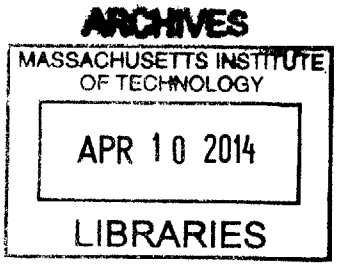
Autotuning Programs with Algorithmic Choice

by

Jason Ansel

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February, 2014



Author.....
Department of Electrical Engineering and Computer Science
January 6, 2014

Certified by.....
Saman Amarasinghe
Professor
Thesis Supervisor

Accepted by.....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Autotuning Programs with Algorithmic Choice

by

Jason Ansel

Submitted to the Department of Electrical Engineering and Computer Science on January 6, 2014, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering

Autotuning Programs with Algorithmic Choice

by

Jason Ansel

Abstract

The process of optimizing programs and libraries, both for performance and quality of service, can be viewed as a search problem over the space of implementation choices. This search is traditionally manually conducted by the programmer and often must be repeated when systems, tools, or requirements change. The overriding goal of this work is to automate this search so that programs can change themselves and adapt to achieve performance portability across different environments and requirements. To achieve this, first, this work presents the PetaBricks programming language which focuses on ways for expressing program implementation search spaces at the language level. Second, this work presents OpenTuner which provides sophisticated techniques for searching these search spaces in a way that can easily be adopted by other projects.

PetaBricks is a implicitly parallel language and compiler where having multiple implementations of multiple algorithms to solve a problem is the natural way of programming. Choices are provided in a way that also allows our compiler to tune at a finer granularity. The PetaBricks compiler autotunes programs by making both fine-grained as well as algorithmic choices. Choices also include different automatic parallelization techniques, data distributions, algorithmic parameters, transformations, and blocking. PetaBricks also introduces novel techniques to autotune algorithms for different convergence criteria or quality of service requirements. We show that the PetaBricks autotuner is often able to find non-intuitive poly-algorithms that outperform more traditional hand written solutions.

OpenTuner is a open source framework for building domain-specific multi-objective program autotuners. OpenTuner supports fully-customizable configuration representations, an extensible technique representation to allow for domain-specific techniques, and an easy to use interface for communicating with the program to be autotuned. A key capability inside OpenTuner is the use of ensembles of disparate search techniques simultaneously; techniques that perform well will dynamically be allocated a larger proportion of tests. OpenTuner has been shown to perform well on complex search spaces up to 10^{3000} possible configurations in size.

Thesis Supervisor: Saman Amarasinghe

Title: Professor

Contents

Abstract 7

Acknowledgments 13

1 Introduction 15
- 1.1 Contributions . . . 20
  - 1.1.1 Language . . . 21
  - 1.1.2 Process and Compilation . . . 21
  - 1.1.3 Autotuning Techniques . . . 22

2 The PetaBricks Language 25
- 2.1 Sorting as an Example of Algorithmic Choice . . . 25
- 2.2 Iteration Order Choices . . . 28
- 2.3 Variable Accuracy . . . 31
  - 2.3.1 K-Means Example . . . 32
  - 2.3.2 Language Support for Variable Accuracy . . . 34
  - 2.3.3 Variable Accuracy Language Features . . . 36
  - 2.3.4 Accuracy Guarantees . . . 38
- 2.4 Input Features . . . 38
- 2.5 A More Complex Example . . . 40
  - 2.5.1 The Choice Space for SeparableConvolution . . . 43
- 2.6 Language Specification . . . 45
  - 2.6.1 Transform Header Flags . . . 45
  - 2.6.2 Rule Header Flags . . . 50
  - 2.6.3 Matrix Definitions . . . 51
  - 2.6.4 Matrix Regions . . . 52

3 The PetaBricks Compiler 53
- 3.1 PetaBricks Compiler . . . 54
- 3.2 Parallelism in Output Code . . . 59
- 3.3 Autotuning System and Choice Framework . . . 60
- 3.4 Runtime Library . . . 62
- 3.5 Code Generation for Heterogeneous Architectures . . . 62
  - 3.5.1 OpenCL Kernel Generation . . . 63
  - 3.5.2 Data Movement Analysis . . . 64
  - 3.5.3 Runtime System . . . 65

