Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2010-032 July 27, 2010

Language and Compiler Support for Auto-Tuning Variable-Accuracy Algorithms

Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Language and Compiler Support for Auto-Tuning Variable-Accuracy Algorithms

Jason Ansel Yee Lok Wong Cy Chan Marek Olszewski
Alan Edelman Saman Amarasinghe

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

Abstract

Approximating ideal program outputs is a common technique for solving computationally difficult problems, for adhering to processing or timing constraints, and for performance optimization in situations where perfect precision is not necessary. To this end, programmers often use approximation algorithms, iterative methods, data resampling, and other heuristics. However, programming such *variable accuracy* algorithms presents difficult challenges since the optimal algorithms and parameters may change with different accuracy requirements and usage environments. This problem is further compounded when multiple variable accuracy algorithms are nested together due to the complex way that accuracy requirements can propagate across algorithms and because of the resulting size of the set of allowable compositions. As a result, programmers often deal with this issue in an ad-hoc manner that can sometimes violate sound programming practices such as maintaining library abstractions.

In this paper, we propose language extensions that expose trade-offs between time and accuracy to the compiler. The compiler performs fully automatic compile-time and install-time autotuning and analyses in order to construct optimized algorithms to achieve any given target accuracy. We present novel compiler techniques and a structured genetic tuning algorithm to search the space of candidate algorithms and accuracies in the presence of recursion and sub-calls to other variable accuracy code. These techniques benefit both the library writer, by providing an easy way to describe and search the parameter and algorithmic choice space, and the library user, by allowing high level specification of accuracy requirements which are then met automatically without the need for the user to understand any algorithm-specific parameters. Additionally, we present a new suite of benchmarks, written in our language, to examine the efficacy of our techniques. Our experimental results show that by relaxing accuracy requirements, we can easily obtain performance improvements ranging from 1.1x to orders of magnitude of speedup.

1. Introduction

Traditionally, language designers and compiler writers have operated under the assumption that programs require a fixed, strongly defined behavior; however, this is not always the case in practice. For certain classes of applications, such as NP-hard problems or problems with tight computation or timing constraints, programmers are often willing to sacrifice some level of accuracy for faster performance. In this paper, we broadly define these types of programs as *variable accuracy* algorithms. There are many different classes of variable accuracy algorithms spanning many different fields.

One class of variable accuracy algorithms are *approximation algorithms* in the area of soft computing [28]. Approximation algorithms are used to find approximate solutions to computationally difficult tasks with results that have provable quality. For many computationally hard problems, it is possible to find such approximate solutions asymptotically faster than it is to find an optimal solution. A good example of this is BINPACKING. Solving the BINPACKING problem is NP-hard, yet arbitrarily accurate solutions may be found in polynomial time [9]. Like many soft computing problems, BINPACKING has many different approximation algorithms, and the best choice often depends on the level of accuracy desired.

Another class of variable accuracy algorithms are *iterative algorithms* used extensively in the field of applied mathematics. These algorithms iteratively compute approximate values that converge toward an optimal solution. Often, the rate of convergence slows dramatically as one approaches the solution, and in some cases a perfect solution cannot be obtained without an infinite number of iterations [26]. Because of this, many programmers create *convergence criteria* to decide when to stop iterating. However, deciding on a convergence criteria can be difficult when writing modular software because the appropriate criteria may not be known to the programmer ahead of time.

A third class of variable accuracy algorithms are algorithms in the signal processing domain. In this domain, the accuracy of an algorithm can be directly determined from the

problem specification. For example, when designing digital signal processing (DSP) filters, the type and order of the filter can be determined directly from the desired sizes of the stop, transition and pass-bands as well as the required filtering tolerance bounds in the stop and pass-bands. When these specifications change, the optimal filter type may also change. In some cases, it may even be desirable to first re-sample an input, process the signal at a lower sampling rate, and then re-sample it back to the original rate. Since many options exist, determining the best approach is often difficult, especially if the exact requirements of the system are not known ahead of time.

A key challenge when writing and using variable accuracy code arises from maintaining the abstraction boundary between the *library writer* and the *library user*. The library writer understands the algorithm and the various choices and parameters affecting accuracy, but does *not* know the accuracy requirements for each use case. Because special-casing an algorithm for each foreseeable accuracy requirement can be extremely tedious and error-prone, library writers often follow the practice of exposing many internal algorithmic parameters to the interface of their library. Regrettably, while the library user knows the application's requirements, how the exposed implementation-specific parameters and algorithmic choices of the library impact these accuracy requirements may not be clear. Thus, this practice represents a major breakdown in the library abstraction barrier of variable accuracy programs.

This practice is exemplified by the `fmincon()` function in Matlab [1], which attempts to find the minimum of a user-specified nonlinear multivariate function subject to a set of specified constraints. `fmincon()` takes accuracy and optimization options specified by an `options` structure. This structure contains 42 fields that the user can set to specify various options such as which of three algorithms to use, how many iterations to run, and what tolerances to use. Additionally, there are a number of options specific to each of the three algorithms, some of which further affect additional algorithmic choices. For example, the value specified in the `PrecondBandWidth` option used by the `trust-region-reflective` algorithm will indirectly affect both the number of preconditioning iterations performed, as well as the type of factorization algorithm used during the preconditioning phase. This option can have a dramatic effect on the performance of the solver; however, since constructing preconditioners is still an open and active area of research, determining a good choice for this option with respect to one's accuracy needs requires familiarity with state-of-the-art preconditioning techniques.

Yet another challenge arises when a program is constructed by composing multiple variable accuracy modules, or by recursively calling variable accuracy functions. In this type of program, the best top-level performance and accuracy may be obtained by using higher or lower accuracies for intermediate components of the algorithm. In such cases, the space of composition choices and intermediate accuracies becomes extremely large with different compositions having drastically different convergence rates. For example, at each recursive level of a multigrid solver, it may be possible to solve to a lower accuracy (e.g.: by performing fewer iterations of an iterative solver) while still meeting the accuracy requirement of the final solution. However, manually determining what accuracy level to use at each recursive step can be extremely onerous because each of the accuracies in-turn dictate whether to continue recursively or to use iterative or direct solutions. The problem is exacerbated by the fact that *full multigrid* solvers (the version of multigrid usually in use today) use an estimation phase to determine an initial approximation to the solution to accelerate convergence. This estimate is also determined using the recursive multigrid algorithm, and its accuracy directly affects the rate of convergence towards the final solution [8].

In this paper we propose a novel set of language extensions and an accuracy-aware compiler to address the challenges in writing variable accuracy code. With our extensions, accuracy time trade-offs are made visible to the compiler, enabling it to perform empirical autotuning to build optimized algorithms for each accuracy level required by a user. As we will show, these extensions simplify writing variable accuracy code both for the library writer and for the library user.

For the library writer, our compiler automates the otherwise tedious search over both the algorithmic search space and the parameter space to find algorithms with the best performance that meet each required level of accuracy. This is done without forcing the library writer to sacrifice control over how the algorithm operates or how accuracy is achieved. For the library user, our extensions allow the specification of top-level accuracy requirements without the library user needing to understand any parameters and choices that are specific to the implementation details of the algorithm. This helps create a better abstraction barrier that separates the details of the algorithms in a library from the requirements of the library user.

By using an autotuning approach, our compiler is also able to automatically find the best composition of nested calls to variable accuracy code. In this way the compiler is able to construct programs that target the computational power of the algorithm to where it counts most, which can have a significant performance impact on the final algorithm. Finally, the resulting code will perform well across architectures as none of the accuracy-based decisions need to be hard-coded. Instead, when porting to a new platform, the program can be autotuned again without requiring programmer intervention.

We have implemented our language extensions in the context of the PetaBricks programming language and compiler [3]. PetaBricks is a programming language that allows

algorithmic choice to be expressed at the language level. PetaBricks automatically selects and constructs algorithms optimized for each target architecture using empirical autotuning methods. More background on the PetaBricks language and compiler is provided in Sections 2 and 4.1.

1.1 Contributions

We make the following contributions:

- We have introduced a novel programming language that incorporates algorithmic accuracy choices. This includes support for multiple accuracy metrics, which provide a general-purpose way for users to define arbitrary accuracy requirements in any domain.
- We have developed a technique for mapping variable accuracy code so that it can be efficiently autotuned without the search space growing prohibitively large.
- We have implemented a language, compiler, and runtime system to demonstrate the effectiveness of our techniques.
- We have implemented a suite of six benchmarks, representative of commonly used algorithms with variable accuracy requirements.
- We demonstrate the importance of algorithmic choice by showing that for different accuracy levels our autotuner chooses different algorithmic techniques.
- We show that by using variable accuracy one can get a speedup up to four orders of magnitude over using the highest accuracy code.

1.2 Outline

Section 2 provides a description of the PetaBricks language through an example program. Section 3 continues by describing the language extensions made to support variable accuracy. Section 4 provides background on the PetaBricks compiler infrastructure. Section 5 describes our accuracy-aware autotuner. Section 6 presents our suite of benchmarks and results. Finally, Sections 7 and 8 present related work and draw conclusions.

