SiblingRivalry: Online Autotuning Through Local Competitions

Jason Ansel   Maciej Pacula   Yee Lok Wong   Cy Chan   Marek Olszewski
Una-May O'Reilly   Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{jansel,mpacula,ylwong,cychan,mareko,unamay,saman}@csail.mit.edu

ABSTRACT

Modern high performance libraries, such as ATLAS and FFTW, and programming languages, such as PetaBricks, have shown that autotuning computer programs can lead to significant speedups. However, autotuning can be burdensome to the deployment of a program, since the tuning process can take a long time and should be re-run whenever the program, microarchitecture, execution environment, or tool chain changes. Failure to re-autotune programs often leads to widespread use of sub-optimal algorithms. With the growth of cloud computing, where computations can run in environments with unknown load and migrate between different (possibly unknown) microarchitectures, the need for online autotuning has become increasingly important.

We present SiblingRivalry, a new model for always-on online autotuning that allows parallel programs to continuously adapt and optimize themselves to their environment. In our system, requests are processed by dividing the available cores in half, and processing two identical requests in parallel on each half. Half of the cores are devoted to a known safe program configuration, while the other half are used for an experimental program configuration chosen by our self-adapting evolutionary algorithm. When the faster configuration completes, its results are returned, and the slower configuration is terminated. Over time, this constant experimentation allows programs to adapt to changing dynamic environments and often outperform the original algorithm that uses the entire system.

Categories and Subject Descriptors

I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; D.3.4 [**Programming Languages**]: Processors—*Compilers*

Keywords

Autotuning, Evolutionary Algorithm, Genetic Algorithm

1. INTRODUCTION

Autotuning is becoming one of the most effective methods for gaining efficient performance on complex applications which run on modern hardware systems. Libraries such as Atlas [35], FFTW [19], and SPARSITY [24]; frameworks such as PERI [36], SPIRAL [28], and Green [7]; and languages such as PetaBricks [3], allow the programmer to set up an application to be autotuned for a given microarchitecture.

However, while offline autotuning provides great performance gains, it has two major problems. First, it adds an additional step to the software installation and upgrade process. Second, offline autotuning is unable to construct programs that respond to dynamically changing conditions. As we will show, changes to machine load can substantially degrade an application's performance. When such changes occur, an offline autotuned algorithm may no longer be the best choice. This situation is further exacerbated in the emerging cloud and data center computing environments, where in addition to sharing a machine with varying load, applications may be transparently migrated between machines, and thus potentially between microarchitectures. Such changes to computer architectures and microarchitectures have been shown to lead to significant performance loss for autotuned applications [3].

In response to some of these challenges, there is a growing body of work [7, 9, 12, 22, 23, 25] focused on creating applications that can monitor and automatically tune themselves to optimize a particular objective (e.g. meeting response time goals by trading quality of service (QoS) for increased performance or lower power usage). In order to provide stability, convergence and predictability guarantees, many of these systems construct (either by hand, or automatically) a linear model of their application and employ control theory techniques to perform dynamic tuning. The success of such techniques depends on the degree to which the configurable choices can be mapped to a linear system, a task that can be difficult when tuning large complex applications with interdependent configuration choices.

In contrast, offline evolutionary (a.k.a. genetic) autotuning techniques, such as the one used in PetaBricks [3], are model-free. They adaptively sample the search space of candidate solutions and take advantage of both large and small moves in the search space. Thus, they are able to find global optima regardless of how non-linear or interdependent

the choice space and can do so without a model. Their selection component allows them to improve execution time even though they generate random variations on current solutions with unpredictable performance. When a specific variation is extremely slow, in an offline setting, it can be killed and prevent a sampling bottleneck. Unfortunately, this characteristic has meant that evolutionary techniques are generally considered to be unsuitable for use in the online setting. Executing multiple generations of a sizable population at runtime (even at periodic intervals) is too costly to be feasible. Additionally, the alternative approach of continuously replacing the components being executed by different experimental variations, offered by an autotuner, is also a poor choice as there is no execution standard to compare the variation against. Thus, the learning system has no way of knowing whether a particular variation is performing particularly poorly and thus should be aborted.

In this paper, we take a novel approach to online learning that enables the application of evolutionary tuning techniques to online autotuning. Our technique, called *SiblingRivalry*, divides the available processor resources in half and runs the current best algorithm on one half and a variation on the other half. If the current best finishes first, the variation is killed, the failure of the variation is reported to the online learning algorithm which controls the selection of both configurations for such "competitions" and the application continues to the next stage. If the variation finishes first, we have found a better solution than the current best. Thus, the current best is killed and the results from the variation are used as the program continues to the next stage. Using this technique, SiblingRivalry produces predictable and stable executions, while still exploiting an evolutionary tuning approach. The online learning algorithm is capable of adapting to changes in the environment and progressively identifies better configurations over time without resorting to experiments that might deliver extremely slow performance. As we will show, despite the loss of resources, this technique can produce speedups over fixed configurations when the dynamic execution environment changes. To the best of our knowledge, SiblingRivalry is the first attempt at employing evolutionary tuning techniques to online autotuning computer programs.

We have implemented a prototype of the SiblingRivalry algorithm within the context of the PetaBricks language [3, 4]. Our results show that SiblingRivalry's always-on racing technique can lead to an autotuned algorithm that uses only half the machine resources (as the other half is used for learning) but that is often faster than an optimized algorithm that uses the entire processing resources of the machine. Furthermore, we show that SiblingRivalry dynamically responds and adapts to changes in the runtime environment such as system load.