3.5.4 Memory Management . . . 69
3.5.5 GPU Choice Representation to the Autotuner . . . 70
3.6 Choice Space Representation . . . 72
3.6.1 Choice Configuration Files . . . 72
3.7 Deadlocks and Race Conditions . . . 73
3.8 Automated Consistency Checking . . . 74

4 Benchmarks and Experimental Analysis 75
4.1 Fixed Accuracy Benchmarks . . . 76
4.1.1 Symmetric Eigenproblem . . . 76
4.1.2 Sort . . . 78
4.1.3 Matrix Multiply . . . 80
4.2 Autotuning Parallel Performance . . . 80
4.3 Effect of Architecture on Autotuning . . . 81
4.4 Variable Accuracy Benchmarks . . . 82
4.4.1 Bin Packing . . . 83
4.4.2 Clustering . . . 84
4.4.3 Image Compression . . . 85
4.4.4 Preconditioned Iterative Solvers . . . 86
4.5 Experimental Results . . . 88
4.5.1 Analysis . . . 88
4.5.2 Programmability . . . 91
4.6 Heterogeneous Architectures Experimental Results . . . 92
4.6.1 Methodology . . . 92
4.6.2 Benchmark Results and Analysis . . . 95
4.6.3 Heterogeneous Results Summary . . . 100
4.7 Summary . . . 102

5 Multigrid Benchmarks 103
5.1 Autotuning Multigrid . . . 104
5.1.1 Algorithmic choice in multigrid . . . 104
5.1.2 Full dynamic programming solution . . . 106
5.1.3 Discrete dynamic programming solution . . . 108
5.1.4 Extension to Autotuning Full Multigrid . . . 109
5.1.5 Limitations . . . 111
5.2 Results . . . 112
5.2.1 Autotuned multigrid cycle shapes . . . 112
5.2.2 Performance . . . 116
5.2.3 Effect of Architecture on Autotuning . . . 124

6 The PetaBricks Autotuner 127
6.1 The Autotuning Problem . . . 128
6.1.1 Properties of the Autotuning Problem . . . 129
6.2 A Bottom Up EA for Autotuning . . . 130
6.3 Experimental Evaluation . . . 136
6.3.1 GPEA . . . 136

6.3.2 Experimental Setup . . . 136
6.3.3 INCREA vs GPEA . . . 137
6.3.4 Representative runs . . . 139

7 Input Sensitivity **145**
7.1 Usage . . . 148
7.2 Input Aware Learning . . . 148
7.2.1 A Simple Design and Its Issues . . . 148
7.2.2 Design of the Two Level Learning . . . 149
7.2.3 Level 1 . . . 150
7.2.4 Level 2 . . . 152
7.2.5 Discussion of the Two Level Learning . . . 156
7.3 Evaluation . . . 157
7.3.1 Input Features and Inputs . . . 158
7.3.2 Experimental Results . . . 159
7.3.3 Input Generation . . . 162
7.3.4 Model of Diminishing Returns with More Landmark Configurations . . . 164

8 Online Autotuning **167**
8.1 Competition Execution Model . . . 169
8.1.1 Other Splitting Strategies . . . 169
8.1.2 Time Multiplexing Races . . . 170
8.2 SiblingRivalry Online Learner . . . 171
8.2.1 High Level Function . . . 172
8.2.2 Online Learner Objectives . . . 173
8.2.3 Selecting the Safe and Seed Configuration . . . 174
8.2.4 Adaptive Mutator Selection (AMS) . . . 174
8.2.5 Population Pruning . . . 176
8.3 Experimental Results and Discussion . . . 177
8.3.1 Sources of Speedups . . . 177
8.3.2 Load on a System . . . 178
8.3.3 Migrating Between Microarchitectures . . . 180
8.4 Hyperparameter Tuning . . . 182
8.4.1 Tuning the Tuner . . . 183
8.4.2 Evaluation metrics . . . 185
8.4.3 Results . . . 187
8.4.4 Hyperparameter Robustness . . . 189