2. PetaBricks Language Background

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. 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 any common procedural language. The major difference with PetaBricks 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. Dependencies are specified by naming the inputs and outputs of each rule, but unlike in a traditionally dataflow programming model, more than one rule can be defined to output the same data. Thus, the input dependences of a rule can be satisfied by the output of one or more rules. It is up to the PetaBricks compiler and autotuner to decide which rules to use to satisfy such dependencies by determining which are most computationally efficient for a given architecture and input. For example, on architectures with multiple processors, the autotuner may find that it is preferable to use rules that minimize the critical path of the transform, while on sequential architectures, rules with the lowest computational complexity may fair better. The following example will help to further illustrate the PetaBricks language.

```
1  transform kmeans
2  from Points[n,2] // Array of points (each column
3                   // stores x and y coordinates)
4  through Centroids[sqrt(n),2]
5  to Assignments[n]
6  {
7    // Rule 1:
8    // One possible initial condition: Random
9    // set of points
10   to(Centroids.column(i) c) from(Points p) {
11     c=p.column(rand(0,n))
12   }
13
14   // Rule 2:
15   // Another initial condition: Centerplus initial
16   // centers (kmeans++)
17   to(Centroids c) from(Points p) {
18     CenterPlus(c, p);
19   }
20
21   // Rule 3:
22   // The kmeans iterative algorithm
23   to(Assignments a) from(Points p, Centroids c) {
24     while (true) {
25       int change;
26       AssignClusters(a, change, p, c, a);
27       if (change==0) return; // Reached fixed point
28       NewClusterLocations(c, p, a);
29     }
30   }
31 }
```

Figure 1. Pseudocode for kmeans

2.1 Example Program

Figure 1 presents an example PetaBricks program, kmeans. This kmeans program groups the input `Points` into a number of clusters and writes each points cluster to the output `Assignments`. Internally the program uses the intermediate

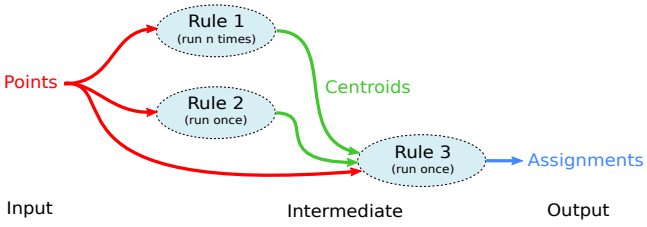


Figure 2. Dependency graph for kmeans example. The rules are the vertices while each edge represents the dependencies of each rule. Each edge color corresponds to each named data dependence in the pseudocode.

data `Centroids` to keep track of the current center of each cluster. The transform header declares each of these data structures as its inputs (`Points`), outputs (`Assignments`), and intermediate or "through" data structures (`Centroids`). The rules contained in the body of the transform define the various pathways to construct the `Assignments` data from the initial `Points` data. The transform can be depicted using the dependence graph shown in Figure 2, which indicates the dependencies of each of the three rules.

The first two rules specify two different ways to initialize the Centroids data needed by the iterative kmeans solver in the third rule. Both of these rules require the Points input data. The third rule specifies how to produce the output Assignments using both the input Points and intermediate Centroids. Note that since the third rule depends on the output of either the first or second rule, the third rule cannot be executed until the intermediate data structure Centroids has been computed by one of the first two rules. The autotuner and compiler will make sure the program will satisfy these dependencies when producing tuned code.

Additionally, the first rule provides an example of how the autotuner can synthesize outer control flow. Instead of using a rule that explicitly loops over every column of Centroids 2D array, the programmer has specified a computation that is done for each column of the output. The order over which these columns are iterated is then synthesized and tuned by the compiler and autotuner. Columns may also be processed in parallel if dependencies are satisfied. Operations can be specified on a per-row or per-cell basis as well, allowing optimizations such as cache-blocking to be automatically discovered.

To summarize, when our transform is executed, the cluster centroids are initialized either by the first rule, which performs random initialization on a per-column basis with synthesized outer control flow, or the second rule, which calls the CenterPlus algorithm. Once Centroids is generated, the iterative step in the third rule is called.

3. PetaBricks Language Extensions for Variable Accuracy

In this section, we introduce our language extensions to PetaBricks that add our variable accuracy support. We start by presenting our kmeans example again, this time augmented to support variable accuracy invocations.

At a high level, our language extensions extend the idea of algorithmic choice to include choices between different accuracies. The extensions also allow the user to specify how accuracy should be measured. Our new accuracy-aware autotuner then searches to optimize for both time and accuracy. The result is code that probabilistically meets users' accuracy needs. Optionally, users can request hard guarantees at runtime by checking output quality, and if needed, re-executing a transform.

3.1 Example with Variable Accuracy

Figure 3 presents our kmeans example with our new variable accuracy extensions. The updates to the code are highlighted in light blue. The example uses three of our new variable accuracy features.

First the `accuracy_metric`, on line 2, defines an additional transform, kmeansaccuracy, which computes the accuracy of a given input/output pair to kmeans. PetaBricks uses this transform during autotuning and sometimes at runtime to test the accuracy of a given configuration of the kmeans transform. The accuracy metric transform computes the $\sqrt{\frac{2n}{\sum D_i^2}}$, where D_i is the Euclidean distance between the i-th data point and its cluster center. This metric penalizes clusters that are sparse and is therefore useful for determining the quality of the computed clusters. Accuracy metric transforms such as this one might typically be written anyway for correctness or quality testing, even when programming without variable accuracy in mind.

The `accuracy_variable` k, on line 3 controls the number of clusters the algorithm generates by changing the size of the array `Centroids`. The variable k can take different values for different input sizes and different accuracy levels. The compiler will automatically find an assignment of this variable during training that meets each required accuracy level.

The `for_enough` loop on line 26 is a loop where the compiler can pick the number of iterations needed for each accuracy level and input size. During training the compiler will explore different assignments of k, algorithmic choices of how to initialize the `Centroids`, and iteration counts for the `for_enough` loop to try to find optimal algorithms for each required accuracy.

The next section goes on to explain each new variable accuracy feature in more detail.

3.2 Variable Accuracy Extensions

In order to support variable accuracy we made the following extensions to PetaBricks:

```
transform kmeans
accuracy_metric kmeansaccuracy
accuracy_variable k
from Points[n,2] // Array of points (each column
                 // stores x and y coordinates)
through Centroids[k,2]
to Assignments[n]
{
  // Rule 1:
  // One possible initial condition: Random
  // set of points
  to(Centroids.column(i) c) from(Points p) {
    c=p.column(rand(0,n))
  }

  // Rule 2:
  // Another initial condition: Centerplus initial
  // centers (kmeans++)
  to(Centroids c) from(Points p) {
    CenterPlus(c, p);
  }

  // Rule 3:
  // The kmeans iterative algorithm
  to(Assignments a) from(Points p, Centroids c) {
    for_enough {
      int change;
      AssignClusters(a, change, p, c, a);
      if (change==0) return; // Reached fixed point
      NewClusterLocations(c, p, a);
    }
  }
}

transform kmeansaccuracy
from Assignments[n], Points[n,2]
to Accuracy
{
  Accuracy from(Assignments a, Points p){
    return sqrt(2*n/SumClusterDistanceSquared(a,p));
  }
}
```

Figure 3. Pseudocode for variable accuracy kmeans. The new variable accuracy code is highlighted in light blue.

- The `accuracy_metric` keyword in the `transform header` allows the programmer to specify the name of another user-defined transform to compute accuracy from an input/output pair. This allows the compiler to test the accuracy of different candidate algorithms during training. It also allows the user to specify a domain specific accuracy metric of interest to them.
- The `accuracy_variable` keyword in the `transform header` allows the user to define one or more algorithm-specific parameters that influence the accuracy of the program. These variables are set automatically during training and are assigned different values for different input sizes. The compiler explores different values of these variables to create candidate algorithms that meet accuracy requirements while minimizing execution time.
- The `accuracy_bins` keyword in the `transform header` allows the user to define the range of accuracies that should be trained for and special accuracy values of interest that should receive additional training. The compiler can add such values of interest automatically based on how a transform is used. If not specified, the default range of accuracies is 0 to 1.0.
- The `for_enough` statement defines a loop with a compiler-set number of iterations. This is useful for defining iterative algorithms. This is syntactic sugar for adding an `accuracy_variable` to specify the number of iterations of a traditional loop.
- The `scaled_by` keyword on data inputs and outputs allows the user to indicate that data may be down-sampled or up-sampled using a user provided transform (or one of a number of built-in transforms). This is useful for algorithms operating on discrete samples of continuous data, such as signal/image processing. This is syntactic sugar for adding a wrapper-`transform` that has algorithmic choices for scaling with each allowed re-sampler or not re-sampling at all. The size to re-sample to is controlled with an `accuracy_variable` in the generated transform.
- The semantics for calling variable accuracy transforms is also extended. When a variable accuracy transform calls another variable accuracy transform (including recursively), the required sub-accuracy level is determined automatically by the compiler. This is handled by expanding each sub-call into an `either ... or` statement which allows the compiler to call the variable accuracy transform with different sub-accuracies.

