Inferred Models for Dynamic and Sparse Hardware-Software Spaces

Weidan Wu   Benjamin C. Lee
Duke University
{weidan.wu, benjamin.c.lee}@duke.edu

Abstract

Diverse software and heterogeneous hardware pose new challenges in systems and architecture management. Managers benefit from improving introspective capabilities, which provide data across a spectrum of platforms, from software and datacenter profilers to performance counters and canary circuits. Despite this wealth of data, management has become more difficult as sophisticated decisions are demanded.

To address these challenges, we present modeling strategies for integrated hardware-software analysis. These strategies include (i) identifying shared software behavior; (ii) quantifying that behavior in a portable, microarchitecture-independent manner; (iii) inferring generalized trends with statistical regression models; and (iv) automatically constructing/updating these models as new software profiles are obtained.

Models produced by these strategies are accurate for general SPEC2006 applications with median errors of 8-10%. Predicted and actual performance are strongly correlated with coefficients of $\rho > 0.9$. Moreover, when we exploit application semantics and domain-specific software parameters, model accuracy improves and model complexity falls. In a case study for sparse linear algebra, we present models with 5% median error and new capabilities in coordinated hardware-software tuning.

1. Introduction

Hardware management is the art of linking data to decisions. But forging this link is increasingly difficult. Across a spectrum of computing platforms, decisions must be made with increasingly sparse data. Not every node can be profiled yet datacenter managers must navigate diverse hardware-software interactions. Not every configuration can be profiled yet adaptive chips must navigate performance and power trade-offs.

Our ability to collect data has advanced significantly at all scales, from software and datacenter profilers to performance counters and canary circuits. But even as introspection has supplied more data, the decisions demanded of hardware managers have become more complicated. Datacenters must allocate and schedule software while navigating heterogeneity and contention. Architectures must adapt operating parameters and structural resources to dynamic application behavior.

All of these decisions require linking data to expectations of performance and closing the data-to-decision gap. But because many of these decisions involve diverse software on heterogeneous hardware, prior efforts that separate application analysis and architectural optimization are insufficient. Instead, we must reason about software, hardware, and their interactions in a coordinated fashion.

To integrate hardware-software analysis yet keep costs tractable, we present strategies to share profiled behavior (§2). First, we break an application into shards and profile microarchitecture-independent measures of behavior. Given profiles, we construct models that predict performance as a function of software behavior and hardware parameters. Finally, because the number of parameters explodes in an integrated hardware-software space, an automated heuristic constructs the model.

We demonstrate this strategy in two settings. In the general and more difficult setting, we measure detailed software behavior for arbitrary applications and infer a model. Models in the second, domain-specific setting exploits programmer-level knobs in tunable codes. In both settings, we link hardware-software interactions to performance. Thus, we make the following contributions:

- Laying a foundation for system management, we construct predictive models for hardware-software interactions. But software behavior is dynamic, exhibits high variance, and introduces an unwieldy number of parameters. We present a heuristic that automatically builds and updates regression models as software behavior is profiled. (§3)
- Inferred models interpolate and extrapolate performance for diverse hardware-software interactions. Median errors are 8-10%. Predicted and actual performance are strongly correlated with coefficients of $\rho > 0.9$. (§4)
- Given domain knowledge, software behavior is captured more concisely. Domain-specific software parameters produce smaller, more accurate models. In a case study for sparse linear algebra, we show accurate models for highly irregular performance topologies and demonstrate their application to coordinated hardware-software tuning. (§5)

Collectively, these results lay the foundation for understanding diverse software on heterogeneous hardware. By linking sparse data to performance predictions, we enable future work in control mechanisms for reconfigurable architectures and allocation mechanisms for heterogeneous datacenters.

2. Sharing – Principles and Strategies

Capturing hardware-software performance is made difficult by highly variable software behavior. To address this challenge, we infer shared behavior and construct integrated models with four strategies.

2.1. Shard-level Profiles

Models are most effective when inferred from diverse data. To increase diversity, we break an application into shards, each with an equal number of instructions. When collected for short shards, profiles detect fine-grained phase behavior. In contrast, monolithic application profiles (e.g., average instruction mix) obscure intra-application diversity.

This diversity is needed when sharing profiles between applications. Suppose we have profiled several applications and are given a new one. Profiles of monolithic application behavior are useful only if the new application resembles a previously observed one. This constraint is too restrictive to meaningfully share profiled insights.

Relaxing this similarity constraint, fine-grained shards in the new application may resemble disparate shards from others. As illustrated in Figure 1, profiles from relevant shards can be drawn from several applications to capture partial similarities. Sharding increases the

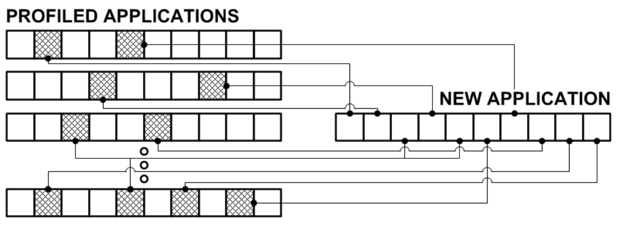

Figure 1: Reflecting partial similarity, a new application might be understood in terms of shards drawn from several previously profiled applications.

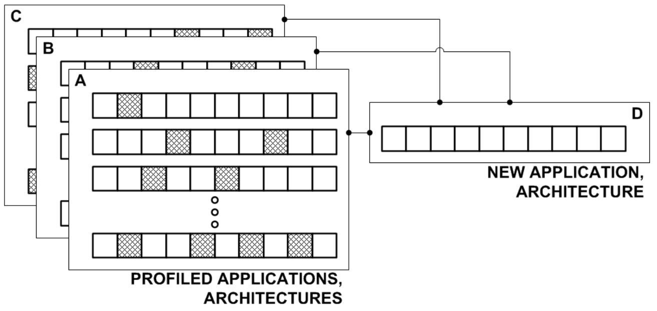
C
B
A
D
NEW APPLICATION, ARCHITECTURE
PROFILED APPLICATIONS, ARCHITECTURES

Figure 2: Portable, microarchitecture-independent characteristics that are profiled for architectures A, B, and C are applicable to new architecture D.

value of an application's profile since part of it is likely relevant for other applications of interest.

In contrast to related work [35], our approach to sharding is agnostic to underlying phase behavior. We simply ensure that shards are shorter than phases so that intra-application diversity is preserved. A short, pre-determined shard length is sufficient. Forgoing sophisticated phase analysis simplifies the mechanics of profiling software characteristics.

2.2. Portable Characteristics

We measure application characteristics that are portable and independent of the microarchitecture [14, 36]. For example, data re-use distance is a portable measure while cache miss rate is not. Such portability is important when profiling tunable codes on reconfigurable cache architectures. We cannot profile every code on every cache configuration [43].

At the system-level, portability is needed to navigate increasing heterogeneity. The Google-wide Profiler samples application behavior across many datacenter nodes [39] . And future node managers must anticipate heterogeneous hardware demand in the form of diverse resource containers [2, 17, 19], contention [30], or big/small cores [38]. Thus, software profiles will be collected from increasingly diverse platforms.

Portable measures of software are needed when sharing profiles between architectures. Figure 2 shows how profiles collected for different shards and architectures might provide insight for a new application on a new architecture. While these links might be found explicitly, perhaps with distance calculations and clustering, the costs of doing so are prohibitive given the number of dimensions in an integrated hardware-software space. Implicitly inferring these links is more tractable.

2.3. Statistical Inference

To infer these links, we extend related work in predictive modeling. Previous models predict performance as a function of parameters from the processor design space. Sampled measurements from the space are used to train neural networks [11, 21, 22] or fit regression models [26, 27]. These efforts infer hardware performance for design space exploration.

In this paper, inferred performance also accounts for software behavior and lays a foundation for run-time decisions. Let z be performance. And let $x = (x_1, \ldots, x_p)$ and $y = (y_1, \ldots, y_q)$ be sets of p hardware parameters and q software characteristics. By sparsely profiling hardware-software interactions, a model can be inferred to predict $z = F(x, y) + \varepsilon$ with some approximation error ε. With statistical regression, we construct an integrated hardware-software model.

