotic performance bound, we have extended the default roofline model by considering the possibility that there may not be enough computation to overlap with memory accesses, especially for blocks with a small number of operations. Assuming T_o is the amount of time that the computation overlaps with memory accesses, the projected performance for a code block is $T = T_c + T_m - T_o$. We compute T_o by $min(T_c, T_m) \times \hat{o}$, where $\hat{o}$ is the degree of overlapping. Based on the heuristic that the chance of computation and memory overlapping increases with the number of floating point operations in the code, we compute $\hat{o}$ as $1 - \frac{1}{Num\_fp\_ops}$, where Num_fp_ops is the number of floating point operations. While this is only a rough approximation of the hardware's capability, we find that such level of accuracy is able to identify the correct hot spots in most cases.

After using the roofline model to estimate the execution time of a single instance of each code block, the total amount of time spent on the BET node is computed as $T \times ENR$, where T is the estimated time for one invocation of the code block, and ENR is its expected number of repetitions. ENR is further computed as $num\_iters \times prob \times ENR_{parent}$, where num_iters is the number of iterations if the corresponding block is a loop, $prob$ is the conditional probability associated with the block, and ENR_{parent} is the ENR of the parent, whose value is 1 for the main function (the root node).

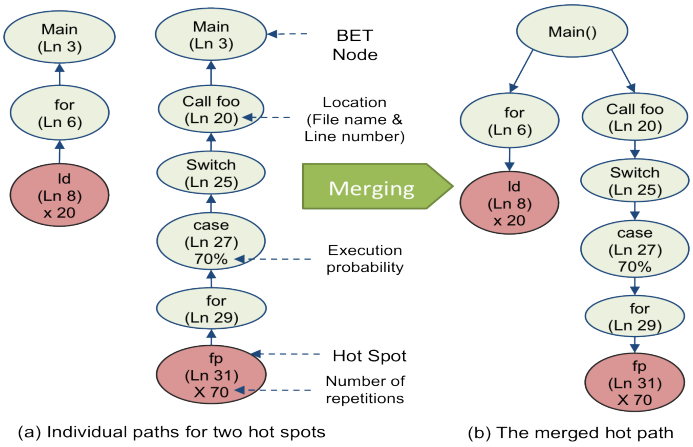

Fig. 3. Building the hot path for Figure 2.

B. Identifying Hot Spots

The definition of hot spots is often subjective and needs to be flexible to satisfy varying requirements. We offer two criteria that a user can use to configure hot spot selection based on his/her individual needs:

1) Time coverage, defined as the minimal percentage of a workload's execution time spent in the hot spots. Together, the hot spots should consume a significant portion of the total run time.
2) Code leanness, defined as the maximal percentage of the total number of instructions that should fall into the identified hot spots. The hot spots should contain a relatively small amount of code compared to the entire application.

The above two criteria cannot be simply replaced with instruction intensity, which is the average amount of time spent per static instruction, because a code block with relatively lower intensity may have more run time coverage if it has much more instructions. The code leanness criteria takes precedence of the time coverage criteria. In particular, if no code block can be selected to satisfy both criteria, we maximize the time coverage under the constraint of the code leanness criteria. Overall, the problem is similar to the knap-sack problem and is NP complete. We solve it using a greedy algorithm.

[1] The cache miss rates for both L1 and LLC are set to 85%; our performance study during daily operation indicates that most workloads' cache miss rate fall between 75% and 95%. This constant is not tuned specifically for benchmarks presented in this paper.

	Sizes of hot spot selections									
	1	2	3	4	5	6	7	8	9	10
SORD Q.(Modl)	0	1	1	2	1	0	1	2	1	0
SORD Q.(Xeon)	1	2	2	2	2	2	3	4	5	6
SORD X.(Modl)	1	2	1	0	1	1	0	1	1	0
SORD X.(BG/Q)	1	2	2	2	2	2	3	4	5	6
CFD Q.(Modl)	0	1	1	2	2	2	3	2	1	0
SRAD Q.(Modl)	0	1	0	1	0	0	0	-	-	-
CHARGEI Q.(Modl)	0	0	0	1	0	0	-	-	-	-
STASSUIJ Q.(Modl)	0	0	0	0	0	-	-	-	-	-

TABLE I

DIFFERENCES BETWEEN THE PROJECTED HOT SPOT SELECTION AND THE MEASURED HOT SPOT SELECTION BASED ON PROFILING. ZERO MEANS THE SET OF N HOT SPOTS EQUALS THE TOP N HOT SPOTS. Q.Modl AND X.Modl REFER TO HOT SPOT SELECTIONS FOR BG/Q AND XEON BASED ON OUR MODELING, RESPECTIVELY. Q.Xeon REFERS TO HOT SPOT SELECTION FOR BG/Q BASED ON XEON PROFILING. X.BG/Q REFERS TO HOT SPOT SELECTION FOR XEON BASED ON BG/Q PROFILING.