  When a variable accuracy transform is called from fixed accuracy code, the desired level of accuracy must be specified. We use template-like, "`<N>`", syntax for specifying desired accuracy. This syntax may also be optionally used in variable accuracy transforms to prevent the automatic expansion described above.
- The keyword `verify_accuracy` in the rule body directs the compiler to insert a run time check for the level of accuracy attained. If this check fails the algorithm can be retried with the next higher level of accuracy or the user can provide custom code to handle this case. This keyword can be used when strict accuracy guarantees, rather than probabilistic guarantees, are desired for all program inputs. See Section 3.3 for a discussion of accuracy guarantees.

3.3 Accuracy Guarantees

PetaBricks supports the following types of accuracy guarantees:

- **Statistical guarantees** are the most common technique used, and the default behavior of our system. They work by performing off-line testing of accuracy using a set

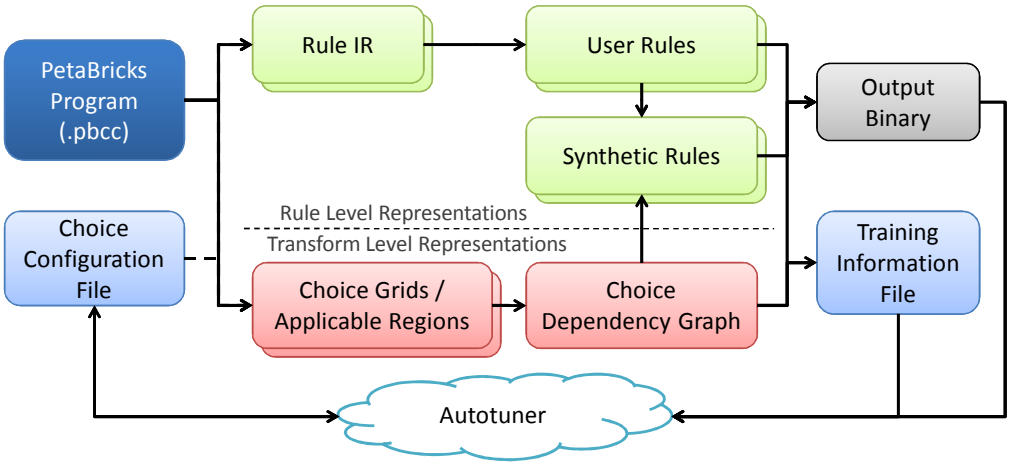


Figure 4. Flow for the compilation of a PetaBricks program with a *single* transform. (Additional transforms would cause the center part of the diagram to be duplicated.)

of program inputs to determine statistical bounds on an accuracy metric to within a desired level of confidence.

- **Run-time checking** can provide a hard guarantee of accuracy by testing accuracy at runtime and performing additional work if accuracy requirements are not met. Run-time checking can be inserted using the `verify_accuracy` keyword. This technique is most useful when the accuracy of an algorithm can be tested with low cost and may be more desirable in case where statistical guarantees are not sufficient.
- **Domain specific guarantees** are available for many types of algorithms. In these cases, a programmer may have additional knowledge, such as a lower bound accuracy proof or a proof that the accuracy of an algorithm is independent of data, that can reduce or eliminate the cost of runtime checking without sacrificing strong guarantees on accuracy.

As with variable accuracy code written without language support, in Petabricks, deciding which techniques to use to guarantee accuracy and what accuracy metrics to use is a decision left to the programmer.

4. Compiler

In this section, we describe the changes we made to the PetaBricks compiler to support our new language extensions. We start by providing a brief background of the original compiler.

4.1 PetaBricks Compiler Infrastructure Background

Figure 4 displays the general flow for the compilation of a PetaBricks transform. Compilation is split into two representations. The first representation operates at the rule level, and is similar to a traditional high level sequential intermediate representation. The second representation operates at the transform level, and is responsible for managing choices and for code synthesis.

The main transform level representation is the *choice dependency graph*, which is the primary way that choices are represented in PetaBricks. At a high level, the information contained in the choice dependency graph is similar to the dependency graph shown for our example program in Figure 2, however, the data is represented as an "inverse" of that graph: data dependencies (previously represented by edges) are represented by vertices, while rules (previously represented by vertices) are represented by graph hyperedges. Additionally, data may be split into multiple vertices in the choice dependency graph if the transform contains rules that operate on just subregions of that data. The PetaBricks compiler uses this graph to manage code choices and to synthesize the outer control flow of the rules.

The final phase of compilation generates an *output binary* and a *training information file* containing static analysis information. These two outputs are used by the autotuner (described in Section 5), to search the space of possible algorithmic choices. Autotuning creates a *choice configuration file*, which can either be used by the output binary to run directly or can be fed back into the compiler to allow additional optimizations.

For more details on the PetaBricks compiler infrastructure see our prior work [3].

4.2 Representing Variable Accuracy

Representing variable accuracy algorithms presents a key challenge both while compiling and while autotuning. The main difficulty is that variable accuracy adds a new dimension to how one can evaluate candidate algorithms. With fixed accuracy algorithms, the metric of performance can be used to order algorithms. With variable accuracy, we plot candidates on an accuracy/time grid. This naturally leads to an *optimal frontier* of algorithms for which no other al-

```
1 population = [ ... ]
2 mutators = [ ... ]
3 for inputsize in [1, 2, 4, 8, 16, ..., N]:
4   testPopulation(population, inputsize)
5   for round in [1, 2, 3, ..., R]:
6     randomMutation(population, mutators, inputsize)
7     if accuracyTargetsNotReached(population):
8       guidedMutation(population, mutators, inputsize)
9     prune(population)
```

Figure 5. Pseudocode showing the high level flow of our autotuning algorithm. The individual phases are described in Sections 5.5.1 to 5.5.4.

gorithm can provide a greater accuracy in less time. It is not possible to evaluate the entire optimal frontier, however, since it can potentially be of infinite size. Instead, to make this problem tractable, we discretize the space of accuracies by placing each allowable accuracy into a bin. The discretization can be specified by the user or can be automatically inferred by the compiler based on how a variable accuracy algorithm is used. For example, if an algorithm is called with a specific accuracy, that specific accuracy can be added as extra bin boundary by the compiler.

In the compiler, we represent these bins by extending and using the representation for templates. A variable accuracy algorithm is called with the syntax "`Foo<accuracy>`" and, similar to templates, each requested accuracy is considered by the compiler as a separate type. When variable accuracy code calls other variable accuracy code, the sub-accuracy is automatically determined by the compiler. This is done by representing the sub-accuracy as an algorithmic choice to call one of any of the accuracy bins. If a user wishes to call a transform with an unknown accuracy level, we support dynamically looking up the correct bin that will obtain a requested accuracy.

5. Variable Accuracy PetaBricks Autotuner

In this section, we describe our autotuning approach for constructing tuned variable accuracy programs. We will start with a overview of our algorithm, then provide background on some structures we use to represent different candidate algorithms (configurations, training info files, and mutators), and finally provide details on each of the phases of the autotuning process.

The autotuner we present here has been rewritten both to support variable accuracy and to explore the search space of algorithms more efficiently. The autotuner in our prior work [3] did not contain any support for tuning variable accuracy programs because it could only optimize a single metric (execution time). Our new autotuner is now able to optimize two metrics. Since this increases the search space dramatically, we had to introduce new guided search and pruning techniques to help search the space. Each of the following sections describe new features added to the autotuner that enable this change.

5.1 Overview

Figure 5 presents high level pseudocode for our autotuning algorithm. The autotuner follows a genetic algorithm approach to search through the available choice and accuracy space. It maintains a population of candidate algorithms which it continually expands using a set of mutators (described in Section 5.4) and prunes in order to allow the population to evolve more optimal algorithms. The input sizes used for testing during this process grow exponentially, which naturally exploits any optimal substructure inherent to most programs.

5.2 Choice Configuration Files

The PetaBricks compiler and autotuner represents different possible candidate algorithms through configuration files representing an assignment of decisions to all available choices. Broadly, one can divide the choices contained in the configuration file into the following categories.

- **Decision trees** to decide which algorithm to use for each choice site, accuracy, and input size.
- **Cutoffs values**. For example, switching points from a parallel work stealing scheduler to sequential code or the blocking sizes for data parallel operations.
- **Switches**. For example, the type of storage for intermediate data.
- **Accuracy Variables**. For example, how many iterations in a `for_enough` loop.
- **User defined parameters**.

5.3 Training Information File

The training information file (formatted in XML) contains static analysis information extracted from each PetaBricks program. It is primarily used by the autotuner to construct the pool of mutators (Section 5.4). It assists in this process by containing high level descriptions of all the logical constructs in the configuration file. It also contains dependency information in the form of call graphs, encodes accuracy

requirements for variable accuracy algorithms, and stores other meta-information.