9 OpenTuner **193**
9.1 The OpenTuner Framework . . . 194
9.1.1 OpenTuner Usage . . . 195
9.1.2 Search Techniques . . . 197
9.1.3 Configuration Manipulator . . . 198
9.1.4 Objectives . . . 201
9.1.5 Search Driver and Measurement . . . 201
9.1.6 Results Database . . . 202

9.2 Experimental Results . . . 202
9.2.1 GCC/G++ Flags . . . 204
9.2.2 Halide . . . 206
9.2.3 High Performance Linpack . . . 207
9.2.4 PetaBricks . . . 209
9.2.5 Stencil . . . 211
9.2.6 Unitary Matrices . . . 211
9.2.7 Results Summary . . . 212

10 Related Work **215**
10.1 Autotuning . . . 215
10.2 Variable Accuracy . . . 219
10.3 Multigrid . . . 221
10.4 Autotuning Techniques . . . 221
10.5 Online Autotuning . . . 222
10.6 Autotuning Heterogeneous Architectures . . . 223

11 Conclusions **227**

Bibliography **251**

Acknowledgments

This thesis includes text and experiments from a large subset of my publications while at MIT [8, 9, 11–13, 15–17, 37, 53, 110, 113]. The coauthors and collaborators of these publications deserve due credit and thanks: Clarice Aiello, Saman Amarasinghe, Cy Chan, Yufei Ding, Alan Edelman, Sam Fingeret, Sanath Jayasena, Shoaib Kamil, Kevin Kelley, Erika Lee, Deepak Narayanan, Marek Olszewski, Una-May O'Reilly, Maciej Pacula, Phitchaya Mangpo Phothilimthana, Jonathan Ragan-Kelley, Xipeng Shen, Michele Tartara, Kalyan Veeramachaneni, Yod Watanaprakornku, Yee Lok Wong, Kevin Wu, Minshu Zhan, and Qin Zhao.

I would specifically like to acknowledge some substantial contributions by: Cy Chan for multigrid benchmarks; Yufei Ding for input sensitivity; Maciej Pacula for bandit based online learning; Jonathan Ragan-Kelley for Halide (one of the six shown OpenTuner applications) and help on the GPU backend; and Phitchaya Mangpo Phothilimthana for the GPU backend and help providing feedback on this thesis.

Additionally, much of my academic work is beyond the scope of this thesis, but helped shape my experiences at MIT. This includes: collaboration on deterministic multi-threading [106, 107, 109], which was the primary project of Marek Olszewski, who is a dear friend, former office mate, and founder of Locu; self modifying code under software fault isolation [14], while I was a Google; and distributed checkpointing [10, 43, 44, 123] which was my focus as an undergraduate.

I am grateful for the advice and guidance of my advisor Saman Amarasinghe. I would also like to extend special thanks to my undergraduate advisor Gene Cooperman. Without their guidance I would not be where I am today. I would also like to thank the other members of the Commit group and those who provided feedback, both on draft manuscripts and on practice talks. I am

grateful to the members of my thesis committee, Armando Solar-Lezama and Martin Rinard for their feedback. I finally would like to thank my family and friends for their love and support.

This work is partially supported by NSF Award CCF-0832997, DOE Award DE-SC0005288, DOD DARPA Award HR0011-10-9-0009, and an award from the Gigascale Systems Research Center. We wish to thank the UC Berkeley EECS department for generously letting us use one of their machines for benchmarking and NVIDIA for donating a graphics card used to conduct experiments. We would like to thank Clarice Aiello for contributing the Unitary benchmark program. We gratefully acknowledge Connelly Barnes and Andrew Adams for helpful discussions and bug fixes related to autotuning the Halide project. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Chapter 1

Introduction

The developers of languages and tools have invested a great deal of collective effort into extending the lifetime of software. To a large degree, this effort has succeeded. Millions of lines of code written decades ago are still being used in new programs. Early languages such as Fortran largely achieved their goals of having a single portable source code than can be compiled for almost any machines. Libraries and interfaces allow code to be reused in ways the original programmer could not foresee. Languages, such as Java, provide virtual machines allowing compiled bytecode to run on almost any system. We live in an era of write it once and use everywhere software.