Since several BET nodes may refer to different invocations of the same basic block, we first sum up the estimated performance for BET nodes that correspond to the same code block. This projects the total amount of time spent for each block. We then determine the hot spot selection that meet the aforementioned constraints, by first sorting all the blocks based on their projected total execution time and then picking the top-ranking blocks until the constraints are met.

C. Extracting Hot Paths

A hot path summarizes the hierarchical sequence of function calls, loops, and branches, that eventually lead to the hot spots. Since each hot spot corresponds to a BET node, we can obtain the control flow path leading to it by simply back-tracing the BET node's parent until we reach the root node (the main function). This step produces a path for a single hot spot. Figure 3(a) shows the individual paths leading to each hot spot in the example of Figure 2. We then "merge" the hot paths for all hot spots by scanning all the obtained paths, starting from the root node. Shared nodes and edges are included in the hot path, distinct nodes and edges become branches in the hot path. Figure 3(b) shows the resulting hot path that connects the hot spots in Figure 3(a).

Since the BET keeps track of the context values of each code block, the derived hot path, which is a subset of the BET, includes the contexts of each node such as the number of iterations, its branching probability, and the data sizes involved. Such information is helpful for application developers and performance engineers to track down the algorithmic causes of the performance bottlenecks. The hot path also depicts the execution order of the hot spots and thus can help performance engineers analyze the data flow and catch interactions among the hot spots. Such information can also be passed to a compiler to enable path-based optimizations.

VI. EXPERIMENTAL METHODOLOGY

Our current target applications are scientific applications with array accesses. Therefore, the benchmarks we use come from production scientific applications and data-intensive benchmarks, covering different scientific domains including earth science, nuclear physics, computational fluid dynamics, and medical imaging. They include a full application, several mini-applications, and kernel operations, which are written in different languages and exhibit different computation and memory access patterns.

- SORD stands for Support Operator Rupture Dynamics, which is an earthquake simulator that solves a 3D viscoelastic wave propagation problem numerically over a structured grid [9]. It is written in Fortran and parallelized with MPI, containing 5139 lines of code and 370 functions. About 11% of the total instructions are branches, and 8% are loops. SORD is not an memory intensive application and the L1 cache miss rate is less than 8%. Our test case has a 3D grid input where one MPI rank processes $50 \times 400 \times 400$ cells.
- CHARGEI is a function of the Gyrokinetic Toroidal Code (GTC) – a Fortran 3D particle-in-cell application that studies turbulent transport in magnetic fusion [10]. This function calculates the total ion density for a given ion distribution and contains eight loop structures where some loops produce the array structures that are consumed in other loops.
- SRAD is an operation in medical imaging that removes speckles for ultrasonic or radar images without destroying important image features [6]. It first computes a signature from a sample image area that contains speckles; it then diffuses the image with coefficients of each pixel set to the similarity between the local image signature and the speckle's signature. Our test case analyzes an image of 2048×2048 pixels with a sample area of 128×128 pixels.
- CFD is an unstructured-grid, finite-volume solver for the 3-D Euler formulation of Navier-Stokes equations for compressible flow [6]. It is a miniapplication which contains a main time stepping loop that iteratively performs three operations to update pressure, momentum, and density. Our test case has a grid of moderately sized 97,000 cells.
- STASSUIJ function lies in the core of nuclear physics Green's Function Monte Carlo application [18], [24], [25], which performs the calculation of a structure for light nuclei. The function applies a two-body correlation operator including tensor correlations to the many-body wave function. This builds in the correlations induced by the two- and three-nucleon potentials. Algorithmically, the function consists of two distinct phases. First, it multiplies a 132×132 sparse matrix of real numbers with a 132×2048 dense matrix of complex numbers. Second, it exchanges the groups of four elements in each row of the resulting matrix in a butterfly pattern. The indices of the elements to be exchanged are stored in a separate array.

We validated the modeling results on the Blue Gene/Q supercomputer located at Argonne National Laboratory (referred to as *BG/Q*), and a local, Intel Xeon E5-2420 machine (referred to as *Xeon*). On BG/Q, each node has 16 Power A2 cores, 16 GB of DDR3 memory, a messaging unit, a shared 32 MB L2 cache, and a crossbar. A Power A2 core has a clock frequency of 1.6 GHz and accesses private 16 KB L1D and 16KB L1I caches. To obtain the BG/Q hardware model, we identified (with a series of in house micro benchmarks) the latency of accessing L2 cache to be 51 processor clock cycles and the latency of accessing the DRAM to be 180 processor clock cycles. The Intel Xeon machine comes with 12 cores, 1.9 GHz clock frequency, and 64 GB memory. Our benchmarks were compiled using the native IBM XL Compiler Suite with the "-O3" optimization flag on BG/Q and were compiled using GFortran 4.7 with "-O3" on the Intel Xeon.