2.4. Automated Modeling

Unfortunately, an integrated hardware-software space is unwieldy. To infer a model, we must navigate a space of hardware and software parameters, several non-linear transformations on them, and a combinatorially increasing number of interactions between them. In prior work, users manually specified hardware-only models. But specifying hardware-software models is complicated by the sheer number of variables and requires additional guidance.

We present a heuristic to search for effective model specifications, which are defined by variables, transformations, and interactions. We encode model specifications as genetic sequences, which evolve toward better fits. Unlike stepwise regression, which considers only one term at a time, crossovers and mutation in genetic algorithms support a rapid search of possible models. Moreover, the heuristic accommodates new data by updating the model specification and fitting new regression coefficients. This capability supports dynamic run-time environments with evolving software profiles.

3. Generalized Hardware-Software Models

Accommodating dynamic software behavior is difficult for two reasons. First, software characteristics have high variance and long tails (i.e., infrequent instances of large values), which models have difficulty capturing. Second, profilers have little control over the software behavior used for training.

Nonetheless, modeling software behavior is critically important. Behavior differs dramatically between application phases and between different applications. These differences are exploited by many of the most innovative heterogeneous systems and adaptive architectures. The viability of these innovations depends on linking software dynamics to system and architecture preferences.

Consider a system with diverse software that requests computation from heterogeneous hardware. The space of software behavior is large. Moreover, it is sparsely and non-uniformly populated by real applications. As applications run, models have the difficult task of generalizing trends by profiling software behavior that it cannot precisely sample and manipulate. In comparison, hardware modeling is easier since simulators allow architects to sample uniformly from a cleanly defined design space.

3.1. Inference and Software Behavior

Regression is the starting point for our models. Suppose we have independent variables $x = (x_1, \ldots, x_p)$ for software and $y = (y_1, \ldots, y_q)$

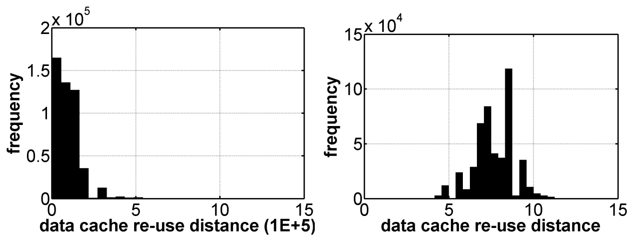

Figure 3: Each shard reports the sum of its re-use distances for 256B data cache blocks. (a) Histogram for this sum-of-distances (x) shows long tail of outliers across SPEC2006 shards. (b) Transforming $x \to x^{1/5}$ stabilizes variance.

for hardware. And we have a dependent variable z for performance. A basic regression fits $z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_{p+q} y_q + \varepsilon$ by finding β's to minimize error given x, y, and z from training data.

In practice, more sophisticated models are needed. Users must determine which independent variables to include (x_i or y_j), how these variables are transformed to accommodate non-linear trends ($S(x_i)$), and which variables interact to affect performance ($x_i y_j$). In each of these decisions, software characteristics introduce new challenges that we address with an automated heuristic for constructing and updating models.

Choosing Variables. In some cases, the impact of software behavior is clear from domain knowledge. For example, rare floating-point divides are not strong predictors of performance. But more generally, we cannot always anticipate the precise mix of applications in a system and determining the best software predictors of performance is complicated.

Further affecting the choice of variables are strong relationships between software characteristics. For example, consider measures of locality. Temporal locality measures time between two consecutive accesses to a cache block. And spatial locality is the quotient of two measures for temporal locality at different block sizes. From an architect's perspective, this link is clear and both locality measures should be modeled.

However, from a statistician's perspective, these locality measures are linearly dependent and highly correlated. Such subtle collinearity, which prevents solvers from fitting a model, is common amongst software variables. Although we use domain expertise to eliminate obvious cases of collinearity, many are not easily discovered until model construction. For this reason, the modeling heuristic must also check for and eliminate collinear variables as it dynamically and automatically seeks a mix of variables with high predictive ability.

Transforming Variables. Once chosen for the model, variables benefit from non-linear transformations that provide flexibility and mitigate the high variance in software behavior, which is particularly important since we cannot explicitly control training samples from the space of software behavior.

To illustrate challenges posed by behavioral asymmetry, Figure 3(a) plots a histogram of temporal locality for SPEC2006 shards. We measure a shard's locality as the sum of all re-use distances within it.[1] While most profiles report small sum-of-distances, many profiles report much larger ones. Outliers are an order of magnitude larger than the common case; sum-of-distances at 5E+4 are most common but sum-of-distances at 5E+5 are observed. Such tails are typical in software.

Because this heteroscedasticity (i.e., non-constant variance) breaks underlying regression assumptions, we apply variance stabilizing transformations. Rather than use x, we use $x^{1/n}$ in the model.[2] With such a transformation, our measure of locality exhibits more symmetry and less variance around the mean as shown in Figure 3(b).

Transformations also provide flexibility to capture non-linear or non-monotonic trends. We can use splines that divide a variable into pieces and fit different cubic polynomials to each piece [18]. For example, $S(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \beta_4 (x-a)_+^3 + \beta_5 (x-b)_+^3 + \beta_6 (x-c)_+^3$ splits x into pieces delimited by three inflections at a, b, and c. Since $(x-a)_+ = max(x-a,\ 0)$, β_4 has an effect only if x is greater than a. In this way, different coefficients are fit to different parts of the space.

These transformations require decisions, such as the exponent in variance stabilization or the number of inflections in splines. The right choice depends on the training data. But since this data evolves in dynamic systems, our modeling heuristic searches the space of transformations and applies the best one automatically.

Specifying Interactions. Lastly, variables interact to affect the predicted value. For example, the performance impact of branch prediction is larger in deeper pipelines because the price of wrong-path execution is higher. Pairwise hardware-software interactions are particularly important since they are a fundamental determinant of performance.

Regression captures such interactions with a product term. In $z = \beta_0 + \beta_1 x + \beta_2 y + \beta_3 xy + \varepsilon$, the interaction between x and y is captured by coefficient β_3. Note the partial derivative $\delta z/\delta x_1 = \beta_1 + \beta_3 x_2$. Given p variables and t possible transformations on those variables, there are $\binom{p \times t}{2}$ possible pairwise interactions. These possibilities are explored and evaluated by the modeling heuristic.

3.2. Accommodating System Dynamics

Training models in dynamic systems is difficult. Because we add software behavior to the model and rely on real applications for sparse profiles, we have little control over training data. Some software behavior may be well represented while others are not. And, in systems with run-time profiling, new software behavior requires model updates.

We describe the modeling process for a large system with diverse architectures and applications. As software runs, sparse run-time profiling collects hardware-software interactions and their effect on performance (e.g., Google-wide Profiler [39]). And in the future, these profiles may be collected on heterogeneous platforms with big/small cores [38], differentiated virtual machines [2], or diverse servers in federated clouds. As data accrues, the model is updated.

An Inductive Analysis. To describe system and model dynamics, we take an inductive approach. In the inductive hypothesis, the system is in steady state. Architectures and applications from spaces H and S have been sparsely profiled. And this data has trained an accurate model M for the integrated H/S space. In practice, this hypothesis holds because models can be boot-strapped with data from benchmark suites.

In the inductive step, the system is perturbed by a new architecture or application.[3] Suppose software space S is perturbed by a new application $+s$. Since the application runs on at least one architecture, there exists a profile with microarchitecture-independent software

[1]Re-use distance is the number of instructions separating two consecutive accesses to the same data block.

[2]$n \geq 1$ and statistics packages, like `ladder` in Stata, help identify the best power transformation.

[3]We use the words "architecture" to represent a hardware environment. A new architecture could arise from new hardware, virtual machines, or contention conditions. Similarly, a new application could arise from new jobs, input data, or code optimizations.

behavior x_{+s}, hardware parameters y, and performance z. With this data, we check the existing model's accuracy, comparing measured performance z against a prediction $M(x_{+s}, y)$.