5.4 Mutator Functions

Abstractly, a mutator function creates a new algorithm configuration by changing an existing configuration, its signature is:

$$\text{Configuration} \times N \rightarrow \text{Configuration}$$

where N is the current input size used in training, and each Configuration is a algorithm configuration file described in Section 5.2.

The set of mutator functions is different for each program, and is generated fully automatically with the static analysis information contained in the training information file.

The created mutators can be divided into four categories:

- **Decision tree manipulation mutators** either add, remove, or change levels to a specific decision tree represented in the configuration file. When adding new levels, the cutoff point is initially set to $\frac{3N}{4}$. This leaves the behavior for smaller inputs the same, while changing the behavior for the current set of inputs being tested. This cutoff point can later be changed, as a *log-normal random scaling mutator* is introduced for each active cutoff value in the decision tree.

  Initially decision trees are very simple, set to use just a single algorithm. At this point, the only mutators that can be applied simply change this single algorithm or add a level to the decision tree. As the decision tree becomes more complex, more and more mutators can legally be applied and the search space grows.

- **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. For example, blocking sizes, cutoffs in decision trees, and cutoffs to switch between sequential and parallel code.

  The intuition for why a log-normal distribution is used comes from the observation that small changes have larger effects on small values than large values. For example, most readers would expect changing a blocking size from 5 to 6 to have a much larger impact that changing a blocking size from 105 to 106. This observation is a general trend that applies to most configuration values dealing with sizes. We have confirmed this intuition experimentally by observing much faster convergence times with this type of scaling.

- **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. 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.

- **Meta mutators** multiply or reverse the effects of other mutators. The two types of mutators that fall into this category either randomly apply a number of other mutators (allowing larger jumps to be taken in the configuration space) or undo the effects of a previously applied mutator.

The log-normal random scaling and uniform random mutators both affect program accuracy directly when they are applied to accuracy variables. The decision tree manipulation mutators affect accuracy by making algorithmic changes that can result in different accuracies. Meta mutators affect accuracy only indirectly as the call or undo the other types of mutators. That autotuner conservatively assumes all mutators affect accuracy when training and thus accuracy is retested after each mutation.

Mutators also perform two other tasks as an optimization. First, in cases where the behavior of the algorithm is unchanged either below or above a threshold (for example, when a new level is added to a decision tree, with the bottom of the tree unchanged) the mutator copies unaffected results gathered on the input candidate algorithm to the output candidate algorithm. This reduces the need for future testing. Secondly, mutators can enable or disable other mutators for a given candidate. For example, when levels are added to a decision tree, the mutators to manipulate this higher level are enabled by the mutator creating the level.

5.5 Autotuning Phases

5.5.1 Population Testing

The dominant time requirement of our autotuner is testing candidate algorithms by running them on training inputs. This testing measures both the time required and the resulting accuracy of each candidate algorithm. Accuracy is measured by running the accuracy metric defined by the user after each testing run. We represent both time and accuracy by using least squares to fit a normal distribution to the observed data. This fitting allows us to give statistical bounds (for example with a 95% confidence) for accuracy. When the programmer elects to use statistical accuracy guarantees, this alone is sufficient to guarantee accuracy. When the programmer elects to have runtime verification of accuracy, this runtime verification is disabled during autotuning to allow exploration of the choice space. When the programmer has provided fixed (hand proven) accuracies the accuracy metrics will return a constant value for each candidate algorithm and the normal distributions will become singular points.

An important decision that must be made is how many times to test each candidate algorithm. With too few tests, random deviations may cause non-optimal decisions to be made, while with too many tests, autotuning will take an unacceptably long time.

A simple solution would be to use some fixed number of tests for all candidate algorithms, however this is non-optimal. The number of tests needed depends both on what one is comparing an algorithm against (larger differences can be verified with fewer tests than smaller differences) and on the variance of results.

There is a configurable upper and lower limit for the number of tests to run. Reasonable values for this upper and lower limit are 3 and 25, however these values can be set by the user based on their needs. In the testing phase of the tuner (line 4 in Figure 5), we run the minimum number of tests for each candidate algorithm in the population. Then, during the subsequent phases of autotuning we dynamically run more tests as needed.

We use the following heuristic to decide when to run additional tests. When comparing two candidate algorithms, C_1 and C_2, we perform the following steps:

1. Use statistical hypothesis testing (a t-test [18]) to estimate the probability P(observed results $\mid C_1 = C_2$). If this results in a p-value less than 0.05, we consider C_1 and C_2 different and stop.[1]
2. Use least squares to fit a normal distribution to the percentage difference in the mean performance or accuracy of the two algorithms. If this distribution estimates there is a 95% probability of less than a 1% difference, consider the two algorithms the same and stop.[1]
3. If both candidate algorithms have reached the maximum number of tests, consider the two algorithms the same and stop.
4. Run one additional test on either C_1 or C_2. Decide which candidate to test based on the highest expected reduction in standard error and availability of tests without exceeding the maximum.
5. Go to step 1.

This heuristic results in a good balance, where additional tests are run only where they are needed. It also gives our autotuner the ability to adapt to variance in the environment. As an interesting anecdotal experiment, if one begins to rapidly move the mouse during autotuning, the increased variance from this I/O activity causes the average number of trials to approximately triple for our Sort benchmark (from approximately 5 to 15 for a specific run and configuration).

5.5.2 Random Mutation

The random mutation phase of autotuning (line 6 in Figure 5), attempts to add new members to the population using the mutators described in Section 5.4. Repeatedly (for a configurable number of *attempts*), it picks a random candidate algorithm (the *parent*) from the population and a random mutator from the mutator pool and uses that mutator to create a new candidate algorithm (the *child*). Parent algorithms remain in the population, and are only removed by the pruning phase.

Upon creation, a child algorithm is first tested the minimum number of times. It is then compared against its parent using the process described in Section 5.5.1. This process may result in both the parent and the child being tested additional times. If the child is better than the parent either in time or in accuracy it is added to the population.

Child algorithms may provide different accuracies than the parent when the mutator applied makes changes that affect accuracy. This difference in accuracy is accounted for during pruning as the autotuner attempts to maintain a variety of algorithms that are optimal for each level of accuracy.

5.5.3 Guided Mutation

Infrequently, the random mutation process may not produce any candidate algorithms that meet the accuracy requirements given by the user. This most commonly happens with the initial population, after only 1 round of random mutation. In this case we use a guided mutation process process to attempt to create a candidate algorithm that meets the user's requirements.

This guided mutation process is possible because the training information file contains hints as to which configuration values affect accuracy. These *accuracy variables* are things such as the iteration counts in `for_enough`s loop. The guided mutation simply does hill climbing on this accuracy variables. If the required accuracy cannot be attained (the guided mutation process fails), an error is reported to the user.

5.5.4 Population Pruning

The pruning phase of autotuning (line 9 in Figure 5), removes unfit candidates from the population. For each accuracy bin required by the user, the pruning keeps the fastest K algorithms that meet the accuracy requirement, where K is a configurable parameter of the autotuner. Any algorithms not included in these sets are removed from the population.

When considering the pruning process, one could imagine an optimal frontier of algorithms. This optimal frontier can be thought of as a curve on a 2D plane where one axis is time and the other is accuracy. For each possible accuracy, there exists a single algorithm that provides at least that accuracy in the lowest possible time. Conversely, for each maximum execution time, there exists a single algorithm that provides the highest accuracy possible in that time. Collectively these algorithms make up a, possibly infinite in size, optimal frontier of algorithms that one may want to use. The multiple accuracy bins in our autotuner store a discretized version of this optimal frontier of algorithms. The bins are selected such that the fastest algorithms for the accuracies actually used are maintained.

[1] The constants shown here are configurable, and can be changed based on the needs of the user. We present a typical value for clarity.

Since comparisons between candidates can result in additional trials (see Section 5.5.1), determining the fastest K algorithms in each accuracy bin can be expensive. To reduce the number of comparisons we use the following procedure:

1. Roughly sort the candidate list by mean performance without running any additional trials.
2. Split the list at the Kth element into a $KEEP$ list and a $DISCARD$ list.
3. Fully sort the $KEEP$ list, running additional trials as needed to gain the needed confidence.
4. Compare each element in the $DISCARD$ list to the Kth element in the $KEEP$ list (running additional trials as needed). If any of these elements are faster, move them to the $KEEP$ list.
5. Fully sort the $KEEP$ list again, running additional trials as needed to gain the needed confidence.
6. Return the first K elements of the $KEEP$ list.

This technique avoids fully sorting the elements of the population that will be discarded. This causes more testing time to be invested in the candidate that will be kept in the population. It also exploits the fact that comparing algorithms with larger differences in performance is cheaper than comparing algorithms with similar performance.