To evaluate the quality of hot regions selected by our model-based projections, we compare the resulting hot regions identified by our framework with the results generated by the machines' native profiler [33] as well as manually instrumented timing functions. The profiled output contains the most time-consuming statements in the source code and ranks them according to their run time. Based on such information, we identify the most time-consuming code blocks and then further measure the time consumed by these blocks. We use a low-overhead, high-resolution timer based on the machines' special registers. The resulting time profile of the basic blocks are used as the baseline for our comparison.

The quality of the projected set of hot spots is quantitatively measured by the *selection quality*. Since the application developer

	Sizes of hot spot selections									
	1	2	3	4	5	6	7	8	9	10
SORD Q.(Modl) %	100.00	80.49	86.96	87.73	90.28	100.00	99.43	99.45	99.49	100.00
SORD Q.(Xeon) %	57.75	59.21	73.26	87.73	80.26	74.77	73.74	72.94	72.23	71.95
SORD X.(Modl) %	96.93	94.16	97.12	100.00	99.96	99.96	100.00	99.96	99.96	100.00
SORD X.(BG/Q) %	96.93	56.59	68.34	56.51	77.22	96.12	92.08	89.30	86.86	85.18
CFD Q.(Modl) %	100.00	90.43	85.15	80.63	80.56	93.13	90.38	92.13	93.62	100.00
SRAD Q.(Modl) %	100.00	95.05	100.00	99.56	100.00	100.00	100.00	-	-	-
CHARGEI Q.(Modl) %	100.00	100.00	100.00	97.30	100.00	100.00	-	-	-	-
STASSUIJ Q.(Modl) %	100.00	100.00	100.00	100.00	100.00	-	-	-	-	-

TABLE II

THE HOT SPOT SELECTION QUALITY, DEFINED AS THE RUN TIME COVERAGE OF THE FIRST N PROJECTED HOT SPOTS COMPARED TO THAT OF THE FIRST N PROFILED HOT SPOTS. Q.Modl AND X.Modl REFER TO HOT SPOT SELECTIONS FOR BG/Q AND XEON BASED ON OUR MODELING, RESPECTIVELY. Q.Xeon REFERS TO HOT SPOT SELECTION FOR BG/Q BASED ON XEON PROFILING. X.BG/Q REFERS TO HOT SPOT SELECTION FOR XEON BASED ON BG/Q PROFILING.

is mostly interested in the run time coverage of the identified hot spots, the selection quality should reflect how similar the run time coverage of the projected hot spots is to that of the measured hot spots. We formulate the selection quality for N hot spots as $(1 - \frac{|P(N)-M(N)|}{M(N)}) \times 100\%$, where $P(N)$ is the measured run time coverage for the first N *projected* hot spots, and $M(N)$ the measured run time coverage for the first N *measured* hot spots. A 100% selection quality means the projected selection of hot spots have the same run time coverage as the measured selection, but selections do not necessary need to match. Note that different selections may arrive at the same selection quality if there are multiple hot spots with approximately the same run time coverage.

Instead of using performance modeling, one may suggest selecting the hot spots based on empirical knowledge gained from a different machine, with the assumption that the selection of hot spots may remain the same across hardware. To evaluate this alternative, we study the selection quality if we use the hot spots obtained from Xeon for BG/Q. The resulting selection quality is then compared with that from our projection-based selection.

VII. EVALUATION RESULTS

This section uses the results generated by profilers on BG/Q and Xeon to verify our model-based hot-region analysis. For each application and each machine, we collected two sets of data. The first set, `Prof`, refers to the hot spots identified using the profilers, with the sequence of hot spots ranked in descending order according to their *profiled* runtime. The second set, `Modl`, refers to the hot spots obtained from our model-based analysis and ranked in descending order according to their *projected* runtime. To further investigate how portable the hot spot selection is across different hardware, we also attempted to select the hot spots on each machine according to the runtime profile over the other machine.

For each set of hot spots in `Modl`, `Modl(p)` refers to the *projected* runtime coverage of the hot spot selection suggested by our performance model, while `Modl(m)` refers to the *actual* runtime coverage of the same hot spot selection measured by executing the application on the actual hardware. For all experiments, we set the hot spot selection criteria so that code leanness is less than 10% of the instructions and time coverage is more than 90% of the total runtime. A hot spot selection of size N would compare the first N hot spots in `Prof` with the first N hot spots in `Modl`. Among our benchmarks, SORD is a full application and is the most complex. It is therefore discussed in detail.