If predicted performance is accurate, the new application likely shares behavior with already observed software. A sufficiently accurate model will have errors for $+s$ that are competitive with those for applications in S. And, in practice, the desired accuracy depends on how predictions are used. For example, median errors less than 10-15% may be sufficient to make coarse-grained resource allocations.

```
Input: Profiles $P_S$
Output: Regression Model $M$
foreach Generation $g \leq G$ do
    foreach Model $m \in g$ do
        foreach Software $s \in S$ do
            Split $P_s$ data into training $T_s$, validation $V_s$
            Fit $m$ using $\{P_{-s}, T_s\} \times w$
            Set software fitness $f_s$ as $m$'s accuracy on $V_s$
        end
        Set model fitness $f_m$ as $(\sum_{s \in S} f_s)/|S|$
    end
    Populate N% of $g+1$ with $g$'s N% best models
    Populate (100-N)% of $g+1$ with crossovers, mutations
end
```

3.3. Updating System Models

An inaccurate prediction may suggest that the new application is poorly served by existing regression coefficients and/or model specifications. However, the error could also be an outlier. To determine whether to trigger a model update, more data is needed. Profiling $+s$, or variants of it, on a few more architectures would provide additional insight. In practice, we find 10-20 additional data points are sufficient.

Exactly when additional profiles are obtained determines model responsiveness. A model might be updated immediately by invoking profilers for $+s$ on various hardware. Alternatively, because large systems invoke profilers periodically and selectively, a model might be updated only after a sufficient number of additional profiles has accrued. This latter scenario would introduce hysteresis into system models.

To update the model, we insert the new application and its profiles into S. With profiles P_s for each application $s \in S$, we invoke a heuristic to re-specify and perform a weighted fit of the model. This heuristic chooses new variables, transformations, and interactions. To ensure model updates accommodate all profiled applications, the inner loop evaluates the fit of a candidate model for every application ($f_s, s \in S$), which then determines average model fitness (f_m).

3.4. Genetic Search

The heuristic's outer loops implement a genetic search, in which the best models propagate into the next generation while the others are subject to crossovers and mutations. Each model is described by a chromosome that encodes variables, transformations, and interactions.

Each gene encodes a variable. If the genetic value for x_i is 0, the variable is excluded. If the value is 1, 2, or 3, we add x_i with a linear, quadratic, or cubic transformation. And if the genetic value is 4, we apply a piecewise-cubic transformation with three inflection points.

Instruction Mix	
x_1	# Control
x_2	# Taken Branches
x_3	# Float ALU
x_4	# Float Mul/Div
x_5	# Integer Mul/Div
x_6	# Integer ALU
x_7	# Memory
Memory – Temporal Locality	
x_8	average re-use distance for 64B d-cache blocks
x_9	average re-use distance for 64B i-cache blocks
Instruction-Level Parallelism	
x_{10}	# of instructions between floating-point ALU and its consumer
x_{11}	# of instructions between floating-point multiply and its consumer
x_{12}	# of instructions between integer multiply and its consumer
x_{13}	Average basic block size # instructions / # branches

Table 1: Software characteristics that are microarchitecture-independent and profiled for 10M-instruction shards.

Pipeline Parameter		
y_1	Width	1 :: 2x :: 8
y_2	Load/Store queue	11 :: 5+ :: 38
	Physical registers	86 :: 42+ :: 300
	Instruction queue	22 :: 10+ :: 72
	Reorder buffer	64 :: 32+ :: 224
Cache		
y_3	L1 Associativity	1 :: 2x :: 8
	L2 Associativity	2 :: 2x :: 8
y_4	MSHR	1 2 4 6 8
y_5	Data cache size (KB)	16 :: 2x :: 128
y_6	Instruction cache size (KB)	16 :: 2x :: 128
y_7	L2 cache size (KB)	256 :: 2x :: 4096
y_8	L2 latency (cy)	6 :: 2+ :: 14
Functional Unit Number		
y_9	Integer ALU	1 :: 1+ :: 4
y_{10}	Integer Mult/Div	1, 2
y_{11}	Float ALU	1 :: 1+ :: 3
y_{12}	Float Mult	1, 2
y_{13}	Cache Read/Write Port	1 :: 1+ :: 4

Table 2: Hardware parameters that include extreme designs so that models infer interior points more accurately.

The chromosome also encodes interactions, specifying a pair of numbers $i-j$ for interaction $x_i x_j$. Given p variables and t transformations on them, $\binom{p \times t}{2}$ interactions are possible interactions. Because we cannot statically specify a chromosome long enough to accommodate so many interactions, we dynamically expand/shrink its length as the search runs.

The genetic search starts with a random population of models, which evolves with crossovers and mutations. With evolving chromosomes, the heuristic quickly covers a large space of model specifications. Models may be affected by three crossover operators: (C1) single variable randomly exchanged between two chromosomes, (C2) interaction randomly exchanged between two chromosomes, (C3) interaction randomly created using single variables from two chromosomes.

We further consider two mutation operators: (M1) interaction randomly changed for a chromosome, (M2) single variable randomly

	Software Parameters	Hardware Parameters
un-used		y_{12}
linear	x_6, x_8, x_9	y_3, y_4, y_8, y_{10}
poly, degree 2	$x_1, x_4, x_5, x_7, x_{10}$ x_{11}, x_{12}, x_{13}	y_1, y_6
spline, 3 knots	x_2, x_3, x_6	$y_2, y_7, y_9, y_{11}, y_{13}$

Table 3: Transformations after 20 genetic search generations.

changed for a chromosome. Each crossover occurs with 12.5% probability and each mutation occurs with 5% probability, which we find experimentally effective.

4. Evaluating Generalized Models

We evaluate the effectiveness of the modeling heuristic, illustrating its convergence and describing the nature of the produced model. We further demonstrate accuracy, both in steady state and after updates, for integrated hardware-software performance prediction.

4.1. Experimental Methodology

To define an integrated hardware-software space, we consider a spectrum of microarchitectures and key measures of software behavior. We sparsely sample application-architecture profiles to train models.

Software Parameters. We break SPEC2006 applications into shards of 10M dynamic instructions. For each shard, we profile portable measures of software behavior as listed in Table 1. In the datapath, these characteristics capture instruction mix. They also capture instruction-level parallelism via the number of instructions that separate producer and consumer instructions. In the cache hierarchy, the profile captures locality by measuring the number of instructions separating two consecutive accesses to the same data block [40].

These characteristics primarily capture processor-bound workload behavior. Other workloads may require memory or I/O characteristics. For memory-bound workloads, such parameters might include memory hierarchy latencies, memory channel bandwidth, application concurrency, and memory request burstiness. Similar strategies apply for I/O-bound workloads.

Hardware Parameters. Applications are profiled on the diverse microarchitectures listed in Table 2. Such hardware diversity may manifest physically in an implemented design, or manifest logically during run-time as partitioning schemes or contention for shared resources. To collect profiles, these microarchitectures must support introspective performance counters.

We embed such counters into Gem5 [1], extending the simulator to profile software behavior during the commit pipeline stage, which ensures that software behavior is independent of the out-of-order microarchitecture. Gem5 simulates the Alpha instruction set and we cross-compiled the following SPEC2006 applications: *astar, bwaves, bzip2, gemsFDTD, hmmer, omnetpp, sjeng*.

Just as large system profilers selectively profile hardware-software pairs [39], we sample the integrated hardware-software space to produce profiles. With profiled data and R statistics libraries, the modeling heuristic fits a regression model [18]. For performance, we parallelize the genetic search with R libraries `doMC` and `Multicore`, which automatically fork and join R threads.

4.2. Automated Modeling

An initial collection of random models may produce a few with reasonable accuracy. And as these models evolve toward better specifications, errors fall. In practice, useful models begin appearing

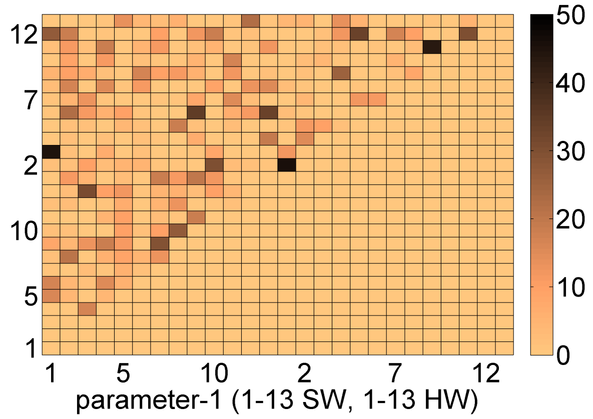

Figure 4: Frequency of interactions in the 50 best models after 20 generations of a genetic search. Interactions are shown between software parameters (lower-left), software-hardware parameters (upper-left), and hardware parameters (upper-right). Matrix is symmetric and we show upper triangle without loss of generality.