6. Experimental Results

In this section, we present experimental results for a suite of new variable accuracy PetaBricks benchmarks. We first describe each of the benchmarks in detail. Next, we present the performance improvements that can be attained by varying the desired accuracy of the benchmarks. Subsequently, we describe some interesting compositional outcomes generated by our autotuner. These results illustrate the complexity of the choice space for variable accuracy algorithms. We believe that these results help motivate the need for automatic search and for abstractions that isolate the user from these complex choices. Finally, we discuss our experiences with programming with and without our new language extensions.

6.1 Benchmarks

6.1.1 Bin Packing

Bin packing is a classic NP-hard problem where the goal of the algorithm is to find an assignment of items to unit sized bins such that the number of bins used is minimized, no bin is above capacity, and all items are assigned to a bin. It is an interesting problem because, while finding the optimal assignment is NP-hard, there are a number of polynomial time approximation algorithms that each provide different levels of approximation and performance.

The bin packing benchmark demonstrates the ability of our system to handle a large number of algorithmic choices. Variable accuracy is attained primarily through using different algorithms. We implemented the following algorithmic choices in PetaBricks:

- **FirstFit** – Iterate through the items, placing each in the first bin that has capacity. This will use no more than $17/10 \times OPT$ bins in the worst case where OPT is the number of bins used in an optimal packing.
- **FirstFitDecreasing** – Reverse-sort the items and call **FirstFit**. Sorting the items before applying **FirstFit** reduces the worst case bounds to $10/9 \times OPT$.
- **ModifiedFirstFitDecreasing** – A variant of **FirstFitDecreasing** that classifies items into categories to improve the provable accuracy bound to 71/60 from optimal [16].
- **BestFit** – Iterate through the items, placing each in the most-full bin that has capacity. This has the same worst case packing performance as **FirstFit**.
- **BestFitDecreasing** – Reverse-sort the items and call **BestFit**. This has the same worst case packing performance as **FirstFitDecreasing**.
- **LastFit** – Iterate through the items, placing each in the last nonempty bin that has capacity.
- **LastFitDecreasing** – Reverse-sort the items and call **LastFit**.
- **NextFit** – Iterate through the items, placing each in the last nonempty bin if possible, otherwise start a new bin. This has been shown to perform $2 \times OPT$ in the worst case.
- **NextFitDecreasing** – Reverse-sort the items and call **NextFit**.
- **WorstFit** – Iterate through the items, placing each in the least-full nonempty bin that has capacity.
- **WorstFitDecreasing** – Reverse-sort the items and call **WorstFit**.
- **AlmostWorstFit** – A variant of **WorstFit**, that instead puts items in the kth-least-full bin. **AlmostWorstFit** by definition sets $k = 2$, but our implementation generalizes it and supports a variable compiler-set k. This has the same worst case packing performance as **FirstFit**.
- **AlmostWorstFitDecreasing** – Reverse-sort the items and call **AlmostWorstFit**.

To train this benchmark, we generate training data by dividing up full bins into a number of items such that the resulting distribution of item sizes matches that of a distribution of interest to us. Using this method, we can construct an accuracy metric that measures the relative performance of an algorithm to the optimal packing at training time, without the need for an exponential search. In this way, we are able to efficiently autotune the benchmark for a particular distribution of item sizes with an effective accuracy metric.

6.1.2 Clustering

Clustering assigns a set of data into subgroups (clusters) of similar patterns. Clustering is a common technique for statistical data analysis in areas including machine learning, pattern recognition, image segmentation and computational biology. The k-means clustering algorithm is a common method of cluster analysis which partitions n data points into k clusters ($k \leq n$) in which each data point is assigned to the nearest cluster. The k-means problem is NP-hard when k is fixed. As a result, algorithms that seek fixed points, such as Lloyd's algorithm, are used in practice.

The first step of solving the k-means problem is to find the number of clusters k in the data set. There have been many studies on choosing k, but the best choice is often not obvious since it depends on the underlying distribution of data and desired accuracy. Once the number of clusters is determined, Lloyd's algorithm starts with k initial centers chosen randomly or by some heuristics. Each data point is assigned to its closest center, measured by some distance metric. The cluster centers are then updated to be the mean of all the points assigned to the corresponding clusters. The steps of partitioning points and recalculating cluster centers are repeated until no changes occur.

In our PetaBricks transform, the number of clusters, k, is the accuracy variable to be determined on training. Several algorithmic choices are implemented in our version of k-means clustering: The initial set of k cluster centers are either chosen randomly among the n data points, or according to the k-means++ algorithm [4], which chooses subsequent centers from the remaining data points with probability proportional to the distance squared to the closest center. Once the initial cluster centers are computed, the final cluster assignments and center positions are determined by iterating, either until a fixed point is reach or in some cases when the compiler decides to stop early.

The training data is a randomly generated clustered set of n points in two dimensions. First, $\sqrt{n}$ "center" points are uniformly generated from the region $[-250, 250] \times [-250, 250]$. The remaining $n - \sqrt{n}$ data points are distributed evenly to each of the $\sqrt{n}$ centers by adding a random number generated from a standard normal distribution to the corresponding center point. The optimal value of $k = \sqrt{n}$ is not known to the autotuner. The accuracy metric used is $\sqrt{\frac{2n}{\sum D_i^2}}$, where D_i is the Euclidean distance between the i-th data point and its cluster center. The reciprocal is chosen such that a smaller sum of distance squared will give a higher accuracy.

6.1.3 3D Variable-Coefficient Helmholtz Equation

The variable coefficient 3D Helmholtz equation is a partial differential equation that describes physical systems that vary through time and space. Examples of its use are in the modeling of vibration, combustion, wave propagation, and climate simulation. It can be expressed as:

$$\alpha(a\phi) - \beta \bigtriangledown \cdot (b \bigtriangledown \phi) = f, \tag{1}$$

where α and β are constants, a, b, and f are scalar valued functions over the area of interest, and ϕ is the unknown quantity we are solving for.

As an accuracy metric, we used the ratio between the RMS error of the initial guess fed into the algorithm and the RMS error of the guess afterwards. The values of a and b were taken from the uniform distribution between 0.5 and 1 to ensure the system is positive-definite.

This benchmark utilizes multiple resolution levels, where each recursive call works on a problem with half as many points in each dimension. Since this is a three dimensional problem, every time a recursive call is made, the amount of data decreases by a factor of eight, possibly changing key performance parameters such as iteration strategy. Additionally, there is a lot of state data that needs to be transformed (either averaged down or interpolated up) between levels of recursion due to the presence of the variable coefficient arrays a and b. This overhead of making recursive calls influences the decision of when it is optimal to transition from recursing further to smaller problem sizes or to stop and solve the problem as best we can on the current problem size using a direct or iterative method.

6.1.4 Image Compression

In image compression, we can express an m-by-n image by an m-by-n matrix, where the (i, j) entry represents the brightness of the pixel at that point. Instead of storing all mn entries, we can compress the image by storing less data, which can later be used to reconstruct an image similar to the original to within a desirable accuracy.

One way to perform this approximation is by Singular Value Decomposition (SVD). For any $m \times n$ real matrix A with $m \geq n$, the SVD of A is $A = U\Sigma V^T$, where the columns u_i of U are called the left singular vectors, the columns v_i of V are called the right singular vectors, and the diagonal values σ_i of Σ are called the singular values. The best rank-k approximation of A is given by $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$ [10]. Only the first k columns of U and V and the first k singular values σ_i need to be stored to reconstruct the image approximately.

In this paper, we consider $n \times n$ input matrices, generated from a uniform distribution on (0, 1). Matrix entries taking values from 0 to 1 can represent pixels in gray-scale, ranging from black (0) to white (1). The SVD of a square matrix A can be computed using the eigenvalues and eigenvectors of the matrix $H = [0\ A^T; A\ 0]$. The number of singular values, k, to be used in the approximation is the accuracy variable to be determined by the PetaBricks autotuner. Our PetaBricks transform for matrix approximation consists of a hybrid algorithm for finding all eigenvalues and eigenvectors, which combines Divide and Conquer, QR Iteration and Bisection

method and is implemented using LAPACK routines. Another algorithmic choice in our PetaBricks transform is Bisection method for only k eigenvalues and eigenvectors. The accuracy metric used is the ratio between the RMS error of the initial guess (the zero matrix) to the RMS error of the output compared with the input matrix A, converted to log-scale.

6.1.5 2D Poisson's Equation

The 2D Poisson's equation is an elliptic partial differential equation that describes heat transfer, electrostatics, fluid dynamics, and various other engineering disciplines. The continuous and discrete versions are

$$\bigtriangledown^2\phi = f \quad \text{and} \quad Tx = b, \tag{2}$$

where T, x, and b are the finite difference discretization of the Laplace operator, ϕ, and f, respectively.

To build an autotuned multigrid solver for Poisson's equation, we consider the use of three basic algorithmic building blocks: one direct (band Cholesky factorization through LAPACK's DPBSV routine), one iterative (Red-Black Successive Over Relaxation), and one recursive (multigrid).