1.1 Contributions

SiblingRivalry makes the following contributions:

- To the best of our knowledge, the first general technique to apply evolutionary tuning algorithms to the problem of online autotuning of computer programs.

- A new model for online autotuning where the processor resources are divided and two candidate configurations compete against each other.

- A multi-objective, practical online evolutionary learning algorithm for high-dimensional, multi-modal, and non-linear configuration search spaces.

- A scalable learning algorithm for high-dimensional search spaces, such as those in our benchmark suite which average 97 search dimensions.

- Support for meeting dynamically changing time or accuracy targets which are in response to changing load or user requirements.

- Experimental results showing a geometric mean speedup of 1.8x when adapting to changes in microarchitectures and a 1.3x geometric mean speedup when adapting to moderate load on the system.

- Experimental results showing how, despite accomplishing more work, SiblingRivalry can actually reduce average power consumption by an average of 30% after a migration between microarchitectures.

1.2 Use cases

We envision a number of common use cases for our online learning techniques:

- *Adapting to dynamic load*: Production code is usually run not in isolation but on shared machines with varying amounts of load. Yet it is impossible for offline training to pre-compute a best strategy for every type of load. SiblingRivalry enables programs to dynamically adapt to changing load on a system. It ensures continual good performance and eliminates pathological cases of interference due to resource competition.

- *Migration in the cloud:* In the cloud, the type of machine on which a program is running is often unknown. Additionally, the virtual machine executing a program can be live migrated between systems. SiblingRivalry allows programs to dynamically adapt to these circumstances as the architecture changes underneath them.

- *Dynamically changing accuracy targets:* Depending on the situation, a user may need varying levels of accuracy (or quality of service) from an application. SiblingRivalry allows the user to dynamically change either the accuracy or performance target of an application. It supports trading-off execution time with accuracy.

- *Deploying to a wide variety of machines:* SiblingRivalry greatly simplifies the task of deploying an application to a wide variety of architectures. It enables a single centralized configuration, perhaps on a shared disk, to be deployed. This is followed by online customization for each machine on the network.

- *Reducing over-provisioning requirements hardware resources:* Data centers must often over-provision resources to handle rare load spikes. By supporting dynamic changes to desired accuracies during load spikes, SiblingRivalry can reduce the amount of required over-provisioning.

2. PETABRICKS LANGUAGE

The PetaBricks language provides a framework for the programmer to describe multiple ways of solving a problem while allowing the autotuner to determine which of those ways is best for the user's situation [3]. It provides both algorithmic flexibility (multiple algorithmic choices) as well as coarse-grained code generation flexibility (synthesized outer control flow).

At the highest level, the programmer can specify a *transform*, which takes some number of inputs and produces some number of outputs. In this respect, the PetaBricks transform is like a function call in a procedural language. The major difference is that we allow the programmer to specify multiple pathways to convert the inputs to the outputs for each transform. Pathways are specified in a dataflow manner using a number of smaller building blocks called *rules*, which encode both the data dependencies of the rule and C++-like code that converts the rule's inputs to outputs.

One of the key features of the PetaBricks programming language is support for variable accuracy algorithms, which can trade output accuracy for computational performance (and vice versa) depending on the needs of the user. Approximating ideal program outputs is a common technique used for solving computationally difficult problems, adhering to processing or timing constraints, or optimizing performance in situations where perfect precision is not necessary. Algorithmic methods for producing variable accuracy outputs include approximation algorithms, iterative methods, data resampling, and other heuristics. A detailed description of the variable accuracy features of PetaBricks is given in [4].

2.1 The PetaBricks Autotuning Setup

Choices are represented in a configuration file that contains three types of structures. The first type is selectors which allow the autotuner to make algorithmic choices. Selectors can make different decisions when called with different input sizes dynamically. Using this mechanism, selectors can be used by the autotuner to construct poly-algorithms that dynamically switch techniques at recursive call sites. Formally, a selector s consists of $\vec{C_s} = [c_{s,1}, \ldots, c_{s,m-1}] \cup \vec{A_s} = [\alpha_{s,1}, \ldots, \alpha_{s,m}]$ where $\vec{C_s}$ are the ordered interval boundaries (cutoffs) associated with algorithms $\vec{A_s}$. During program execution the runtime function $SELECT$ chooses an algorithm depending on the current input size by referencing the selector as follows:

$$SELECT(input, s) = \alpha_{s,i} \text{ s.t. } c_{s,i} > size(input) \geq c_{s,i-1}$$

where $c_{s,0} = 0$ and $c_{s,m} = \infty$. The components of $\vec{A_s}$ are indices into a discrete set of applicable algorithmic choices available to s, which we denote $Algorithms_s$.

The synthesized functions in the configuration file define continuous functions that specify a transform parameter that varies with input size. Each synthesized function is defined by a series of points $\vec{S_g} = [s_{g,0}, \ldots, s_{g,31}]$ which define the value of the synthesized function at exponentially increasing sample points with linear interpolation between the defined points. The value of the parameter at any dynamic transform call is defined by the function:

$$SYNTHFUNC(input, g) = \frac{s_{g,a}(2^b - n) + s_{g,b}(n - 2^a)}{2^b - 2^a}$$

where $n = size(input)$ and $a = \lfloor \lg_2 n \rfloor$ and $b = \lceil \lg_2(n+1) \rceil$.