A. SORD: Case Study for a Full Application

Figure 4 shows the hot spot selection results for SORD. `Prof.Q` and `Prof.X` refer to the runtime coverage of the hot spot selections obtained from profiling on BG/Q and Xeon, respectively. They serve as our baselines for validation. `Modl.Q(m)` and `Modl.Q(p)` refer to the measured and projected runtime coverage on BG/Q using the hot spot selections obtained from our models; `Modl.X(m)` and `Modl.X(p)` are

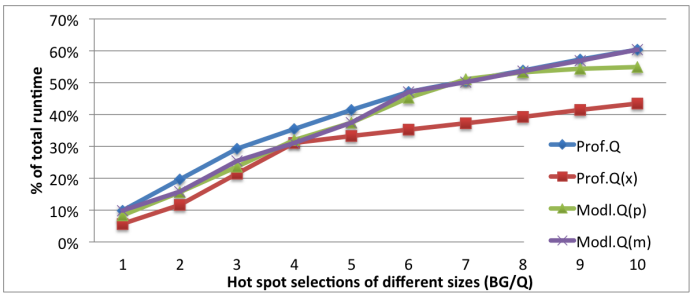

Fig. 4. Profiled and projected hot spot selections for SORD on BG/Q. `Prof.Q` refers to the hot spot selections resulted from profiling on BG/Q and it serves as the baseline. `Modl.Q(p)` and `Modl.Q(m)` show the projected and measured runtime coverage on BG/Q for the model-projected hot spot selections, respectively. `Prof.Q(x)` shows the measured runtime coverage on BG/Q for the hot spot selections obtained from Xeon profiling.

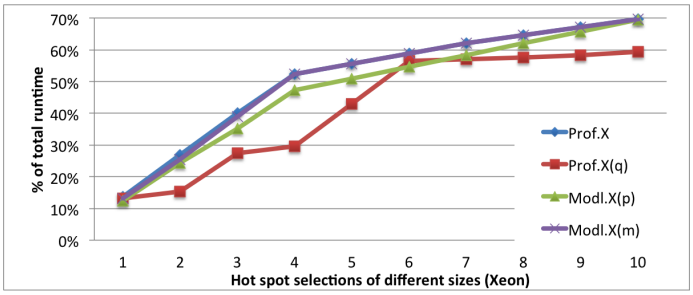

Fig. 5. Profiled and projected hot spot selections for SORD on Xeon. `Prof.X` refers to the hot spot selections resulted from profiling on Xeon and it serves as the baseline. `Modl.X(p)` and `Modl.X(m)` show the projected and measured runtime coverage on Xeon for the model-projected hot spot selections, respectively. `Prof.X(q)` shows the measured runtime coverage on Xeon for the hot spot selections obtained from BG/Q profiling.

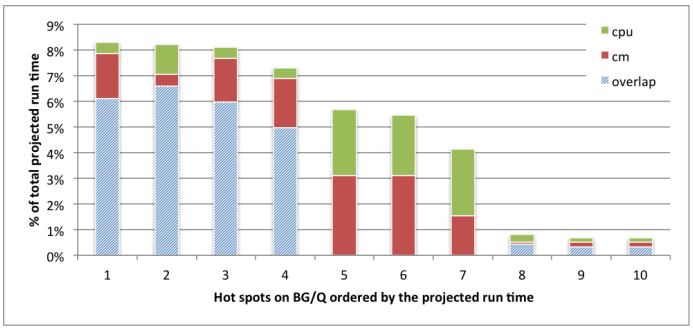

Fig. 6. The projected performance breakdown for SORD's hot spots on BG/Q.

the corresponding hot spot selections on Xeon. The similarity between the `Prof.Q` curve and the `Modl.Q(m)` curve shows that the measured selection and the projected selection of hot spots have nearly the same coverage for all selection sizes. The `Prof.Q` curve has three distinct segments, each with a relatively constant slope – the first three spots, the next three spots, and the rest of them. The slopes of the first two segments