As an accuracy metric, we used the ratio between the RMS error of the initial guess fed into the algorithm and the RMS error of the guess afterwards. The right hand side vector b was taken to be uniform over the interval $[-2^{31}, 2^{31})$.

Of these three algorithmic choices, only two of them are variable accuracy, while the third (direct) solves the problem to machine-precision (assuming reasonably behaved inputs). This variable accuracy algorithm is one of the more complex of our benchmarks due to its ability to tune the number of iterations at each level of recursion thus making varying accuracy guarantees during each stage of execution.

For example, we may want to iterate many times at a lower recursion level to obtain a high-fidelity estimate before interpolating up to the initial problem size to save on expensive operations at the highest grid resolution. On the other hand, if our final accuracy requirements are high enough, it may not pay to do many iterations at a lower recursion level if we are going to have to do many expensive full resolution iterations either way. It is this kind of trade-offs that our variable accuracy auto-tuner excels at exploring.

6.1.6 Preconditioned Iterative Solvers

Solving a linear system of equations $Ax = b$ is a common problem in both scientific research and real-world applications such as cost optimization and asset pricing. Iterative methods are often used to provide approximate solutions as direct solvers are usually too slow to produce exact solutions. Preconditioning is a technique that speeds up the convergence of an iterative solver.

The convergence of a matrix iteration depends on the properties of the matrix A, one of which is called the condition number. A preconditioner P of a matrix A is a matrix that if well chosen, the condition number of $P^{-1}A$ is smaller than that of A. Although the preconditioned system $P^{-1}Ax = P^{-1}b$ has the same solution as the original system, the rate of convergence depends on the condition number of $P^{-1}A$. The preconditioner $P = A$ has the optimal condition number, but evaluating $P^{-1}b = A^{-1}b$ is equivalent to solving the original system. Achieving a faster convergence rate while keeping the operation of P^{-1} simple to compute is the key to finding a good preconditioner.

Our preconditioner PetaBricks transform implements three choices of preconditioners and solves the system. The first choice is the Jacobi preconditioner coupled with Preconditoned Conjugate Gradient (PCG). The preconditioner is chosen to be the diagonal of the matrix $P = \text{diag}(A)$. Another choice is to apply the polynomial preconditioner $P^{-1} = p(A)$, where $p(A)$ is an approximation of the inverse of A by using a few terms of the series expansion of A^{-1}, and solve the preconditioned system with PCG. We also implemented the Conjugate Gradient method (CG) which solves the system without any preconditioning.

For training data, we set A to be the discretized operator of the Poisson Equation, and use randomly generated entries for b. The accuracy metric is the ratio between the RMS error of the initial guess Ax_{in} to the RMS error of the output Ax_{out} compared to the right hand side vector b, converted to log-scale.

6.2 Experimental Setup

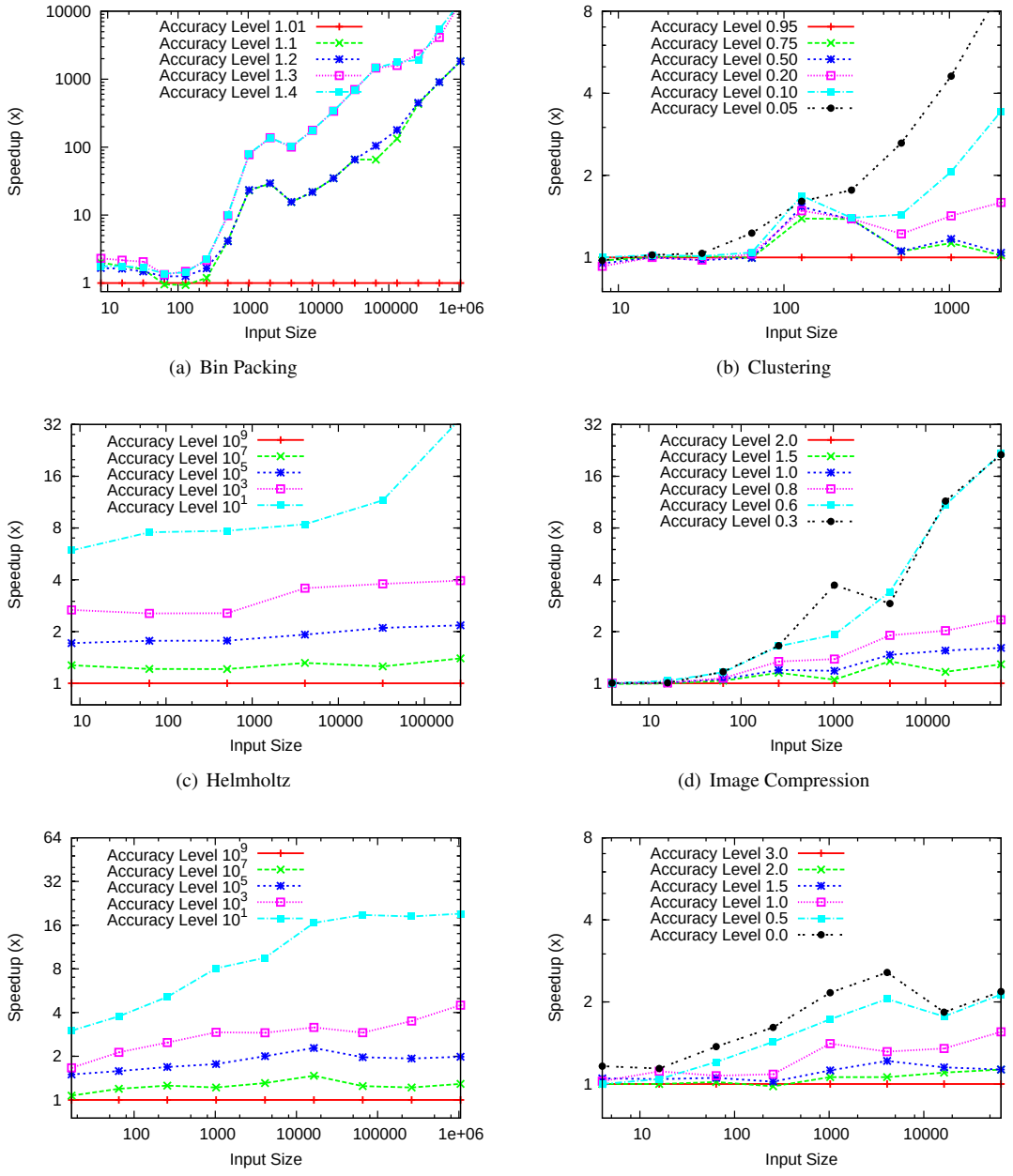
We performed all tests on an 8-core (dual-Xeon X5460) system clocked at 3.16 GHz with 8 GB of RAM. The system was running Debian GNU/Linux 5.0.3 with kernel version 2.6.26. All benchmarks are automatically parallelized by the PetaBricks compiler and were run and trained using 8 threads.

6.3 Speedups Compared to Highest Accuracy Level

Figures 6(a)-6(f) show the speedups that are attainable when a user is in a position to use an accuracy lower than the maximum accuracies of our benchmarks. On the largest tested input size, for benchmarks such as Clustering and Preconditioner speedups range from 1.1 to 9.6x; for benchmarks such as Helmholtz, Image Compression, and Poisson speedups range from 1.3 to 34.6x; and for the Bin Packing benchmark speedups ranged from 1832 to 13789x. Such dramatic speedups are a result of algorithmic changes made by our autotuner that can change the asymptotic performance of the algorithm ($O(n)$ vs $O(n^2)$) when allowed by a change in desired accuracy level. Because of this, speedup can become a function of input size and will grow arbitrarily high for larger and larger inputs. These speedups demonstrate some of the performance improvement potentials available to programmers using our system.

6.4 Impact of Accuracy on Algorithmic Choices

Bin Packing Figure 7 depicts the results of autotuning the Bin Packing benchmark for various desired accuracy levels



(a) Bin Packing

(b) Clustering

(c) Helmholtz

(d) Image Compression

(e) Poisson

(f) Preconditioner

Figure 6. Speedups for each accuracy level and input size, compared to the highest accuracy level for each benchmark. Run on an 8-way (2×4-core Xeon X5460) system.

(average number of bins used over the optimal). For any desired accuracy between 1 and 1.5, the figure indicates the approximation algorithm that performs fastest on average, for input data sizes between 8 and 2^{20} generated by our training data generator. The results show that each of the 13 approximation algorithms used by the benchmark perform fastest for some areas of the accuracy/data size space. This presents a major challenge to developers seeking high performance when using today's programming languages since there exists no clear winner among the algorithms. Instead, the best choice will depend on the desired accuracy and input size. Thus, when writing a Bin Packing library, today's high performance programmers have the option of either producing a brittle special-casing of the algorithmic choices manually (which would be very tedious given the number of well performing choices), or break the algorithm's abstraction to let the user specify which choice to go with. Either of the two options are undesirable.