In addition to these algorithmic choice selectors and synthesized functions, the configuration file contains many other discrete tunable parameters. These tunable parameters include things such as blocking sizes for local memory, sequential/parallel cutoffs, and user defined parameters. Each tunable parameter is an integer with a positive bounded range.
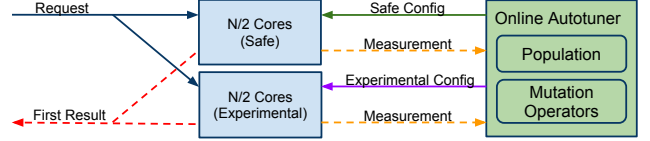
3. COMPETITION EXECUTION MODEL

Figure 1: High level flow of the runtime system. The data on dotted lines may not be transmitted for the slower configuration, which can be terminated before completion.

Figure 1 shows the high level flow of how requests are processed by the PetaBricks runtime system. The cores on our system are split in half into two groups. One group of cores is designated to run safe configurations, while the other group runs experimental configurations. When a request is received, the autotuner runs the same request on both groups of cores in parallel using a safe configuration on one group and an experimental configuration on the other group. When the first configuration completes (and provides a satisfactory answer) the system terminates the slower one. The output of the better algorithm is returned to the user, and timing and quality of service measurements are sent to the autotuner so that it may update its population of configurations and mutation operator priorities.

3.1 Other Splitting Strategies

Our racing execution model requires that there be two groups of cores, one that executes an experimental configuration, while the other executes a safe configuration. While we have chosen to divide our resources in a 50/50 split, other divisions (such as 60/40 or 75/25) are possible.

We do not consider splits where we devote fewer cores to the experimental group than the safe group since doing so would prevent some superior configurations from completing (they would be killed immediately after the safe strategy completes). Further, tuning for fewer than half of the cores limits the potential benefits from autotuning.

One of the reasons we chose a 50/50 split over other possible splits was to minimize the gap between best-case and worst-case overheads that result from splitting. Splits that devote very few resources to the safe configuration will incur larger costs when the experimental configuration fails compared to when it succeeds.

Another major advantage of the 50/50 split is that it provides more data to the autotuner, since the performance of both tests can be compared directly. In uneven configurations, very little is learned about the configuration on the smaller part of the chip, since even if it is a better configuration it still may be aborted before completion. This means that the online learner is expected to converge more quickly in the 50/50 case.

3.2 Time Multiplexing Races

Another racing strategy is to run the experimental configuration and the safe configuration in sequence rather than in parallel. This allows both algorithms to utilize the entire machine. It also provides a way to, in some cases, avoid running the safe configuration entirely. These types of techniques are also the most amenable to at some point switching off online learning, if one knows that the dynamic execution environment has stabilized and the learner has converged.

There are two variants to this type of technique:

- *Safe configuration first.* In this variant, the safe configuration is run first, and is always allowed to complete, using the entire machine. Then the experimental configuration is allowed an equal amount of time to run, to see if it would have completed faster. Unfortunately, this method will incur a $2x$ overhead in the steady state, which is the same as the expected worst case for running the races in parallel (assuming linear scalability). For this reason this technique is only desirable if one plans to disable online learning part way through an execution.

- *Experimental configuration first.* In this variant, a model is required to predict the performance of a configuration given a specific input and current dynamic system environment. The model predicts the upper bound performance of the safe configuration. The experimental configuration is given this predicted amount of time to produce an answer before being terminated. If the experimental configuration produces an acceptable answer, then the safe configuration is never run, otherwise the system falls back to the safe configuration.

  The efficacy of this technique depends a lot on the quality of the model used and the probability of the learning system producing bad configurations. In the best case, this technique can have close to zero overhead. However, in the worst case, this technique could both fail to converge and produce overheads exceeding 2x. If the performance model under-predicts execution time, superior configurations will be terminated prematurely and autotuning will fail to make improvements. If the performance model over-predicts execution time, then the cost of exploring bad configurations will grow. For our problem, the probability of a bad configuration is high enough that this type of technique is not desirable, however, with search spaces with more safer configurations this technique may become more appealing.

4. SIBLINGRIVALRY ONLINE LEARNER

The online learner is an evolutionary algorithm (EA) that is specially designed for the purpose of identifying, online, the best configuration for the program. It has a multitude of exacting requirements: It must be lightweight because it is always running. It cannot add significant computational or memory overhead to the application or it will diminish the overall value of autotuning. It must conduct its search in accordance with the structure of the pairwise competition execution model as described in Section 3. Accordingly, it must effectively search and adapt candidate solutions by offering competition configurations and integrating the feedback from their measurement results. Because the competition execution model is processing real requests, it must provide at least one configuration that is sufficiently safe to ensure quality of service. Despite the search space of candidate configurations being very large, it must converge to a high quality configuration quickly. It must not assume the underlying environment is stationary. It must converge in the face of high execution time variability (due to load variance) and react to system changes in a timely way without being notified of them.

To meet its convergence goals, the online learner, in effect, must ideally balance exploration and exploitation in its search strategy. Exploitation should investigate candidates in the "neighborhood" of currently high performing configurations. Exploration should investigate candidates that are very different from the current population to ensure no route to the optimum has been overlooked by the greedy nature of exploitation. This final required property of the online learner motivates one of its key capabilities. The online learner performs "adaptive mutator selection" which we explain in more detail in Section 4.5.

4.1 High Level Function

In the process of tuning a program, the online learner maintains a population of candidate configurations. The population is relatively small to minimize the computational and memory overhead of learning.

The online learner keeps two types of performance logs: per-configuration and per-mutator. Per-configuration logs record runtime, accuracy, and confidence for a given candidate, and are used by the learner to select the "safe" configuration for each competition, and to prune configurations which are demonstrably worse. Per-mutator logs record performance along the three objectives for candidates generated by a given mutator. This information allows the online learner to select mutators which have a record of producing improved solutions, using a process called Adaptive Operator Selection (see Section 4.5 for more information).