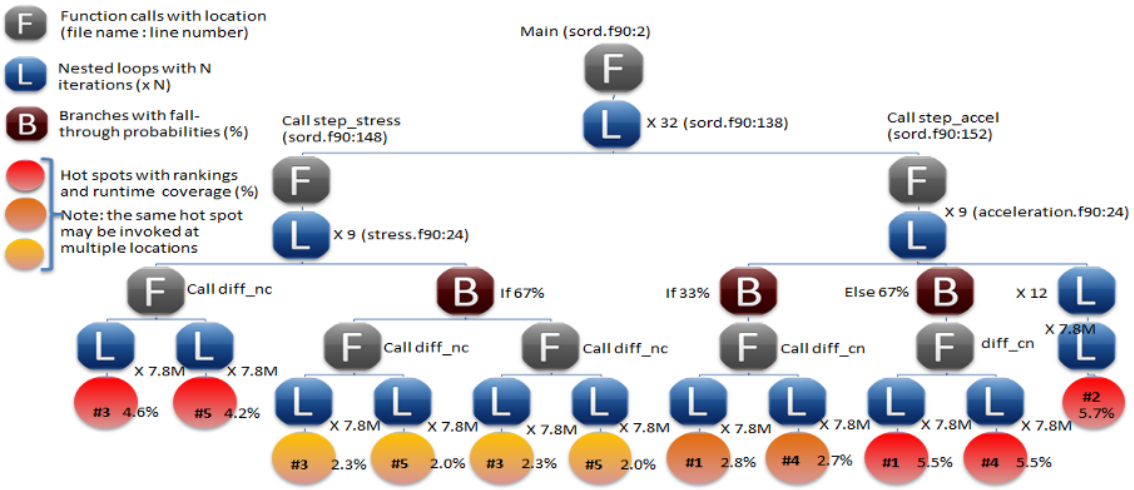

Fig. 9. The SORD hot path for the top 5 hot spots on BG/Q. Note that a hot spot may be invoked from multiple places.

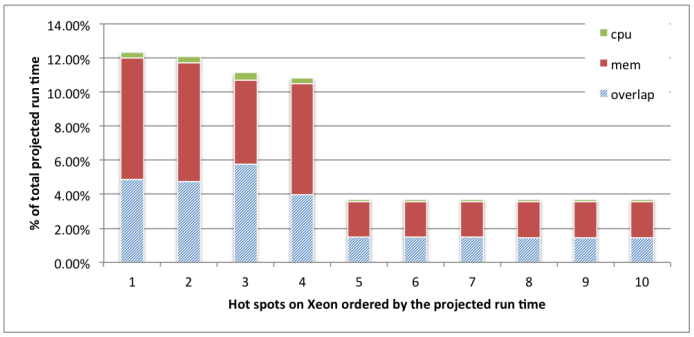

Fig. 7. The projected performance breakdown for SORD's hot spots on Xeon.

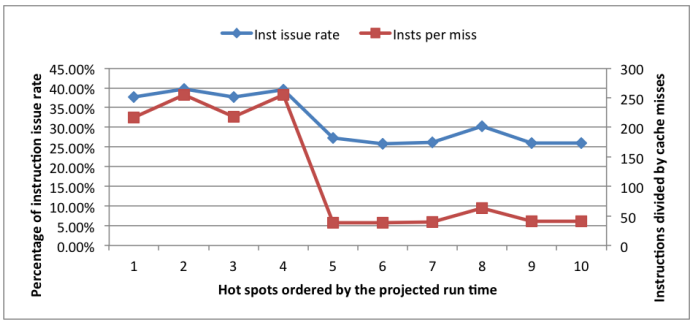

Fig. 8. The profiled performance insights for SORD's hot spots on BG/Q. We show the issue rate and the computation intensity, calculated as the number of completed instructions divided by the number of L1 misses.

are almost identical, which means that the first six hot spots have relatively similar runtime coverage. As Table I shows, our modeling has correctly identified all six of them, but not in the same order. Actually, the very first hot spot was identified correctly even though it only takes 9.9% and the second spot takes 9.7%! The curves marked `Modl.Q(p)` and `Modl.Q(m)` show the sensitivity of our hardware model and the accuracy of projected runtime coverage. They present the same selection of hot spots with projected versus measured runtime coverage and have a reasonably good match. Similar observations can be made from `Prof.X`, `Modl.X(m)`, and `Modl.X(p)`, the profiled and measured hot spot selections on Xeon. The selection quality using our model-projected hot spots is above 80% for BG/Q and above 94% for Xeon.

In fact, the hot spot selections on BG/Q (`Prof.Q`) and Xeon (`Prof.X`) are consisted of different sets of hot spots; the fist six out of ten hot spots are different between the two machines. They come in different ordering as well. Therefore, the hot spot selection on Xeon is a poor representative for the hot spot selection on BG/Q, and vice versa. As can be observed Figure 4, `Prof.Q(x)`, the measured selection quality when the Xeon-suggested hot spots are used to represent BG/Q execution, is quite different from `Prof.Q`. The same is true when comparing `Prof.X(q)` and `Prof.X`. While the hot spot selection is not portable between machines, our model correctly projects the first 10 hot spots for both machines.