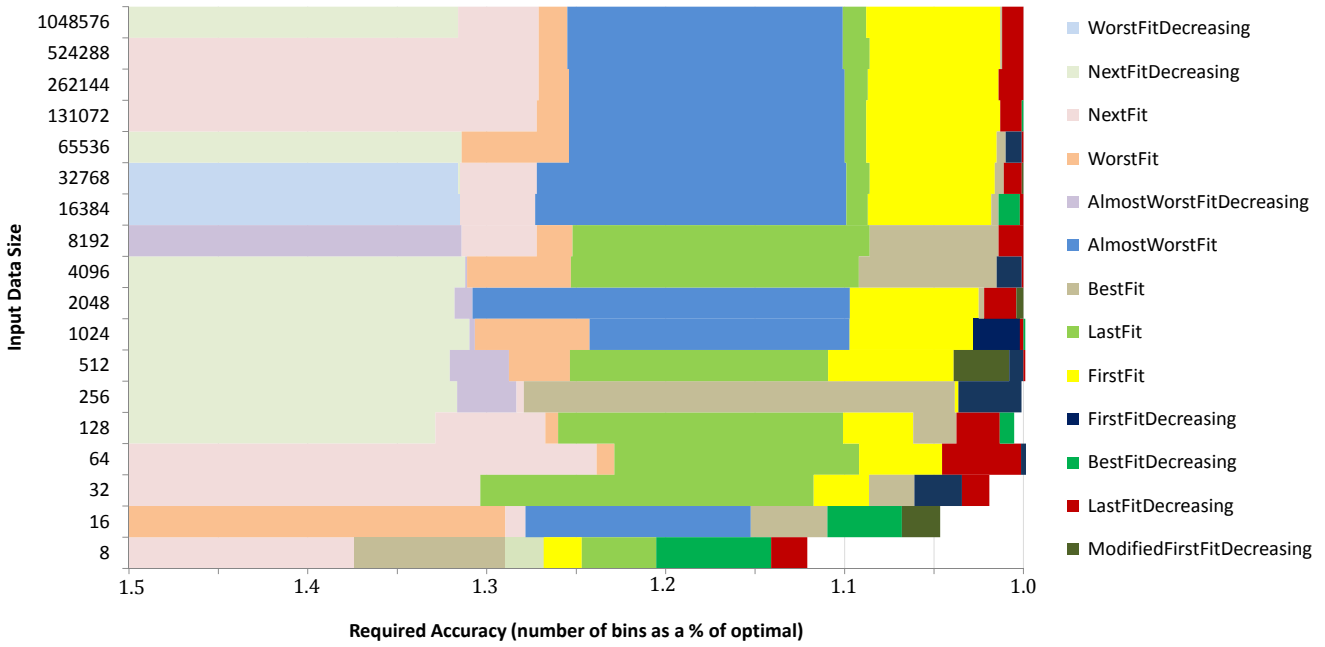


Figure 7. Best algorithm for each accuracy and input size in the Bin Packing benchmark. By best we mean on the optimal frontier (there exists no algorithm with better performance and accuracy for a given input size on average). Accuracy is defined as the number of bins over the optimal number of bins achievable. Lower numbers represents a higher accuracy.

Accuracy	k	Initial Center	Iteration Algorithm
0.10	4	random	once
0.20	38	k-means++	25% stabilize
0.50	43	k-means++	once
0.75	45	k-means++	once
0.95	46	k-means++	100% stabilize

Table 1. Algorithm selection and initial k value results for autotuned k-means benchmark for various accuracy levels with n=2048 and k_optimal = 45

It is also interesting to note the relatively poor performance of `ModifiedFirstFitDecreasing`, despite the fact that it has the best provable accuracy bounds out of the set algorithms. It is best in only three small areas in the accuracy/data size space. Additionally, despite the fact that it is provably guaranteed to be within $71/60$ ($1.18\times$) of optimal, it is never the best performing algorithm when a probabilistic bound of worse than $1.07\times$ accuracy is desired. This result highlights the advantages of using a empirical approach to determining optimal algorithms when probabilistic guarantees on accuracy are permissible.

Clustering Table 1 illustrates the results of autotuning our k-means benchmark on our sample input of size $n = 2048$. The results show interesting algorithmic choices and number of clusters k chosen by the autotuner. For example, at accuracies greater than 0.2, the autotuned algorithm correctly uses the accuracy metric (based on Euclidean distances between data points and cluster centers) to construct an algorithm that picks a k value that is close to 45, which is the number of clusters generated by our training data (which is not known to the autotuner).

At accuracy 0.1, the autotuner determines 4 to be the best choice of k and chooses to start with a random cluster assignment with only one level of iteration. While this is a very rough estimate of k and a very rough cluster assignment policy, it is sufficient to achieve the desired low level of accuracy. To achieve accuracy 0.2, the autotuner uses 38 clusters, which is slightly less than the predetermined value. Our autotuned algorithm determines the initial cluster centers by k-means++, and iterates until no more than 25% of the cluster assignments change. For accuracy 0.5 and 0.75, the ks picked by the autotuner algorithm are 43 and 45 respectively, which are only slightly smaller or equal to the predetermined k. The initial centers are decided by k-means++ and only one iteration is used. By successfully finding a number of clusters that is close to the predetermined k and picking good initial centers, only one iteration is needed on average during training to achieve a high level of accuracy. Finally, to achieve the highest accuracy of 0.95, the algorithm uses k value of 46. Initial centers are determined by k-means++ and iterations are performed until a fixed point is reached. It is interesting to note that on average, the autotuner finds that a value of k that is one higher than the k used to generate the data, is best to minimize the user specified accuracy metric,

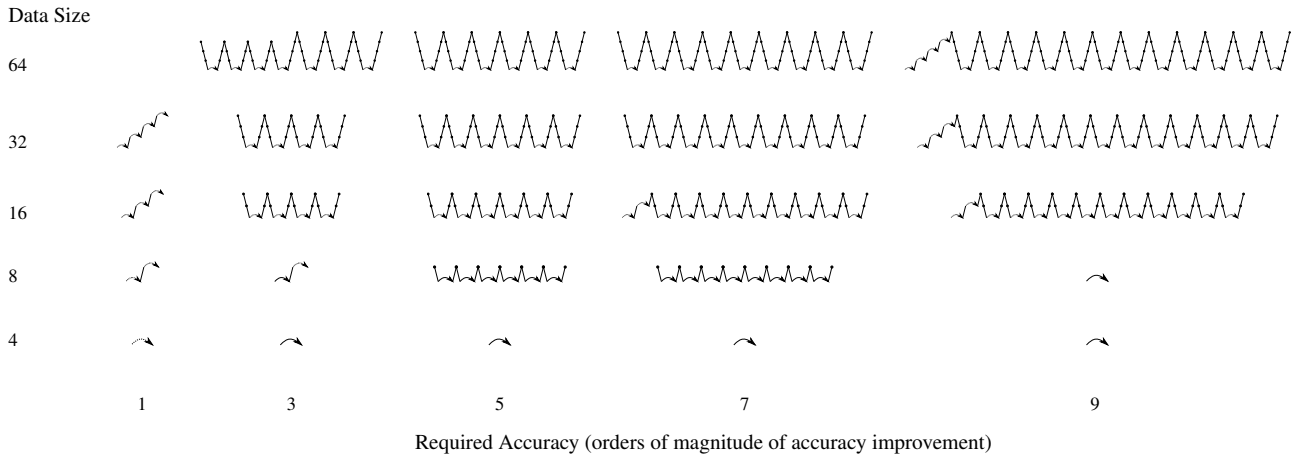


Figure 8. Resulting cycle shapes for Helmholtz after tuning for different input data sizes and required accuracies. The solid arrows at the bottom of the cycles represent shortcut calls to the direct solver, while the dashed arrows represents an iterative solve using SOR. The dots present in the cycle represent single relaxations. Note that some paths in the full multigrid cycles skip relaxations while moving to a higher grid resolution. The notation of these diagrams is similar to that of [10].

3D Variable-Coefficient Helmholtz Equation Figure 8 presents the multigrid cycle shapes chosen by our autotuner for the Helmholtz algorithm. The shapes depict execution traces of the algorithm for varying accuracy levels and input sizes. The results show that compiler is able to do a good job searching the space of possible cycle shapes despite having to make difficult time-accuracy trade-offs at every stage of the recursion.

The asymmetry in some of the figures are due to a property of the algorithm that allows for an *estimation phase*, during which work is done to converge towards the solution at smaller problem sizes before work is expended at the largest problem size. This is done to provide the Helmholtz solver with an initial guess closer to the optimal solution, which in some cases can pay for itself in saved iterations at the most expensive level.

We found that in most cases with a large enough input, the accuracy levels used for recursive calls could be well below the desired accuracy of the final output given enough repetitions of the recursive call. Further, the depth of the V-cycles can be truncated by substituting the traditional deep V-cycle shape with a shallow V-cycle with an iterative bottom solve. This performance enhancing substitution is made possible by the autotuner's awareness of the different algorithmic choices available to achieve each desired accuracy level.