Whenever the program being tuned receives a request, the online learner selects two configurations to handle it: "safe" and "experimental". The safe configuration is the configuration with the highest value of the fitness function (see Section 4.3) in the current population, computed using per-configuration logs. The fitness value captures how well the configuration has performed in the past, and thus the safe configuration represents the best candidate found by the online learner so far. The experimental configuration is produced by drawing a "seed" configuration from the current population and transforming it using a mutator. The probability of a configuration being selected as a seed is proportional to its fitness.

Once the safe and experimental configurations have been selected, the online learner uses both to process the request in parallel, and returns the result from the candidate that finishes first and meets the accuracy target (the "winner"). The slower candidate (the "loser") is terminated. If the experimental configuration is the winner, it is added to the online learner's population. Otherwise, it is discarded. The safe configuration is added back to the population regardless of the result of the race, but might be pruned later if the new result makes it worse than any other candidate.

4.2 Online Learner Objectives

The online learner optimizes three objectives with respect to its candidate configurations:

- *Execution time:* the expected execution time of the algorithm.

- *Accuracy:* the expected value of a programmer metric measuring the quality of the solution found.

- *Confidence:* a metric representing the online learner's confidence in the first two metrics. This metric is 0 if there is only one sample and

$$Confidence = \frac{1}{stderr(\text{timings})} + \frac{1}{stderr(\text{accuracies})}$$

  if there are multiple samples. This takes into account any observed variance in the objective. If the observed variance were constant, the metric would be proportional to $sqrt(T)$ where T is the number of times the candidate has been used.

Confidence is an objective because we expect the variance in the execution times and accuracies of a configuration (as it performs more and more competitions) to be significant. Confidence allows configurations with reliable performance to be differentiated from those with highly variable performance. It prevents an "outlier run" from making a suboptimal configuration temporarily dominate better configurations and forcing them out of the population.

Taken together, these objectives create a 3-dimensional space in which each candidate algorithm in the population occupies a point. In this 3-dimensional space, the online learner's goal is to push the current population towards the Pareto-optimal front.

4.3 Selecting the Safe and Seed Configuration

Each configuration of the population is assigned a fitness, m, that is updated every time it competes against another configuration. Fitness depends upon how well the configuration is meeting a target accuracy, m_a, and its execution time, m_t:

$$m_{config} = \left\{ \begin{array}{ll} \frac{-m_t}{\sum_{n \in P} n_t} - z \frac{g - m_a}{\sum_{n \in P} n_a} & \text{if } m_a < g \\ \frac{-m_t}{\sum_{n \in P} n_t} & \text{if } m_a \geq g \end{array} \right\}$$

where g is a target accuracy, z is a scalar weight set based on how often the online learner has been meeting its goals in the past, and P is the population of all candidates. Fitness prioritizes meeting the accuracy target, but gives no reward for accuracy exceeding the target.

To select the safe configuration, the online learner picks the algorithm in the population that has the highest fitness. When the online learner is not producing configurations that meet the targets, the weight of z is adaptively incremented to put more importance on accuracy targets when it calculates m.

To select a seed configuration, the online learner first eliminates any configuration that has an expected running time that is below the 65th percentile running time of the safe configuration. Then, it randomly draws a configuration from the remaining population using the fitness of each configuration to weight the draw. In evolutionary algorithm terminology, this type of draw is called "fitness proportional selection".

4.4 Mutation Operators

The online learner changes configurations of candidate algorithms though a pool of mutation operators that are generated automatically from information outputted by the PetaBricks compiler. Mutators create a new algorithm configuration from an existing configuration by randomly making changes to a specific target region of the configuration.

One can divide the mutators used by our online learner into the following categories:

- **Selector manipulation mutators** randomly either add, remove, or change levels of a selector decision tree. A decision tree is an abstract hierarchically ordered representation of the selector parameters in the configuration file. It enables the dynamic determination of which algorithm to use at a specific dynamic point in program execution. Each level of the tree has a cutoff value and an algorithmic choice. Each decision tree in the configuration results in 5 mutation operators: one operator to add a level, one operator to remove a level, one operator to make large random changes, and two operators to make small random changes.

- **Log-normal random scaling mutators** scale a configuration value by a random number taken from a log-normal distribution with scale of 1. This type of mutator is used to change cutoff values that are compared to data sizes. Examples of this are blocking sizes, cutoffs in decision trees, and cutoffs to switch between sequential and parallel code.

- **Uniform random mutators** replace an existing configuration value with a new value taken from a discrete uniform random distribution containing all legal values for the configuration item. This type of mutator is used for choices where there are a relatively small number of possibilities. An example of this is deciding the scheduling strategy for a specific block of code or algorithmic choices.

- **Synthesized function manipulation mutators** change the underlying parameter of a function that is used to decide a value that must change dynamically based on input size. For example, the number of iterations in a `for_enough` loop. These functions are represented by $\lg n$ points in the configuration value with runtime interpolation find values lying between the specified points.

4.5 Adaptive Mutator Selection (AMS)

The evolutionary algorithm of the online learner uses different mutators. This provides it with flexibility to generate experimental configurations that range from being close to the seed configuration to far from it, thus controlling its exploration and exploitation. However, the efficiency of the search process is sensitive to *which* mutators are applied and *when*. These decisions cannot be hard coded because they are dependent on what program is being autotuned. Furthermore, even for a specific program, they might need to change over the course of racing history as the population changes and converges. Mutators that cause larger seed-experiment configuration differences should be favored in

early competitions to explore while ones that cause smaller differences should be favored when the search is close to the best configuration to exploit.