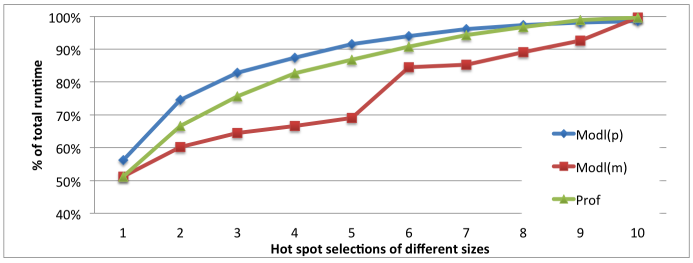

Fig. 10. Profiled and projected hot spot selections for CFD. `Prof`, `Modl` refer to the hot spot selections resulted from BG/Q profiling and performance modeling. `Modl(p)` and `Modl(m)` show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.

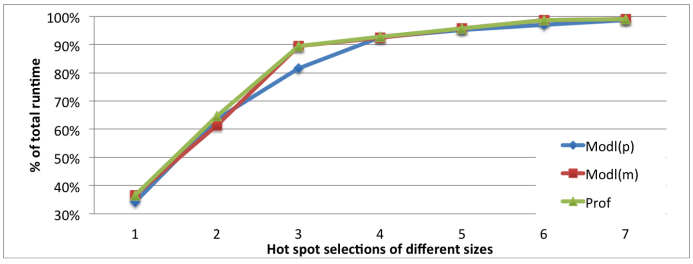

Fig. 11. Profiled and projected hot spot selections for SRAD. `Prof`, `Modl` refer to the hot spot selections resulted from BG/Q profiling and performance modeling. `Modl(p)` and `Modl(m)` show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.

An additional advantage of using model-based performance analysis is that the model can provide insights for each hot spot, which are often not available from results aggregated over the entire execution. Figure 6 shows the model-projected performance breakdown on BG/Q for each hot spot with regard to the time spent in computation, memory accesses, as well as the

time during which computation and memory accesses overlap. While the profiled measurements do not provide such insights directly, Figure 8 shows the profiled issue rate and computation intensity, which indicate that for the latter 6 hot spots, the hardware pipeline is stalled often, and there is a dramatic decrease in the number of instructions per L1 cache miss, which are likely performance bottlenecks. Such observation closely correlates to our projected insights in Figure 6. Further, since our model keeps track of the execution context, we are able to additionally provide the input flags and the data sizes for each invocation of the hot spot (not shown in the figures).

We also used modeling to analyze performance bottlenecks for Xeon. Figure 7 shows that the performance bottlenecks for the hot spots on Xeon are different from those on BG/Q. In general, there is a significant increase in the percentage of time spent in memory accesses. This is because the Intel Xeon machine has smaller L1 cache and larger memory latency but faster processing speed comparing to BG/Q. In addition, Xeon provides wider SIMD than BG/Q and the compiled binary is highly vectorized by default.

Finally, combining the hot spot selection with our model of the execution flow, we can reconstruct the hot path of SORD which tells how the hot spots are invoked and connected during the execution. The hot paths for BG/Q and Xeon are different because of the difference in hot spot selection. Due to space limitation, we only illustrate the hot path for BG/Q in Figure 9. The hot path shows all control flows reaching the hot spots starting from the main function. We can further distinguish different invocations of the same hot spot. Information about how many times a hot spot is repeated, and the probability of each control flow reaching the hot spot, is also revealed. With such information, one can quickly gain a bird-eye view of the application behavior on BG/Q, identify the most important control flow path to optimize, or build a miniapplication accordingly.

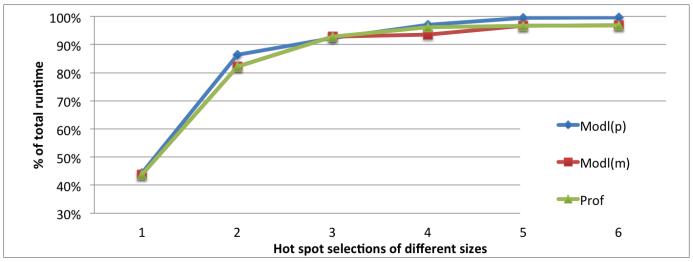

Fig. 12. Profiled and projected hot spot selections for CHARGEI. `Prof`, `Modl` refer to the hot spot selections resulted from BG/Q profiling and performance modeling. `Modl(p)` and `Modl(m)` show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.