Additionally, it is interesting to notice the effect of the desired accuracy on the shape of the cycles. For the lowest accuracy requirement, we find that the result of the estimate phase (the non-symmetric, leftmost part of the cycle shape) is sufficiently accurate to meet the accuracy requirement. When solving for 3 orders of magnitude of accuracy improvement, the algorithm begins to rely more on the solve phase. At 5 and 7 orders of magnitude of accuracy improvement, the algorithm decides not to perform any estimation at all the majority of the time. This result stands in contrast to the results at 9 orders of magnitude of accuracy improvement, where for data sizes greater than 64, the algorithm performs extensive estimation through multiple SOR relaxations at different levels of recursion. Additionally, for this accuracy at input size 8, it is also interesting to see that the algorithm abandons the use of recursion completely, opting instead to solve the problem with the ideal direct solver.

6.5 Programmability

While determining the programmer productivity of a new language can be quite challenging, our anecdotal experience has shown that our extensions greatly simplify the task of programming variable accuracy code. We have written a variable accuracy version of the 2D Poisson's equation solver benchmark in the PetaBricks language both before and after we added our new variable accuracy language constructs. We found that our new language features greatly simplified the benchmark, resulting in a 15.6x reduction in code size.

In the original PetaBricks language, we were able to leverage the language's autotuner to perform the search through the accuracy performance space. However, unlike in the new code, much of the heavy lifting during the training stage had to be performed by code written by the programmer. For example, the original code contained specialized transforms used only during training that predetermined the levels of accuracy required at each recursive step in the multigrid algorithm. These transforms stored this information in a file which was used during subsequent non-training runs. Additionally, we were able to eliminate a substantial amount of code duplication because we were able to represent variable accuracy directly instead of being forced to

represent it as algorithmic choices. Finally, we should note that had the original code been written in a language without autotuning support, the code would have no doubt been even more complex if it were to not expose the numerous choices in the multigrid solver to the user.

7. Related Work

Seeking approximating program outputs is a common technique for determining solutions to computationally hard tasks, such as NP-complete problems. For such problems, programmers often manually employ soft computing, fuzzy logic and artificial intelligence techniques to trade precision s computational tractability [28]. Likewise, in a similar manner, precision is often sacrificed for performance when real-time constraints make precise algorithms unfeasible. However, despite this, few systems exists today to help programmers develop such programs.

There has been a fair amount of research focusing on approximating floating-point computations. For example, Hull *et al.* developed Numeric Turing [14], a programming language for scientific computation that allows developers to dynamically specify the desired precision of floating-point values throughout their program. Numeric Turing works in conjunction with a specialized coprocessor that performs the variable accuracy arithmetic needed to maintain the desired precision. While effective, the specialized hardware incurs a fairly large barrier to entry.

Techniques such as Loop Perforation [19], Code Perforation [12], and Task Skipping [21, 22] automatically transform existing computations and/or programs. The resulting new computations may skip subcomputations (for example loop iterations or tasks) that may not be needed to achieve a certain level of accuracy. The computation may perform less computational work and therefore execute more quickly and/or consume less energy. While this approach can be performed robustly in many cases, it is not sound and therefore may require additional programmer time to verify the correctness of the perforated code (should such verification be needed or desired). On the other hand, our system provides a new language and compiler that enables programmers to write new programs with variable accuracy in mind right from the start. In addition to altering loop iterations, our language allows programmers to specify entirely different algorithmic choices and data representations that may be optimal for different accuracies.

PowerDail [13] is a system that converts static configuration parameters that already exist in a program into dynamic knobs that can be tuned at runtime. Their system can then change these knobs at runtime to make the program meet performance and power usage goals. They use an application wide quality of service metric to measure the loss or gain in accuracy.

Being developed concurrently to our work is the Green system [5], whose primary goal is to lower the power requirements of programs. Green uses pragma-like annotations to allow multiple versions of a function that have different power requirements and resulting accuracies. Green uses a global quality of service metric to monitor the impact of running the various approximate versions of the code. PetaBricks differs from Green in that it supports multiple accuracy metrics per program, allows the definition of a much larger class of algorithmic choices, has parallelism integrated with its choice model, and contains a robust genetic autotuner.

Finally, there exists a large variety of work related to PetaBrick's autotuning approach of optimizing programs. For example, a number of empirical autotuning frameworks have been developed for building efficient, portable libraries in specific domains. PHiPAC [6] is an autotuning system for dense matrix multiply. ATLAS [24] utilizes empirical autotuning to produce a cache-contained matrix multiply. FFTW [11] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [15] for sparse matrix computations, SPIRAL [20] for digital signal processing, UHFFT [2] for FFT on multicore systems, and OSKI [23] for sparse matrix kernels. In addition to these systems, various performance models and tuning techniques [7, 17, 25, 27] have been proposed to evaluate and guide automatic performance tuning.

8. Conclusions

We have presented a new programming model where tradeoffs between time and accuracy are exposed at the language level to the compiler. To the best of our knowledge, this is the first programming language that incorporates a comprehensive solution for choices relating to algorithmic accuracy. We have developed novel techniques to automatically search the space of algorithms and parameters to construct an optimized algorithm for each accuracy level required. We have implemented a suite of 6 benchmarks that are representative of commonly used algorithms that leverage variable accuracy. Using these benchmarks, we have provided evidence of the importance of exposing accuracy and algorithmic choices to the compiler when autotuning variable accuracy programs.

Using our new programming model, library writers are able to hide the implementation details of their variable accuracy algorithms without limiting the user's choice of desired accuracy and the resulting performance. Moreover, library users do not need to burden themselves with learning the implementation specific parameters that would otherwise have been exposed. Additionally, our extensions allow variable accuracy algorithms to adapt to new environments not known to the original programmer. These new environments include new architectures, where different relative costs of operations may change the accuracy/performance trade-offs of the underlying choices. In addition, our system allows variable accuracy algorithms to adapt to changes in required

accuracy and accuracy metrics. This ability to adapt extends the lifetime of programs, since the program can automatically change to meet the needs of future generations of users.

References

[1] *Matlab Documentation for fmincon()*, 2010. URL `http://www.mathworks.com/access/helpdesk/help/toolbox/optim/ug/fmincon.html`.

[2] Ayaz Ali, Lennart Johnsson, and Jaspal Subhlok. Scheduling FFT computation on SMP and multicore systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 293–301, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-768-1.

[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 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[4] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *In SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, January 2007.

[5] W. Baek and T. Chilimbi. Green: A system for supporting energy-conscious programming using principled approximation. Technical Report MSR-TR-2009-89, Microsoft Research, 2009 August.

[6] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM. ISBN 0-89791-902-5.

[7] Eric A. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 80–91, New York, NY, USA, 1995. ACM. ISBN 0-89791-701-6.

[8] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial*. SIAM, second edition, 2000.

[9] Wenceslas Fernandez de la Vega and George S. Lueker. Bin packing can be solved within 1+epsilon in linear time. *Combinatorica*, 1(4):349–355, 1981.

[10] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, August 1997.

[11] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005. Invited paper, special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] 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.

[13] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Power-aware computing with dynamic knobs. Technical Report MIT-CSAIL-TR-2010-027, Massachusetts Institute of Technology, May 2010.

[14] T.E. Hull, M.S. Cohen, and C.B. Hall. Specifications for a variable-precision arithmetic coprocessor. In *In proceedings of the 10th IEEE Symposium on Computer Arithmetic.*, June 1991.

[15] Eun-jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Science*, pages 127–136. Springer, 2001.

[16] David S. Johnson and Michael R. Garey. A 71/60 theorem for bin packing. *Journal of Complexity*, 1(1):65 – 106, 1985. ISSN 0885-064X.

[17] Michail G. Lagoudakis and Michael L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the International Conference On Machine Learning*, pages 511–518. Morgan Kaufmann, 2000.

[18] Carol A. Markowski and Edward P. Markowski. Conditions for the effectiveness of a preliminary test of variance. 1990.

[19] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffman, and Martin Rinard. Quality of service profiling. In *Proceedings of the 2010 International Conference on Software Engineering*, Cape Town, South Africa, May 2010.

[20] Markus Puschel, Jose M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Aca Gacic Franz Franchetti, Robbert W. Johnson Yevgen Voronenko, Kang Chen, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. In *Proceedings of the IEEE*, volume 93, pages 232–275. IEEE, Feb 2005.

[21] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334, 2006. ISBN 1-59593-282-8.

[22] Martin Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 369–386, 2007. ISBN 978-1-59593-786-5.

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

[24] Richard Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X.

[25] Samuel Webb Williams, Andrew Waterman, and David A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008.

[26] DM Young. *Iterative solution of large linear systems*. Dover Publications, 2003. ISBN 0486425487.

[27] Hao Yu, Dongmin Zhang, and Lawrence Rauchwerger. An adaptive algorithm selection framework. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 278–289, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2229-7.

[28] Lotfi A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Commun. ACM*, 37(3):77–84, 1994. ISSN 0001-0782.