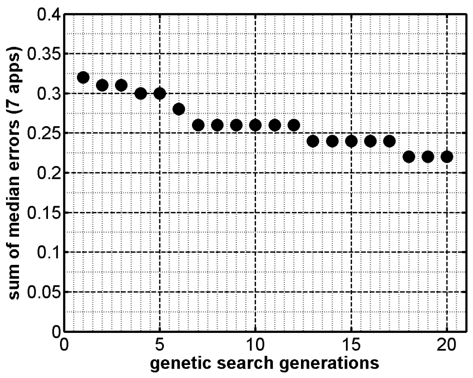

Figure 5: Accuracy improves genetic algorithm evolves for 20 generations. Median errors summed for 7 applications used by a genetic algorithm.

after only a few generations. We see diminishing marginal benefits to accuracy as the search approaches 20 generations. Figure 5 illustrates the benefit to our heuristic's measure of accuracy: sum of median errors for applications of interest.

Comparison with Manual Modeling. A research assistant with no prior experience in regression requires nearly ten months to produce an integrated hardware-software model by hand. Much of this time is spent identifying variables, transformations and interactions.

Moreover, a manually specified model is susceptible to human biases, which limit the number of candidate models he considers. Automatically derived models benefit from a comprehensive search. We find that model errors from genetic search are 10% lower than those from hand-tuning.

Modeling Time. The speed of parallelized and automated genetic search significantly reduces the time to find and fit an accurate model. As long as the heuristic's three nested loops are ordered to minimize data movement, we find that twelve processor cores provide a 9× speedup.

Moreover, the genetic algorithm's inner loop is embarrassingly parallel. Statistical computation for each candidate model in a generation is independent; a generation with n models could benefit from n-way parallelism.

With such speed, the search can evaluate one generation every 20 minutes. In practice, we expect even faster training times. As the search begins with more effective models in the starting population, fewer generations are required. And each generation is evaluated more quickly as more cores provide greater parallelism.

Parameter Significance. System analysts benefit, not only from speed and accuracy, but also from an additional source of insight as the genetic search identifies determinants of performance. The search begins with a population of random models. But as models evolve, the population increasingly prefers certain variables, transformations, and interactions.

Some parameters, such as those that support out-of-order execution (y_2), have complex relationships with performance that require sophisticated spline transformations. Other parameters, such as the number of floating-point multiplies (y_{12}) is less significant and dropped from the model. Consider a genetic algorithm evolved for more than 20 generations using SPEC 2006 shard profiles. After the search converges, we examine the best models and Table 3 presents common transformations.

In practice, hardware-software interactions are sophisticated and span many different parameters. However, the model accommodates only pairwise interactions. The genetic search must use many such pairs to capture the desired effect. Specifically, the two-dimensional histogram of Figure 4 indicates how often a particular pairwise interaction appears in the 50 best models. After 20 generations, the best models still exhibit considerable diversity in its choice of pairwise interactions.

4.3. Accuracy in Steady State – Interpolation

We evaluate model accuracy in two scenarios. In the first, the system is in steady state. An integrated hardware-software space has been sparsely profiled to construct a model that interpolates performance. As illustrated in Figure 6(a), interpolation lends itself to accurate models. For every prediction made, we have likely profiled similar hardware for another application or profiled similar software for another architecture.

Interpolation Accuracy. To evaluate interpolation accuracy, we train and validate a model. First, we randomly sample architectures. For each architecture, we randomly sample applications. The number of samples is many orders of magnitude smaller than the cross-product of applications and architectures. With these sparse samples, the automated heuristic produces a model. On average, each of 7 applications is profiled on 360 architectures.

We assess accuracy in two ways. First, we examine the distribution of prediction errors with boxplots, which show the median and quartiles computed over the validation data. Second, we consider the correlation between predicted and true performance, which is a better measure of accuracy in the context of optimization. For example, correlation is important in hill climbing heuristics that use models to find higher performance.

Training data is randomly selected from application-architecture pairs. Validation data is select randomly and independently of the training data. Validation against 140 separately profiled application-architecture pairs illustrates the effectiveness of our automated modeling heuristic. The resulting model has low median errors of 5% in Figure 7(a) and high correlation coefficients of $\rho > 0.9$ in Figure 8(a).

Reduced Profiling Costs. Not only is the integrated hardware-software model accurate, it requires less data to train when compared against prior approaches in regression and neural networks [21, 26]. Previously, each application would require its own architectural model and 400-800 architectural profiles to train it.

With our integrated approach, we require fewer architectural samples per application to construct a single model shared by all applications. Shared software behavior reduces the number of required profiles. If applications s_1 and s_2 exhibit similar software behavior, each benefits from the other's architectural profiles. By exploiting such shared behavior, profiling costs per application falls by $2-4\times$. Cost reductions are even greater (20-40$\times$) when existing profiles are used to extrapolate new application or architecture performance.

4.4. Accuracy after Updates – Extrapolation

Shared shard behavior is the basis for extrapolation and the second modeling scenario. In this scenario, the system is perturbed by a new architecture, new application, or both. By exploiting similar behavior in existing profiles, models can be updated inexpensively to extrapolate performance as illustrated in Figure 6(b-d).

Extrapolation for Shards. We first evaluate the notion of shard similarity by extrapolating individual shard performance. Profiles of shards from $n-1$ applications train a model, which is used to predict the performance of shards from application n. Each SPEC2006 application takes a turn as application n; the other $n-1$ applications train.

Accurate shard-level predictions indicate exploitable relationships across application shards. For example, *astar* shard performance is predicted accurately by sparse shard profiles from $\{bwaves, \ldots, sjeng\}$. We validate against 300 separately profiled shards for each application. Figure 10 shows low median errors of 8%. Moreover, predictions correlate with true performance values; $\rho \geq 0.9$.

Extrapolation for Applications/Architectures. Automatic model updates further enhance accuracy when the system is perturbed by a new application and/or architecture. As illustrated in Figure 6(d), a system in steady state has sampled shard profiles from $n-1$ applications on diverse architectures. These profiles produce an accurate hardware-software model for interpolation. When perturbed by application n, the model is updated (§3.2–§3.4).

To predict application performance, we predict the performance of its constituent shards and aggregate their contributions to the application. The performance for most shards can be extrapolated accurately. A few inaccurate shard predictions have a small effect on the end-to-end prediction since an application contains many 10M instruction shards.

We first consider systems perturbed by variants of existing applications, which perform the same fundamental computation but differ in code structure or input data. These differences alter the dynamic instruction stream, significantly affecting both performance and the underlying microarchitecture-independent characteristics we profile. For example, we find the choice of back-end compiler optimizations affect performance by up to 60%; mean effect is 26%.

Consider a system perturbed by applications with code optimizations (`-O1`,`-O3`) or input data (`-v1`,`-v2`,`-v3`) that differ from those in existing profiles. For these software variants, updated models accurately predict performance for 150 application-architecture pairs. Median errors are 8% in Figure 7(b). Correlation coefficients $\rho \geq 0.9$ in Figure 8(b).

Beyond the common perturbations of varying software, systems may also encounter fundamentally new software. In such scenarios,

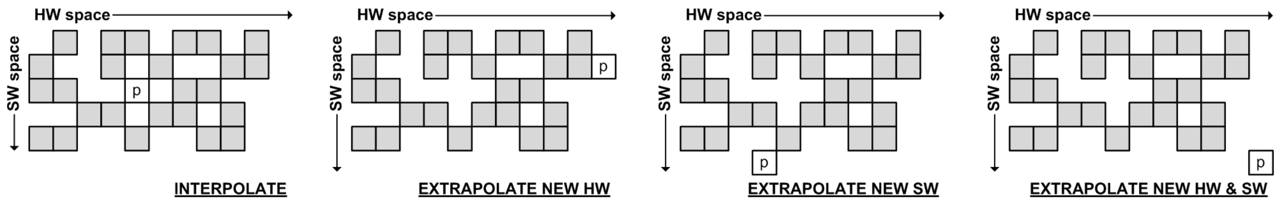

Figure 6: Shaded HW-SW pairs are profiled. Pair p denotes a prediction. (a) Interpolation for previously observed hardware and software. (b)-(d) Extrapolation after updates for new hardware, new software, or both.