B. Results For Mini Applications and Kernels

For the CFD benchmark, we have successfully identified all top 10 hot spots, as shown in Table II. The runtime coverage for measured and projected selections, presented in Figure 10, match reasonably well, and the actual selection quality is better than 80%. Due to space constraints, we only present our projections for the hot spot selection on BG/Q, which can later be used to construct the hot path. When we compare the actual versus projected runtime coverage (the curve Modl(m)), we have identified that the 6th hot spot was significantly underestimated in runtime. Our careful investigation has revealed that this hot spot was expected to take less than 3% of total runtime, but it took 15%. The hot spot was performing the calculation of velocity from a given density and momentum, and involved a series of divisions. Our hardware model does not currently differentiate the varying kinds of floating point instructions and treat all of them equal. On BG/Q, the division is a relatively expensive operation and it is expanded by a compiler into a sequence of predefined instructions based on a reciprocal estimate instruction and a Newton-type iteration refinement, thereby resulting in a much more computationally intensive workload. Once we have picked the "offending" hot spot, the runtime coverage quickly converged.

For the SRAD benchmark, the top three measured hot spots took 37%, 28%, and 25% of the runtime, respectively. Figure 11 shows that our projected hot spot selections have a runtime coverage that is almost identical to that of the measurement-based selections. As seen from Table I, we have switched the order between the second and the third host spots, but because they have a very similar runtime coverage, our hardware model did not differentiate between the two. Note that the first and the third hot spots are standard `exp` and `rand` functions in the math library, and we are applying our empirical modeling technique introduced in Section IV-C and estimate the overhead from these calls which results in a reasonable runtime coverage.

Figure 12 shows the runtime coverage of the hot spot selection of the CHARGEI benchmark. As measured, the CHARGEI benchmark presents two dominating hot spots, one accounting for 44% of the runtime and the other for 38% of the runtime. It is shown in Table I that our model projects the correct ranking of the hot spots as well as their runtime coverage, albeit inverting the order of spot number 4 and spot number 5. The runtime coverage for these spots are around 3% and very close to each other, which is too small to differentiate by our hardware model.

Profiling reveals that the STASUIJ kernel has the top hot spot taking 68% of the runtime and the second taking 23% of the runtime. As Table I shows, the model correctly identifies the hot spot selection and ordering. The `Prof` and `Modl(m)` curves in Figure 13 perfectly overlap. The projected runtime coverage and the execution time for the first hot spot are overestimated by our framework. Further investigation has indicated that this hot spot is the loop which takes an element of a sparse matrix and applies the scaling of the complex vector by this element. The IBM XL Compiler was converting this spot into highly vectorized code, while in the hardware model we do not account for vectorization features.

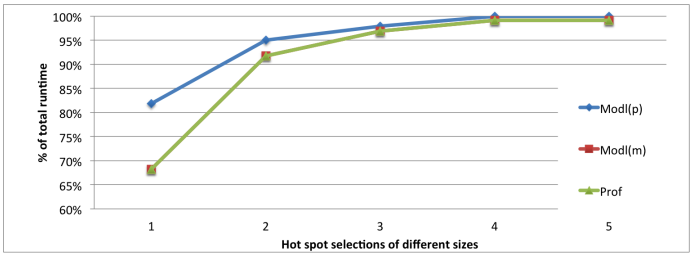

Fig. 13. Profiled and projected hot spot selections for STASSUIJ. `Prof`, `Modl` refer to the hot spot selections resulted from BG/Q profiling and performance modeling. `Modl(p)` and `Modl(m)` show the projected and measured runtime coverage on BG/Q using the model-projected hot spots, respectively.

C. Sources of Inaccuracy

The inaccuracy in selecting the hot spots is caused by the jittering in the projection error across different code blocks. Such jittering can be caused by different degrees of computation and memory access intensities, for which the roofline model may exhibit different sensitivities. We also assume perfect instruction level parallelism, which may not hold true for different code blocks. Projection error can also be caused by performance interactions among the code blocks due to caching effects. For example, the code for the first hot spot in SORD is almost identical to the code of the fourth hot spot, yet the latter can

reuse data brought in from the memory by the former, therefore taking less amount of execution time.

Inaccuracy may also stem from our approximation of the workload characteristics. The instruction count and instruction mix obtained from the static analysis may not reflect the actual number of instructions in the binary. There are also memory accesses that are not captured by the code skeletons, such as loading and storing stack variables. Other effects that we neglect from the modeling, such as CPU pipeline stalls or register spilling may also add to inaccuracies as well.

VIII. Conclusion and Future Work

Understanding potential application behavior for prospective architectures play an important role in hardware-software co-design. We present a technique to analytically model the application execution flow to gain first-order insights into its hardware-dependent performance characteristics without detailed micro-architecture level simulation. The obtained performance insights include hot spots and their run time coverages, limiting performance factors for each hot spot, and the hot path that connects all hot spots in the execution flow. By capturing the statistical behavior of the application control flow and integrating the estimated characteristics with hardware performance models, our technique is able to statically project and analyze the performance within a few minutes, and the projection time remains invariant to the input data size. We have validated our framework over two distinct systems using production codes, including an earthquake simulation application, and showed that the hot spot selection quality averages at 95.8% and is no worse than 80% in all cases. Our execution flow modeling is independent of hardware performance models. In this paper, we use the `roofline` model to project hardware performance. However, more sophisticated models can be used.