For this reason, the online tuner has a specific strategy for selecting mutators on the basis of how well they have performed. The performance of mutators is the extent to which they have generated experimental configurations of better fitness than others. In general, this is called "Adaptive Operator Selection" (AOS) [14, 15, 31] and our version is called "Adaptive Mutator Selection" (AMS).

There are two parts to AMS: credit assignment to a mutator, and mutator selection. AMS uses *Fitness-based Area-Under-Curve* for its credit assignment and a *Bandit* decision process for mutator selection. We use *Fitness-based Area-Under-Curve* because it is appropriate for the comparison (racing) approach taken by the online learner. We use the AUC version of the *Dynamic Multi-Armed Bandit* decision process because it matches up with the online learner's dynamic environment. Our descriptions are adapted and implemented directly from [29].

Credit Assignment

After each competition the AOS stops and assigns credit to operators based on their performance over the interval. *Fitness-based Area-Under-Curve* adapts the Area Under the ROC Curve criteria [11] to assign credit to comparison-based assessment of mutators by first creating a ranked list of the experimental configurations generated in any time window according to a fitness objective. The ROC (Receiver Operator Curve) associated to a given mutator, μ, is then drawn by scanning the ordered list, starting from the origin: a vertical segment is drawn when the current configuration has been generated by μ, a horizontal segment is drawn otherwise, and a diagonal one is drawn in case of ties. Finally, the credit assigned to mutator, μ, is the area under this curve (AUC).

Bandit Mutator Selection

The bandit-based mutator selection deterministically selects the mutator based on a variant of the Upper Confidence Bound (UCB) algorithm [5]:

$$\text{Select } \arg\max_i \left(AUC_{i,t} + C\sqrt{\frac{2\log \sum_k n_{k,t}}{n_{i,k}}} \right)$$

where $AUC_{i,t}$ denotes the empirical quality of the i-th mutator during a user-defined time-window W (exploitation term), $n_{i,t}$ the total number of times it has been used since the beginning of the process (the right term corresponding to the exploration term), and C is a user defined constant that controls the balance between exploration and exploitation. Bandit algorithms have been proven to optimally solve the exploration vs. exploitation dilemma in a stationary context. The dynamic context is addressed in this formulation by using AUC as the exploitation term. See [29] for more details.

4.6 Population Pruning

Each time the population has an experimental configuration added, it is pruned. Pruning is a means of ensuring the experimental configuration should appropriately stay in the population and removing any configuration wholly

Acronym	Processor Type	Operating System	Processors
Xeon8	Intel Xeon X5460 3.16GHz	Debian 5.0	2 (×4 cores)
Xeon32	Intel Xeon X7560 2.27GHz	Ubuntu 10.4	4 (×8 cores)
AMD48	AMD Opteron 6168 1.9GHz	Debian 5.0	4 (×12 cores)

Table 1: Specifications of the test systems used and the acronyms used to differentiate them in results.

inferior to the experimental configuration. The experimental configuration should stay if, for any weighting of its objectives, it is better than any other configuration under the same weighting. This condition is expressed as:

$$\arg\max_{m\in P}\left(\frac{w_a}{\sum_{n\in P} n_a}m_a - \frac{w_t}{\sum_{n\in P} n_t}m_t + \frac{w_c}{\sum_{n\in P} n_c}m_c\right)$$

where P is the population and w defines a weight. The subscripts $_a$, $_t$, and $_c$ of w represent the accuracy, time, and confidence objectives for each configuration.

If the experimental configuration results in an extant configuration no longer being non-dominated, the extant configuration is pruned. We set $w_t = 1 - w_a$ and sample values of w_a and w_c in the range $[0, 1]$. We sample the time-accuracy trade-off space more densely than the confidence space, with approximately 100 different weight combinations total.

5. EXPERIMENTAL RESULTS AND DISCUSSION

We evaluate SiblingRivalry with two experimental scenarios. In the first scenario, we use a single system and vary the load on the system. In the second scenario we vary the underling architecture, to represent the effects of a computation being migrated between machines. In both cases we compare to a fixed configuration found with offline tuning that utilizes all cores of the underlying machine.

We performed our experiments on three systems described in Table 1. We refer to these three systems using the acronyms Xeon8, Xeon32, and AMD48. Power measurements were performed on the AMD48 system, using a WattsUp device that samples and stores the consumed power at 1 second intervals.

5.1 Sources of Speedups

The speedups achieved for different benchmarks can come from a variety of sources. Some of these sources of speedup can apply even to the case where the environment does not change dynamically. Different benchmarks obtained speedups for different reasons in different tests.

- Algorithmic improvements are a large source of speedup, and the motivation for this work. When the dynamic environment changes, the optimal algorithmic choices may be different and SiblingRivalry can discover better algorithms dynamically.

- For the variable accuracy benchmarks, additional speedup can be obtained since the online tuner receives runtime feedback on how well it is meeting its accuracy targets. If it observes that it is over delivering on its quality of service target it can opportunistically change algorithms, enabling it to be less conservative than offline tuning. For all tests, both SiblingRivalry and the baseline met the required quality of service requirements.

- SiblingRivalry benefits from a "dice effect," since it is running two copies of the algorithm it has an increased chance of getting lucky and having one configuration complete faster than its mean performance. External events, like I/O interrupts, have a lower chance of affecting both algorithms. This leads to a small speedup, which is a function of the variance in the performance of each algorithm.