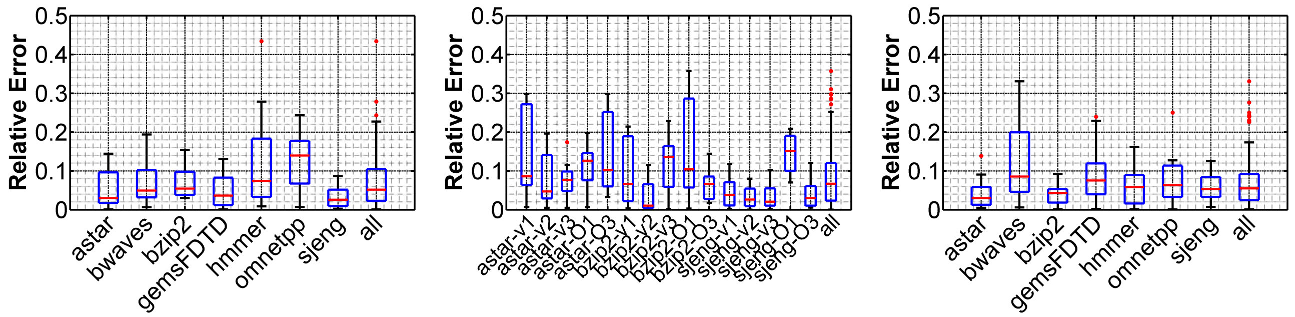

Figure 7: Distributions of performance prediction error when (a) interpolating in steady state, (b) extrapolating for new software variant, (c) extrapolating for new hardware/software.

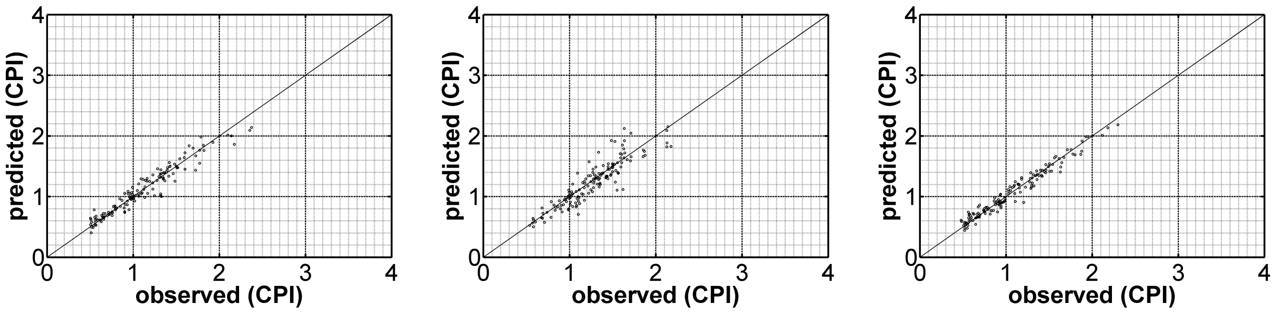

Figure 8: Correlation between predicted and true performance when (a) interpolating in steady state, (b) extrapolating for new software variant, (c) extrapolating for new hardware/software.

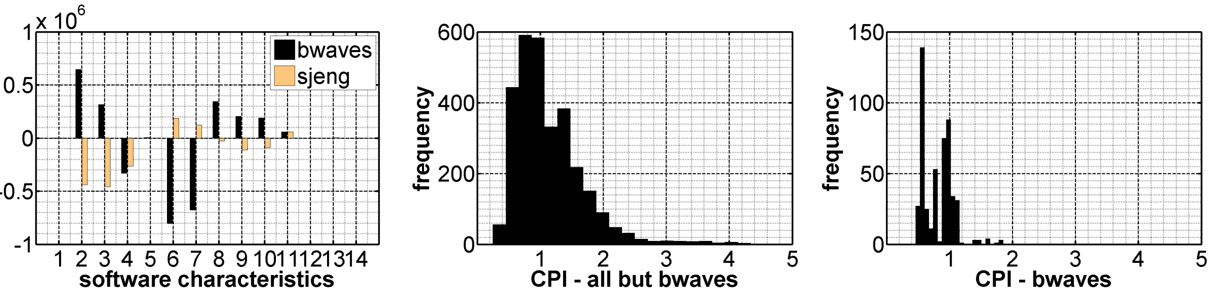

Figure 9: Extrapolation for bwaves suffers from significant differences in software behavior and performance. (a) illustrates the difference between training data mean and bwaves/sjeng mean for various software characteristics; x-axis refers to Table 2. (b) plots CPI distribution for all applications excluding bwaves and (c) plots a very different CPI distribution for bwaves.

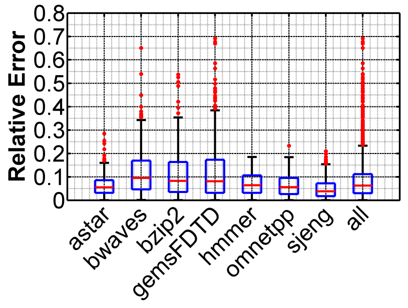

Figure 10: Error distribution when predicting shard performance. Model is trained and validated with separate, randomly sampled shards.

extrapolation reaches further beyond the already profiled space. To assess accuracy, each SPEC2006 application takes a turn as application n; the other $n-1$ train.

We predict the performance of application n for 140 new application-architecture pairs. While inherently more difficult than interpolation, extrapolation with updated models captures performance trends with low median errors of 6% in Figure 7(b) and strong correlations between predicted and true values; $\rho \geq 0.9$ in Figure 8(b).

4.5. Outliers

Extrapolation is more difficult when already profiled software behavior does not cover those in the target application. For example, compare performance extrapolation for *sjeng* and *bwaves*. Software behavior in *sjeng* is very similar to that in the $n-1$ other applications. In contrast, *bwaves* exhibits very different behavior.

Figure 9 quantifies this difference. For each software characteristic of Table 1, we take the mean value observed for a given application and subtract the mean value observed for its training applications. If this difference is zero, application n behaves like the other $n-1$ applications. However, if this difference is large, the training data is not representative of the target application.

Figure 9(a) indicates that, while *sjeng* differences are modest, *bwaves* is not well represented by its training data. Compared to training applications, *bwaves* has far more taken branches and floating-point operations. And it has far fewer integer and memory operations.

Such large behavioral differences translate into performance differences. Figure 9(b-c) show CPI histograms for *bwaves*' shards versus those in all the other applications. While performance of other applications' shards are clustered around CPI=1, *bwaves* CPI exhibits much greater variance and bimodal behavior around CPI = 0.5 and 1.0. Even model updates cannot accommodate this difference in performance distribution.

However, these challenges are not fundamental and, in an avenue for future work, training data can be augmented to better cover the space of software behavior. Synthetic benchmarks provide explicit control on software behavior and enable uniform profiling across the software space [23]. If synthetic benchmarks were used, they would need to be coordinated with real application profiles.

5. Domain-Specific Models

To model performance from instruction-level software behavior, §3–§4 develop an extensive methodology to accommodate generality. This generality is expensive. Capturing performance requires many measures of software behavior that then require automated modeling heuristics. Identifying determinants of performance is difficult without application semantics.

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 & a_{14} & a_{15} \\ 0 & 0 & a_{22} & 0 & a_{24} & a_{25} \\ 0 & 0 & 0 & a_{33} & a_{34} & a_{35} \end{pmatrix}$$

$$\texttt{b\_row\_start} = (0\ \ 2\ \ 4); \qquad \texttt{b\_col\_idx} = (0\ \ 4\ \ 2\ \ 4)$$

$$\texttt{b\_value} = (a_{00}\ \ a_{01}\ \ a_{10}\ \ a_{11}\ \ 0\ \ 0\ \ a_{14}\ \ a_{15}\ \ a_{22}\ \ 0\ \ 0\ \ a_{33}\ \ a_{24}\ \ a_{25}\ \ a_{34}\ \ a_{35})$$

Figure 11: BCSR with 2×2 blocks. 2×2 blocks are stored contiguously in `b_value`. The first column index of entry (1,1) in each 2×2 block is stored in `b_col_idx`. Pointers to block row starting positions in `b_col_idx` are stored in `b_row_start`.