Our future work includes extending our framework to project hot regions and performance bottlenecks for multi-node execution of the applications, to extend the execution flow model to properly handle complex pointer-based data structures, and to refine our hardware performance model to improve its accuracy for a wide variety of architectures.

Acknowledgment

The authors thank the ALCF application and operations support staff for their help. They also gratefully acknowledge the help provided by the application teams whose codes are used herein.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This research is partially funded by the Argonne National Laboratory's LDRD project 2013-213-NO. The research is also partially funded by the National Science Foundation under Grants 1261778 and 1261811, and the Department of Energy of USA under Grant DE-SC001770.

References

[1] gcov documentation.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurr. Comput. : Pract. Exper.*, 22:685–701, April 2010.

[3] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[4] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *ICS*, 2008.

[5] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4), 2006.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphisc processors using CUDA. *JPDC*, 2008.

[7] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *PPoPP*, 2010.

[8] Luiz A. DeRose. The hardware performance monitor toolkit. pages 122–132, 2001.

[9] G. P. Ely, S. M. Day, and J.-B. Minster. Dynamic rupture models for the southern San Andreas fault. *Bull. Seism. Soc. Am.*, 100(1):131–150, 2010.

[10] S. Ethier, W. Tang, and Z. Lin. Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms. *SciDAC 2005, Journal of Physics: Conference Series*, 16:1–15, Nov. 2011.

[11] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, Seattle, WA, 1998.

[12] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Olivier Temam. A pratical method for quickly evaluating program optimizations. In *HiPEAC*, November 2005.

[13] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp. Modeling the performance of an algebraic multigrid cycle on HPC platforms. In *ICS*, 2011.

[14] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22:702–719, 2010.

[15] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39, 2004.

[16] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Curtis L. Janssen, Helgi Adalsteinsson, and Joseph P. Kenny. Using simulation to design extremescale applications and architectures: programming model exploration. *SIGMETRICS Perform. Eval. Rev.*, 38, March 2011.

[18] M. H. Kalos, M. A. Lee, P. A. Whitlock, and G. V. Chester. Modern potentials and the properties of condensed ^{4}He. In *Phys. Rev. C 66, 044310-1:14*, 1981.

[19] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS-XII*, 2006.

[20] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *PPoPP*, 2007.

[21] Ken Kennedy Mary W. Hall. Efficient call graph analysis. *Letters on Programming Languages and Systems (LOPLAS)*, 1(3), Sept 1992.

[22] J. Meng, X. Wu, V. A. Morozov, V. Vishwanath, K. Kumaran, V. Taylor, and C-W. Lee. SKOPE: A Framework for Modeling and Exploring Workload Behavior. 2012.

[23] Sri Hari Krishna Narayanan, Boyana Norris, and Paul D. Hovland. Generating performance bounds from source code. In *Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010)*, 9 2010.

[24] S. C. Pieper, K. Varga, and R. B. Wiringa. Quantum Monte Carlo calculations of A=9,10 nuclei. In *Phys. Rev. C 66, 044310-1:14*, 2002.

[25] S. C. Pieper and R. B. Wiringa. Quantum Monte Carlo calculations of light nuclei. In *Annu. Rev. Nucl. Part. Sci. 51, 53*, 2001.

[26] Daniel J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[27] J. Reinders. VTune performance analyzer essentials. April 2005.

[28] Sameer S. Shende and Allen D. Malony. The Tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20:287–331, 2006.

[29] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *SC*, 2002.

[30] Andrew Stone, John Dennis, and Michelle Strout. Establishing a miniapp as a programmability proxy. In *PPoPP*, 2012.

[31] V. Taylor, X. Wu, and R. Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, March 2003.

[32] Luk Van Ertvelde and Lieven Eeckhout. Workload reduction and generation techniques. *IEEE Micro*, 30(6):57–65, November 2010.

[33] Bob Walkup. Unsupported Library to Access Performance Counters of BG/Q.

[34] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[35] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr 2009.

[36] Rajiv Gupta Xiangyu Zhang. Whole execution traces and their applications. *Transactions on Architecture and Code Optimization (TACO)*, 2(3), Sept 2005.

[37] Qing Yi. POET: a scripting language for applying parameterized source-to-source program transformations. *Softw. Pract. Exper.*, 42(6), Jun 2012.