What we have not yet achieved is performance portability. Hand coded optimizations for one system, often are not the best choice for another system. The result is obsolete optimizations that get carried to newer architectures in this portable software. A typical example of this can be found in the C++ Standard Template Library (STL) routine `std::stable_sort` (Figure 1.1), distributed with the current version of GCC and whose implementation dates back to at least the 2001 SGI release of the STL. This legacy code contains a hard coded optimization, a cutoff constant of 15 between merge and insertion sort, that was designed for machines of the time, having 1/100th the memory of modern machines. Our tests have shown that higher cutoffs (around 60-150) perform much better on current architectures. However, because the optimal cutoff is dependent on architecture, cost of the comparison routine, element size, and parallelism, no single hard-coded

value will suffice. This type of hard coded optimization is typical in modern performance critical software.

/usr/include/c++/4.7.3/bits/stl_algo.h:

```
3508 /// This is a helper function for the stable sorting routines.
3509 template<typename _RandomAccessIterator>
3510   void
3511   __inplace_stable_sort(_RandomAccessIterator __first,
3512                         _RandomAccessIterator __last)
3513   {
3514     if (__last - __first < 15)
3515       {
3516         std::__insertion_sort(__first, __last);
3517         return;
3518       }
3519     _RandomAccessIterator __middle = __first + (__last - __first) / 2;
3520     std::__inplace_stable_sort(__first, __middle);
3521     std::__inplace_stable_sort(__middle, __last);
3522     std::__merge_without_buffer(__first, __middle, __last,
3523                                 __middle - __first,
3524                                 __last - __middle);
3525 }
```

Figure 1.1: Hard coded optimization constant 15 in std::stable_sort

While the paradigm of write once and run it everywhere is great for productivity, a major sacrifice made in these efforts is performance. Write once use everywhere often becomes write once slow everywhere. We need programs with *performance portability*, where programs can re-optimize themselves to achieve the best performance on widely different execution targets. The process of optimizing computer programs can be viewed as a search problem over the space of ways to implement a program. This search is traditionally manually conducted by the programmer, through iterative development and testing, and takes place in many domains. In high performance computing, there is constant optimization to fit algorithms to each new generation of supercomputers. The optimizations for one generation often will not be suitable for the next and may need to be undone. In areas such as graphics, optimizations must often be done to fit each hardware target a system may run on, which may range from a laptop to a powerful desktop computer with a large graphics card. In many web services applications, such as search, programs

will be optimized for quality of service so that they achieve the optimal results in a fixed time budget or a given level of service in the minimum time. In each of these domains, much of the difficulty of programming is in conducting a manual search over the solution space.

One of the most promising techniques to achieve performance portability is *program autotuning.* Rather than hard-coding optimizations, that only work for a single microarchitecture, or using fragile heuristics, program autotuning exposes a search space of program optimizations that can be explored automatically. Autotuning is used to search this optimization space to find the best configuration to use for each platform. Often these optimization search spaces contain complex decisions such as constructing recursive poly-algorithms to solve a problem. Autotuning can also be used to meet quality of service requirements of a program, such as sacrificing accuracy to meet hard execution time limits.

While using autotuners, instead of heuristics or models, for choosing traditional compiler optimizations can be successful at optimizing a single algorithm, when an algorithmic change is required to boost performance, the burden is put on the programmer to incorporate the new algorithm. If a composition of multiple algorithms is needed for the best performance, the programmer must write all the algorithms, the glue code to connect them together, and figure out the best switch over points. Making such changes automatically to the program requires heroic analysis or the analyses required is beyond the capability of all modern compilers. However, this information is clearly known to the programmer. The needs of modern computing require an *either...or* statement, which would allow the programmer to give a menu of algorithmic choices to the compiler. We also show how choices between different erogenous processing units, such as GPUs, can be automated by the PetaBricks compiler.