Given domain knowledge, however, we can express software behavior more concisely. Rather than analyze individual instructions, we can analyze the behavior of libraries and algorithms. This approach is gaining traction in a number of domains, such as signal processing [37], linear algebra [3, 44], and program sketching [42]. From these diverse domain-specific program generators, we choose sparse matrix-vector multiply (SpMV [44]) for a case study.

5.1. Sparse Matrix-Vector Multiply (SpMV)

SPMV poses interesting challenges for integrated hardware-software models. Its performance trends are non-monotonic and its hardware-software interactions are sophisticated. We capture these subtleties while exploiting program-level software characteristics that keep model complexity modest.

Sparse matrix-vector multiply (SpMV) computes $v = v + Au$ when most elements in A are zero. We refer to u and v as the source and destination vectors, respectively. SpMV performance tuning is complicated by irregularity and variation in the best choice of sparse matrix data structure and code transformation across matrices and machines.

A particularly effective transformation improves locality by organizing a sparse matrix into $r\times c$ sub-blocks. Blocks with at least one non-zero are stored. The multiply proceeds block by block, re-using c source elements and streaming through r destination elements in a row-major matrix.

5.2. SpMV Hardware-Software Interactions

Choosing a matrix block size requires navigating sophisticated trade-offs between hardware and software. Sparse matrices store non-zero values with row and column pointers that map each value to a location in the matrix. Index overheads are reduced in a blocked compressed format since indices point to matrix blocks instead of individual matrix values (Figure 11).

However, this reduction is offset by explicit zeros which must be stored to create dense blocks of the same size; Figure 11 stores four such zeros. The fill ratio is the number of stored values (original non-zeros plus filled zeros) in a blocked matrix divided by the original number of non-zeros. Fill increases the number of unnecessary floating-point operations.

The precise balance of blocking's costs and benefits depends on input data, block size, and cache architecture. Sparser matrices and larger block sizes require more filled zeros to produce dense structure. And as density improves spatial locality, SpMV benefits from larger cache lines.

There exists an optimal balance between fill and locality. But we must strike this balance in an irregular performance topology. Rather than profile the cross-product of sparse matrix patterns, block sizes,

	Matrix	Dimension	Non-Zeros	Sparsity
1	3dtube	45330	1629474	7.93E-04
2	bayer02	13935	63679	3.28E-04
3	bcsstk35	30237	740200	8.10E-04
4	bmw7st	141347	3740507	1.87E-04
5	crystk02	13965	491274	2.52E-03
6	memplus	17758	126150	4.00E-04
7	nasasrb	54870	1366097	4.54E-04
8	olafu	16146	515651	1.98E-03
9	pwtk	217918	5926171	1.25E-04
10	raefsky3	21200	1488768	3.31E-03
11	venkat01	62424	1717792	4.41E-04

Table 4: Sparse matrices. N is dimension in a square matrix. Sparsity is number of non-zero elements divided by N^2.

SpMV		
x_1	brow, block row	1 :: 1+ :: 8
x_2	bcol, block column	1 :: 1+ :: 8
x_3	fR, fill ratio	function of brow,bcol,matrix
Cache Architecture		
y_1	lsize, line size	16B :: 2x :: 128B
y_2	dsize, data size	4KB :: 2x :: 256KB
y_3	dways, data ways	1 :: 2x :: ::8
y_4	drepl, data repl	LRU, NMRU, RND
y_5	isize, inst size	2KB :: 2x :: 128KB
y_6	iways, inst ways	1 :: 2x :: ::8
y_7	irepl, inst repl	LRU, NMRU, RND

Table 5: Hardware-software space, which includes software block sizes and hardware cache parameters.

and cache architectures, we infer models that accurately capture these complex relationships for coordinated optimization.

5.3. Accuracy for Coordinated Optimization

Software and Hardware Space. The integrated hardware-software space spans matrices, block sizes, and cache architectures. Table 4 lists matrices drawn from various application domains [34]. SpMV for each matrix has 64 variants with block sizes from 1×1 to 8×8, each with a corresponding fill ratio. These codes are generated automatically by OSKI [44].

Given the diversity in matrices and data structures, we evaluate a reconfigurable architecture that can accommodate them. To accurately assess core and cache reconfigurability, we use a simulator for a 400MHz Tensilica Xtensa processor, supplemented with CACTI and Micron to estimate cache and memory power [31, 33]. Because SpMV is memory-intensive, we focus on the cache (Table 5).

Non-monotonic Performance. Performance in this integrated hardware-software space is highly irregular.[4] Matrix blocking has a direct but discontinuous performance impact. Locality and performance increase with block size. But the largest blocks require the greatest number of filled zeros and produce diminishing marginal returns.

Figure 12 illustrates this irregularity in an example matrix, *raefsky3*. 8 block rows maximize performance but 6 or 7 block rows are only as effective as 2. For block columns, 1, 4, and 8 are equally effective, which reflects inherent dense sub-structure that arises in multiples of 4. A poorly chosen block size increases the fill ratio, which can harm performance (e.g., fR > 1.25).

[4]Performance is true floating-point operations per second. The numerator excludes operations on filled zeros. The denominator includes reduced execution time from blocking.

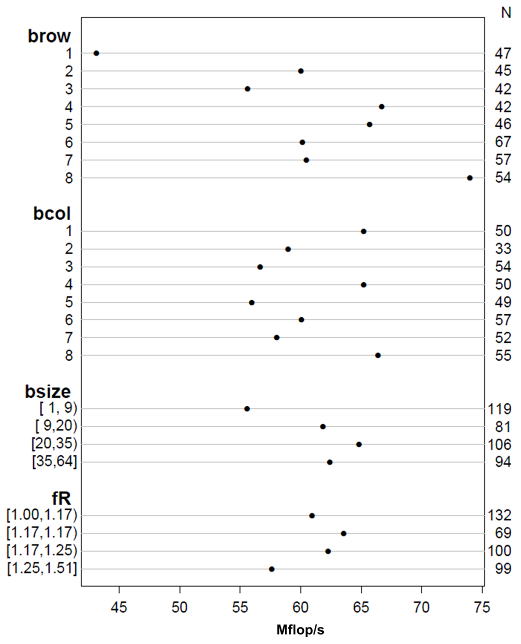

Figure 12: SpMV blocking parameters and performance. Data includes 400 samples drawn from integrated SpMV-cache space for *raefsky3*. Average Mflop/s is reported for all samples at each parameter value.

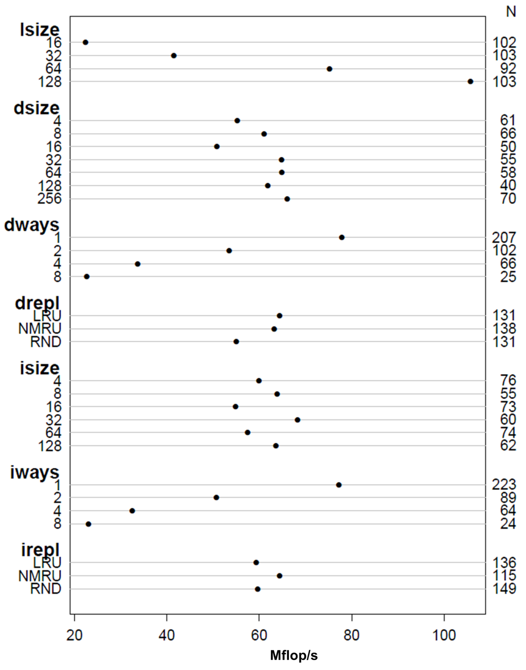

Figure 13: Cache architecture and performance trends. Data includes 400 samples drawn from integrated SpMV-cache space for *raefsky3*. Average Mflop/s is reported for all samples at each parameter value.