- As the number of processing cores continues to grow exponentially, the amount of per core memory bandwidth is decreasing dramatically since per-chip memory bandwidth is growing only at a linear rate [8]. This fact, coupled with Amdahl's law, makes it particularly difficult to write applications with scalable performance. On our AMD48 machine, we found that some benchmarks with high degree of available parallelism exhibit limited scalability, preventing them from fully utilizing all available processors. In cases where the performance leveled off before half of the available processors, the cost of our competition strategy becomes close to zero.
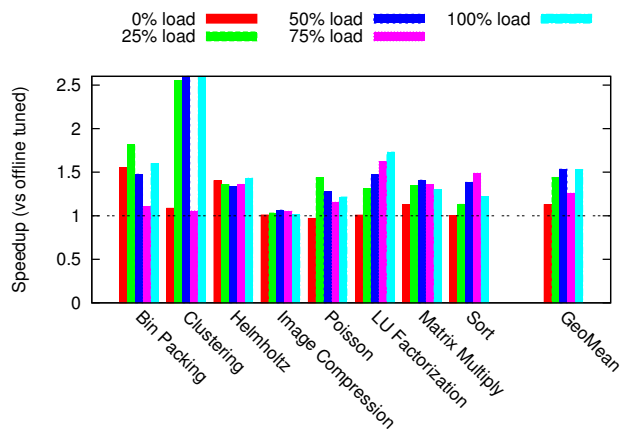
5.2 Load on a System

To test how SiblingRivalry adapts to load on the system, we simulated system load by running concurrently with a synthetic CPU-bound benchmark competing for system resources. We allowed the operating system to assign cores to this benchmark and did not bind it to specific cores. For the different tests, we varied the number of threads in this benchmark to utilize between 0 and 100% of the processors on the system. Combined with the PetaBricks benchmarks, this creates an overloaded system where the number of active threads is double the number of cores. In all cases we compared SiblingRivalry to a baseline of a fixed configuration found with offline tuning on the same machine, without the additional load. We measure average throughput over 10 minutes of execution, which includes all of the learning costs.

We observed different trends of speedups on the two machines tested. On the Xeon8 (Figure 2(a)), the geometric mean cost of running SiblingRivalry (under zero new load) was 16%. This cost is largest for Matrix Multiply, which scales linearly on this system. For other benchmarks, the overheads are lower for two reasons. For the non-variable accuracy benchmarks, some benchmarks do not scale perfectly (These benchmarks exhibit an average speedup of 5.4x when running with 8 threads [3]). For variable accuracy benchmarks, the online autotuner is able to improve performance by taking advantage of using a number of candidate algorithms to construct an aggregate QoS that is closer to the target accuracy level than would be otherwise possible with a single algorithm.

Figure 2(b) shows the performance results on the AMD48 machine. In the zero load case, SiblingRivalry achieves a geometric mean speedup of 1.12x. This speedup comes primarily because of the way the autotuner can dynamically adapt the variable accuracy benchmarks (the same way it did on Xeon8). Additionally, while AMD48 and Xeon8 have very similar memory systems, AMD48 has six times as many cores, and thus 6 times less bandwidth per core. Thus, we found that in some cases, using additional cores on this system did not always translate to better performance.
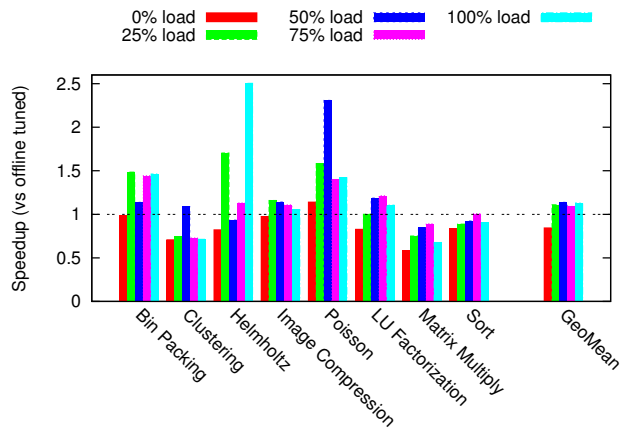
(a) Xeon8

(b) AMD48

Figure 2: Speedups (or slowdowns) of each benchmark as the load on a system changes. Note that the 50% load and 100% load speedups for Clustering in (b), which were cut off due to the scale, are 4.0x and 3.9x.

For example, while some fixed configurations of our matrix multiply benchmark scale well to 48 cores, our autotuner is able to find a less scalable configuration that provides the same performance using only 20 cores. Once load is introduced, SiblingRivarly is able to further adapt the benchmarks, providing geometric speedups of up to 1.53x.

5.3 Migrating Between Microarchitectures

In a second group of experiments we test how SiblingRivalry can adapt to changes in microarchitecture. We first train offline on a initial machine and then move this trained configuration to a different machine. We compare SiblingRivalry to a baseline configuration found with offline tuning on the original machine. The offline configuration is given one thread per core on the system. Figure 3 shows the speedups for each benchmark after such a migration. SiblingRivalry shows a geometric mean speedup of 1.8x in this migration experiment.
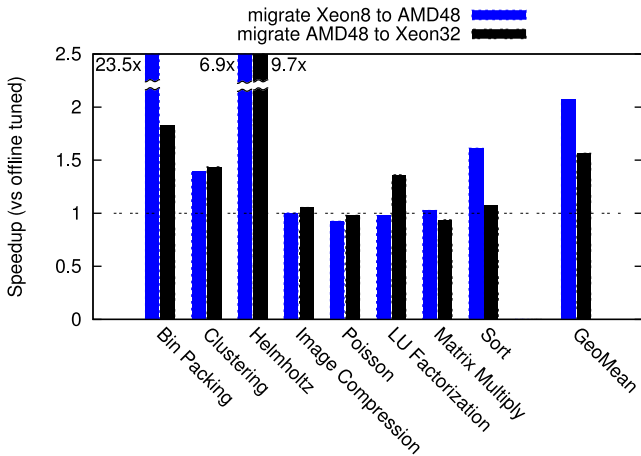
Figure 3: Speedups (or slowdowns) of each benchmark after a migration between microarchitectures. "Normalized throughput" is the throughput over the first 10 minutes of execution of SiblingRivalry (including time to learn), divided by the throughput of the first 10 minutes of an offline tuned configuration using the entire system.