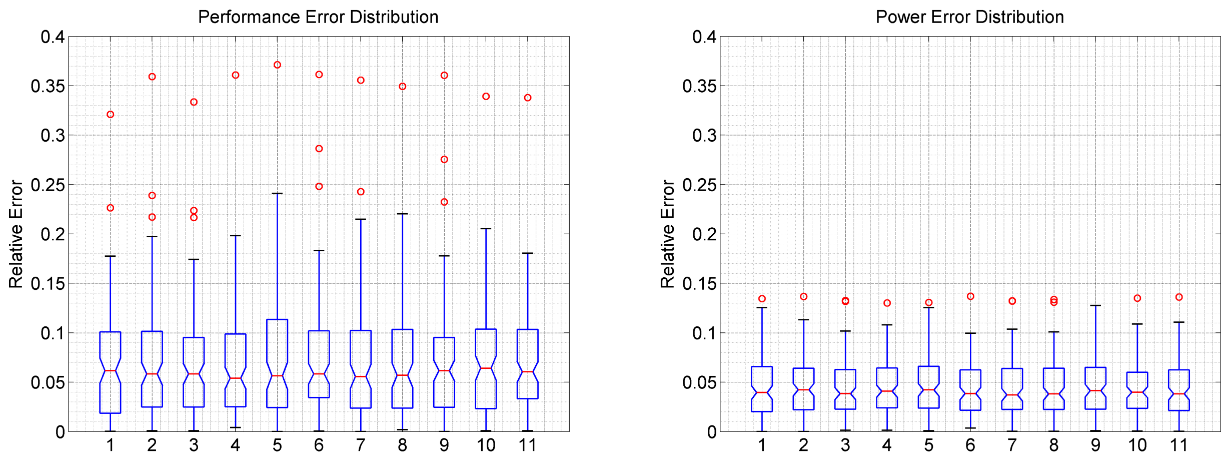

Figure 14: Distributions of (a) performance and (b) power prediction error. Matrix numbers refer to Table 4.

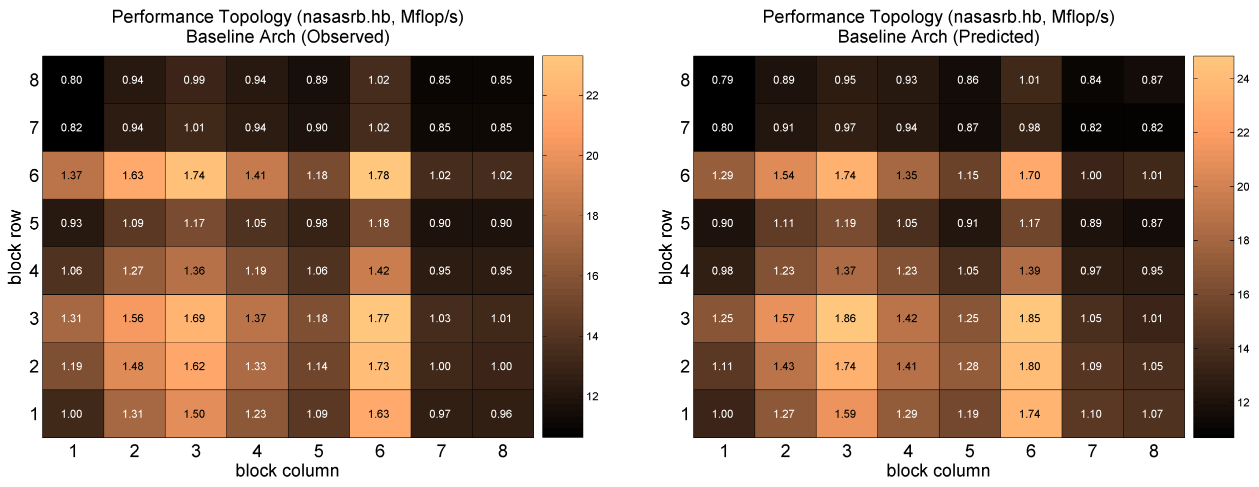

Figure 15: (a) Profiled and (b) predicted performance topologies. Colormap illustrates Mflop/s and numbers in each cell indicate speedup over 1×1 code. Data shown for a representative matrix *nasasrb*.

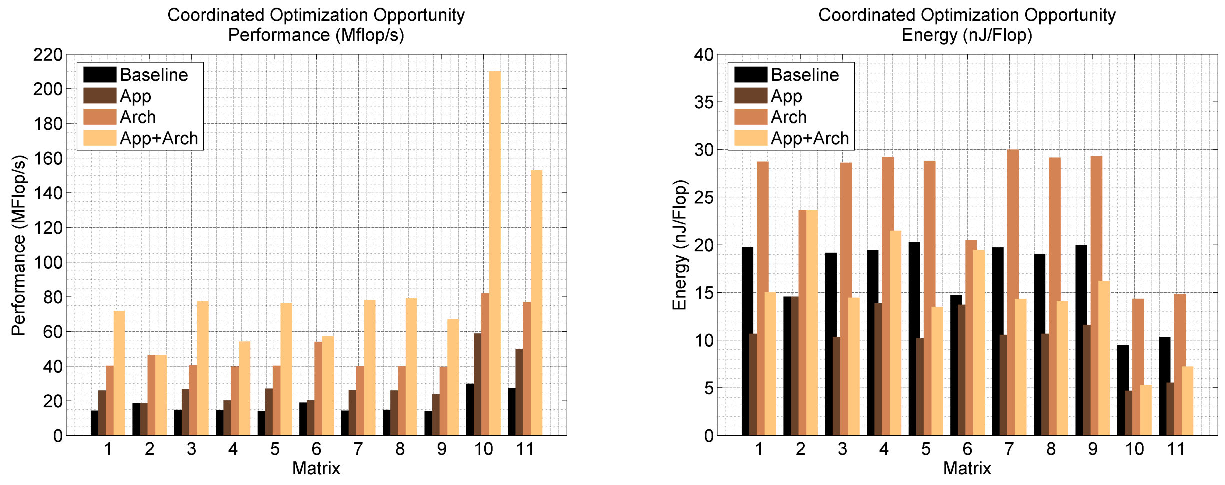

Figure 16: (a) Performance and (b) energy efficiency optimization. Matrix numbers refer to Table 4.

These blocking effects interact with cache structure, as shown in Figure 13. A larger cache line amortizes off-chip latency over a larger number of bytes to increase streaming bandwidth. Ideally, matrix values would not be cached since they are never re-used. But in a highly associative cache, matrix blocks occupy cache lines longer as they must travel down the LRU stack.

SpMV-Cache Modeling Accuracy. We train and validate a model for these complex performance effects. First, we randomly sample combinations of matrix block sizes and cache architectures. With these sparse samples, we fit regression models that predict performance and power using SpMV specific software parameters.

Such domain-specific parameters encapsulate much more information than generic instruction-level profiles. Rather than measure locality with re-use distances, SpMV block sizes directly quantify the amount of exploitable locality. Models use fewer, semantic-rich parameters to greater effect. With 400 sparsely sampled profiles for training and another 100 for validation, performance and power are accurately predicted with median errors between 4-6% across 11 matrices. Accuracy is shown in Figure 14.

Moreover, for a representative matrix *nasasrb*, Figure 15 illustrates the accuracy of inferred models on an irregular performance topology. The model accurately captures high performance at the same block sizes (3×3, 3×6, 6×3, 6×6). The model also captures discontinuities; many block sizes adjacent to 6×6 are worse than not blocking at all.

Coordinated Optimization. Advances in parameterized code generators and reconfigurable architectures make navigating an integrated hardware-software space imperative. In many domains, we can choose to tune the application, the architecture, or both. We compare these strategies for SpMV, exploiting the tractability of inferred models.

For SpMV, application tuning identifies the best matrix block size. Sparse matrices that have dense sub-structure (e.g., matrices 10 and 11) benefit from larger block sizes and modest fill ratios. Because SpMV is memory-bound, architecture tuning tailors the cache. Larger cache lines amortize off-chip memory latency. In Figure 16(a), application and architecture tuning improve performance by 1.6× and 2.7×, respectively. When tuned together, performance improves by 5.0×.

However, these tuning strategies incur different costs. Application tuning improves performance while simultaneously reducing energy as blocking improves locality and reduces the number of expensive memory accesses. With less data movement, both latency and energy fall. The optimal block size reduces energy from 17 to 11 nJ/Flop in Figure 16(b).