Starting Configuration.

Figure 4 shows how using an offline tuned configuration affects the rate of convergence of SiblingRivalry. We show three starting configurations: a random configuration, a configuration tuned on a different machine, and a configuration tuned on the same machine. As one would expect, convergence time increases as the starting point becomes less optimal. Convergence times are roughly 5 minutes, 1 minute, and 0 for the configurations tried, though since changes are constantly being made it is difficult to mark a point of convergence.

Power Consumption.

Figure 5 shows the energy used per request for each of our benchmarks. While one might initially think that the techniques proposed would increase energy usage since up to twice the amount of work is performed, SiblingRivalry actually reduces energy usage by an average of 30% for our benchmarks. The primary reason for this decreased energy usage is the increased throughput of SiblingRivalry, which results in the machine being used for a shorter period of time. The benchmarks that saw increased throughput also saw decreased power consumption per request.

6. RELATED WORK

A number of offline empirical autotuning frameworks have been developed for building efficient, portable libraries in specific domains. PHiPAC [10] is an autotuning system for dense matrix multiply, generating portable C code and search scripts to tune for specific systems. ATLAS [35] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LAPACK. FFTW [18] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [24] for sparse matrix computations, SPIRAL [28] for digital signal processing, and OSKI [34] for sparse matrix kernels.
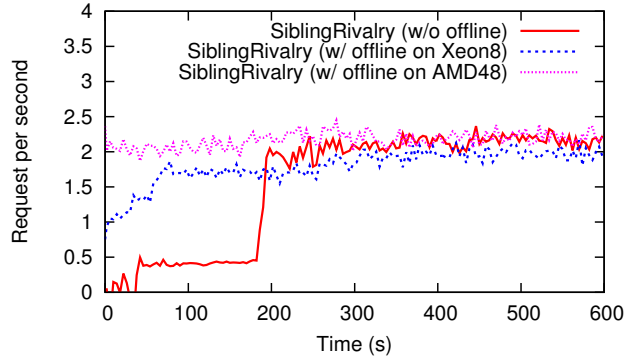

Figure 4: The effect of using an offline tuned configuration as a starting point for SiblingRivaly on the Sort benchmark. We compare starting from a random configuration ("w/o offline") to configurations found through offline training on the same and a different architecture.
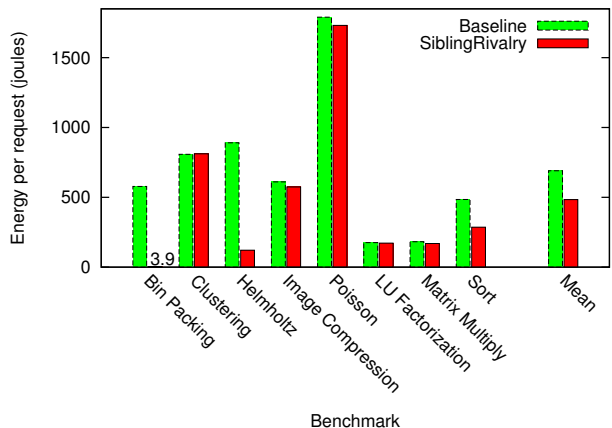

Figure 5: Average energy use per request for each benchmark after migrate Xeon8 to AMD48.

In the dynamic autotuning space, there have been a number of systems developed [7,9,12,22,23,25,30] that focus on creating applications that can monitor and automatically tune themselves to optimize a particular objective. Many of these systems employ a control systems based autotuner that operates on a linear model of the application being tuned. For example, PowerDial [23] converts static configuration parameters that already exist in a program into dynamic knobs that can be tuned at runtime, with the goal of trading QoS guarantees for meeting performance and power usage goals. The system uses an offline learning stage to construct a linear model of the choice configuration space which can be subsequently tuned using a linear control system. The system employs the heartbeat framework [21] to provide feedback to the control system. A similar technique is employed in [22], where a simpler heuristic-based controller dynamically adjusts the degree of loop perforation performed on a target application to trade QoS for performance.

The area of iterative compilation contains many projects that use different machine learning techniques to optimize

lower level compiler optimizations [1, 2, 20, 26]. These projects change both the order that compiler passes are applied and the types of passes that are applied. These projects do not explore the type of algorithmic choices that the PetaBricks language exposes and these systems operate at compile time not runtime.

Additionally, there has been a large amount of work [6, 17, 32, 33] in the dynamic optimization space, where information available at runtime is used combined with static compilation techniques to generate higher performing code. Such dynamic optimizations differ from dynamic autotuning because each of the optimizations is hand crafted in a way that makes it likely that it will lead to an improvement in performance when applied. Conversely, autotuning searches the space of many available program variations without a priori knowledge of which configurations will perform better.

Evolutionary Algorithms Related Work.

There is one evolutionary algorithm, named Differential Evolution (DE) [27], that takes a comparison-based approach to search like our online learner. However DE compares a parent to its offspring, while we compare a safe configuration to the experimental configuration. These two configurations (safe and external) are not related. Further, DE does not generate offspring using mutators.

Our approach to multi-objective optimization is a hybrid of a pareto-based EA [16, 37] and a weighted objectives EA. Our approach avoids the O(nlogn) computational complexity of pareto-based EAs such as the very commonly used NSGA-II [16]. In the latter, these are incurred to identify successive Pareto-fronts and to compute the distance between the solutions on each front. Our approach of using multiple weight combinations and preserving dominating configurations for each is more robust than using only one.

7. CONCLUSIONS

This paper demonstrates that it can sometimes be more effective to devote resources to learning the smart thing to do, than to simple throw resources at a potentially suboptimal configuration. Our technique devotes half of the system resources to trying something different, to enable online adaption to the system environment. The geometric mean speedup of SiblingRivalry was 1.8x after a migration between microarchitectures. Even in comparison to an offline-optimized version on the same microarchitecture that uses the full resources, SiblingRivalry showed a geometric mean performance increase of 1.3x when moderate load was introduced on the machine. These results show that continuously adapting the program to the environment can provide a huge boost in performance that easily overcame the cost of splitting the available resources in half.

In addition, we have showed that an intelligent machine learning system can rapidly find a good solution even when the search space is extremely large. Furthermore, we demonstrated that it is important to provide many algorithmic and optimization choices to the online learner as done by the PetaBricks language and compiler. While these choices increase the search space, they make it possible for the autotuner to obtain the performance gains observed.

SiblingRivalry is able to fully eliminate the offline learning step, making the process fully transparent to users, which is the biggest impediment to the acceptance of autotuning. For example, while Feedback Directed Optimization (FDO) can provide substantial performance gains, the extra step involved in the programmers workflow has stopped this promising technique from being widely adopted [13]. By eliminating any extra steps, we believe that SiblingRivalry can bring autotuning to the mainstream program optimization. As we keep increasing the core counts of our processors, autotuning via SiblingRivalry help exploit them in a purposeful way.

9. REFERENCES

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization*, pages 295–305, 2006.

[2] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES'04*, pages 231–239, 2004.

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *PLDI*, Dublin, Ireland, Jun 2009.

[4] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *CGO*, Chamonix, France, Apr 2011.

[5] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1), 2003.

[6] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *PLDI*, 1996.

[7] Woongki Baek and Trishul Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, June 2010.

[8] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[9] V. Bhat, M. Parashar, . Hua Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed. Enabling self-managing applications using model-based online control strategies. In *International Conference on Autonomic Computing*, Washington, DC, 2006.

[10] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C

coding methodology. In *Supercomputing*, New York, NY, 1997.

[11] Andrew P. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7), 1997.

[12] Fangzhe Chang and Vijay Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing*, 4, March 2001.

[13] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for FDO compilation. In *CGO*, New York, NY, 2010.

[14] Luis DaCosta, Alvaro Fialho, Marc Schoenauer, and Michèle Sebag. Adaptive operator selection with dynamic multi-armed bandits. In *GECCO*, New York, NY, 2008.

[15] Lawrence Davis. Adapting operator probabilities in genetic algorithms. In *ICGA*, San Francisco, CA, 1989.

[16] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, *PPSN*, Berlin, 2000.

[17] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: an effective technique for adaptive computing. In *PLDI*, New York, NY, 1997.

[18] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *IEEE International Conference on Acoustics Speech and Signal Processing*, volume 3, 1998.

[19] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *IEEE*, 93(2), February 2005. Invited paper, special issue on "Program Generation, Optimization, and Platform Adaptation".

[20] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and Francois Bodin. MILEPOST GCC: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, Jul 2008.

[21] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC*, New York, NY, 2010.

[22] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2209-042, Massachusetts Institute of Technology, Sep 2009.

[23] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Power-aware computing with dynamic knobs. In *ASPLOS*, 2011.

[24] Eun-jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *International Conference on Computational Science*, 2001.

[25] Gabor Karsai, Akos Ledeczi, Janos Sztipanovits, Gabor Peceli, Gyula Simon, and Tamas Kovacshazy. An approach to self-adaptive software based on supervisory control. In *International Workshop in Self-adaptive software*, 2001.

[26] Eunjung Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 119 –129, April 2011.

[27] Kenneth Price, Rainer Storn, and Jouni Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Natural Computing Series. Springer-Verlag, Berlin, Germany, 2005.

[28] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1), 2004.

[29] Robert Schaefer, Carlos Cotta, Joanna Kolodziej, and Günter Rudolph, editors. *Parallel Problem Solving from Nature*, volume 6238 of *Lecture Notes in Computer Science*, 2010.

[30] Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automated performance tuning. In *In Proceedings from the Conference on High Performance Networking and Computing*, pages 1–11, 2003.

[31] Dirk Thierens. Adaptive strategies for operator allocation. In Fernando G. Lobo, Cláudio F. Lima, and Zbigniew Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*, volume 54 of *Studies in Computational Intelligence*. 2007.

[32] Michael Voss and Rudolf Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *International Conference on Parallel Processing*, 2000.

[33] Michael Voss and Rudolf Eigenmann. High-level adaptive program optimization with adapt. *ACM SIGPLAN Notices*, 36(7), 2001.

[34] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Scientific Discovery through Advanced Computing Conference*, Journal of Physics: Conference Series, San Francisco, CA, June 2005.

[35] Richard Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing*, Washington, DC, 1998.

[36] S. Williams, K. Datta, J. Carter, L. Oliker, J. Shalf, K. Yelick, and D. Bailey. PERI - auto-tuning memory-intensive kernels for multicore. *Journal of Physics Conference Series*, 125(1), July 2008.

[37] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In K. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou, and T. Fogarty, editors, *Evolutionary Methods for Design, Optimisation and Control*. Barcelona, Spain, 2002.