In contrast, energy increases to 25 nJ/Flop with architecture tuning. Larger cache lines increase the number of memory transfers, which cost 6nJ per 64b double-precision word [31]. This cost is difficult to justify unless the matrix is blocked to increase spatial locality. With coordinated tuning, energy per floating-point operation falls by 0.9× (i.e., 10% reduction) even as performance increases by 5.0×.

Collectively, these results motivate new thinking on efficiency. Architects cannot afford to ignore application tuning, which increases performance and reduces energy. For such tuning, inferred models provide tractability and insight.

6. Related Work

Hardware performance is modeled with statistical regression [26, 28] or neural nets [21]. Dubach et al. and Khan et al. separately construct models that predict new applications as a weighted combination of others by defining canonical machines and software profiles on them [11, 10, 25]. Instead of profiling each application on a few pieces of canonical hardware, we embed fundamental measures of software behavior into an integrated hardware-software model. Alternatively, analytical models link instruction behavior with pipeline structure [15, 24] and can help coordinate multiprocessor management [5].

We leverage prior work in microarchitecture-independent software characteristics to parameterize our model [6, 14, 20, 36]. We encounter challenges with variance in real application behavior, which might be mitigated with synthetic benchmarks controlled to produce uniform training data [13, 23]. Sampling identifies short, representative instruction segments within an application to predict its overall performance [41, 45]. Alternatively, a robust approach to experimental design might identify the subset of architectural simulations required to understand performance trends [46]. Like these prior works, we sample to reduce measurement costs but we do so uniformly at random. Moreover, our software samples profile behavior of fine-grained shards, which are shorter and more diverse than coarse-grained application phases.

Hardware-software co-tuning has been applied to dense matrix-matrix multiply, sparse matrix-vector multiply, and stencil computation [32]. Instead of exploratory profiling, we construct a model and reduce co-tuning costs. Similar models would benefit code generators and optimizers in a variety of application domains, including signal processing [37], linear algebra[3, 44], sorting [29], and back-end compilation [8].

Beyond software tuning for specific application domains, prior work has studied other predictors of software performance, such as how often particular methods are called and which compiler optimizations are applied. Chun et al. extract application-specific features (e.g., method invocation counts) from program analysis to predict performance but these features do not generalize across different applications [7]. Dubach et al. identifies more general predictors of software performance, some of which overlap with ours [9]. Navigating complex, back-end compiler optimizations is difficult and prior work has analyzed their impact on software performance [4, 16].

Many of these prior efforts focus entirely on performance variability in the software space for a given hardware platform. A notable exception, Dubach et al. coordinate the choice of compiler optimizations to the architectural design [12]. Rather than use compiler flags as predictors of software performance, we rely on sharded application behavior or domain-specific tuning parameters, allowing us to apply models beyond optimizing compilers.

7. Conclusions

We demonstrate a model for general applications by relying on shared behavior across application shards. By inferring performance models from software characteristics and hardware parameters, we better understand software preferences for hardware. Moreover, with modeling heuristics, such models can be updated automatically to reflect system dynamics.

Further accuracy is possible with domain-specific code generators. As application and architecture designers pursue performance and efficiency together, we need frameworks that maintain the integrity of abstraction layers while building new bridges across them. We preserve abstractions by synthesizing descriptors of software structure (e.g., matrix block size, fill ratio) and understanding their interactions with underlying hardware.

There are a number of directions for future work. In future, inferred models will need to accommodate virtual machines, which are prevalent in large, datacenter systems. We must determine whether modeling strategies change when hardware resources are virtualized and shared amongst co-located applications. These strategies may further extend to memory and I/O systems. Models for such systems require new parameters to characterize software behavior. By broadening the system scope, these models can be made even more relevant for datacenter management and big data computation.

Acknowledgements

This work is supported by NSF grant CCF-1149252 (CAREER). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] —. gem5. http://www.m5sim.org/.

[2] Amazon. Elastic cloud computing. http://aws.amazon.com/ec2/.

[3] J. Ansel et al. PetaBricks: a language for compiler and algorithmic choice. In *PLDI*, 2009.

[4] J. Cavazos et al. Automatic performance model construction for the fast software exploration of new hardware designs. In *CASES*, 2006.

[5] J. Chen and L. John. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *ICS*, 2011.

[6] J. Chen, L. John, and D. Kaseridis. Modeling program resource demand using inherent program characteristics. In *SIGMETRICS*, 2011.

[7] B. Chun, L. Huang, S. Lee, P. Maniatis, and M. Naik. Mantis: Predicting system performance through program analysis and modeling. *Computing Research Repository (CoRR)*, 2010.

[8] K. Cooper, D. Subramanian, and L. Torczon. Exploring the structure of the space of compilation sequences using randomized search algorithms. *The Journal of Supercomputing*, 36(2), 2006.

[9] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF*, 2007.

[10] C. Dubach, T. Jones, E. Bonilla, G. Fursin, and M. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *MICRO*, 2009.

[11] C. Dubach, T. Jones, and M. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *MICRO*, 2007.

[12] C. Dubach, T. Jones, and M. O'Boyle. Exploring and predicting the architecture/optimising compiler co-design space. In *CASES*, 2008.

[13] L. Eeckhout, K. DeBosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *ISPASS*, 2000.

[14] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IISWC*, 2005.

[15] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, 2006.

[16] G. Fursin and O. Temam. Collective optimization: A practical collaborative approach. *TACO*, 2010.

[17] A. Ghodsi et al. Dominant resource fairness: Fair allocation of multiple resource types. In *Proc. NSDI*, 2011.

[18] F. Harrell. *Regression Modeling Strategies*. Springer, 2001.

[19] B. Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. NSDI*, 2011.

[20] K. Hoste et al. Performance prediction based on inherent program similarity. In *PACT*, 2006.

[21] E. Ipek, S. McKee, B. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS*, 2006.

[22] P. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *MICRO*, 2006.

[23] A. Joshi, L. Eeckhout, L. John, and C. Isen. Automated microprocessor stressmark generation. In *HPCA*, 2008.

[24] T. Karkhanis and J. Smith. Automated design of application specific superscalar processors: An analytical approach. In *ISCA*, 2007.

[25] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra. Using predictive modeling for cross-program design space exploration in multicore systems. In *PACT*, 2007.

[26] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS*, 2006.

[27] B. Lee and D. Brooks. Illustrative design space studies with microarchitectural regression models. In *HPCA*, 2007.

[28] B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *MICRO*, 2008.

[29] X. Li et al. A dynamically tuned sorting library. In *CGO*, 2004.

[30] J. Mars et al. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.

[31] Micron. Technical note TN-47-04: Calculating memory system power for DDR2. In *www.micron.com*, 2006.

[32] M. Mohiyuddin et al. A design methodology for domain-optimized power-efficient supercomputing. In *SC*, 2009.

[33] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO*, 2006.

[34] NIST. Matrix market. http://math.nist.gov/MatrixMarket/.

[35] E. Perelman et al. Using SimPoint for accurate and efficient simulation. In *SIGMETRICS*, 2003.

[36] A. Phansalkar et al. Analysis of redundancy and application balance in SPEC CPU 2006 benchmark suite. In *ISCA*, 2007.

[37] M. Pueschel et al. SPIRAL: Code generation for DSP transforms. *Proc. IEEE*, 2005.

[38] V. Reddi et al. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *ISCA*, 2010.

[39] G. Ren et al. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 2010.

[40] X. Shen, Y. Zhang, and C. Ding. Locality phase prediction. In *ASPLOS*, 2004.

[41] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[42] A. Solar-Lezama, R. Rabbah, R. Bodk, and K. Ebcioglu. Programming by sketching for bitstreaming programs. In *PLDI*, 2005.

[43] A. Solomatnikov et al. Using a configurable processor generator for computer architecture prototyping. In *MICRO*, 2009.

[44] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *SciDAC, Journal of Physics*, 2005.

[45] R. Wunderlich et al. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA*, 2003.

[46] J. Yi, D. Lilja, and D. Hawkins. A statistically rigorous approach for improving simulation methodology. In *HPCA*, 2003.