The need for a concise way to represent these algorithmic choices motivated the design of the PetaBricks programming language, which will be covered in detail in Chapter 2. Programs written in PetaBricks can naturally describe multiple algorithms for solving a problem and how they can be fit together. This information is used by the PetaBricks compiler and runtime, described in Chapter 3, to create and autotune an optimized hybrid algorithm. The PetaBricks system also optimizes and autotunes parameters relating to data distribution, parallelization, iteration, and

accuracy. The knowledge of algorithmic choice allows the PetaBricks compiler to automatically parallelize programs using the algorithms with the most parallelism when it is beneficial to do so.

Another issue that PetaBricks addresses is variable accuracy algorithms. Traditionally, language designers and compiler writers have operated under the assumption that programs require a fixed, precisely defined behavior; however, this is not always the case in practice. For many 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. We broadly define these types of programs as *variable accuracy* algorithms. Often, these accuracy choices are hard coded into the program. The programmer will hand code parameters which provide good accuracy performance on some testing inputs. This choice over accuracy of the implementation can be viewed as another dimension in the solution space. An interesting example of variable accuracy can be found in Chapter 5, which shows our multigrid benchmarks. We show a novel dynamic programming approach for synthesizing multigrid V-cycles shapes tailored to a specific problem an accuracy target.

A large part of this thesis (Chapters 6 and 9) address the challenges developing techniques for autotuning. A number of novel techniques have been developed to efficiently search the search spaces created. The PetaBricks autotuner is based on an evolutionary algorithm (EA), however an off-the-shelf EA does not typically take advantage of shortcuts based on problem properties and this can sometimes make it impractical because it takes too long to run. A general shortcut is to solve a small instance of the problem first then reuse the solution in a compositional manner to solve the large instance which is of interest. Usually solving a small instance is both simpler (because the search space is smaller) and less expensive (because the evaluation cost is lower). Reusing a sub-solution or using it as a starting point makes finding a solution to a larger instance quicker. This shortcut is particularly advantageous if solution evaluation cost grows with instance size. It becomes more advantageous if the evaluation result is noisy or highly variable which requires additional evaluation sampling.

Another fundamental problem, which will be addressed in Chapter 7, is input sensitivity. For a large class of problems, the best optimization to use depends on the input data being processed.

For example, sorting an almost-sorted list can be done most efficiently with a different algorithm than one optimized for sorting random data. This problem is worse in autotuning systems, because there is a danger that the auotuner will create an algorithm specifically optimized for the inputs it is provided during tuning. This may be suboptimal for inputs later encountered in production or be a compromise solution that is not best for any one input but performs well overall. For many problems, no single optimized program exists which can match the performance of a collection of optimized programs autotuned for different subsets of the input space. This problem of input sensitivity is exacerbated by several features common to many classes of problems and types of autotuning systems. Many autotuning systems must handle large search spaces and variable accuracy algorithms with multiple objectives. They encounter inputs with non-superficial features that require domain specific knowledge to extract, Each of these challenges makes the problem of input sensitivity more difficult in a unique way.

Chapter 7 will present a general means of automatically determining what algorithmic optimization to use when different ones suit different inputs. While input sensitivity seems to be intertwined with the complexity of large optimization spaces and input spaces, we show that it can be resolved via simple extensions to our existing autotuning system. We show that the complexity of input sensitivity can be managed, and that a small number of input optimized programs is often sufficient to get most of the benefits of input adaptation.

In Chapter 8, 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.

Chapter 9 will present OpenTuner, a new framework for building domain-specific program autotuners. OpenTuner features an extensible configuration and technique representation able to support complex and user-defined data types and custom search heuristics. It contains a library of predefined data types and search techniques to make it easy to setup a new project. Thus, OpenTuner solves the custom configuration problem by providing not only a library of data types that will be sufficient for most projects, but also extensible data types that can be used to support more complex domain specific representations when needed.

1.1 Contributions

The overriding goal of this thesis work is to automate this optimization search and take it out of the hands over the programmer. The programmer should concisely define the search space, and then the compiler should perform the search. When the processors, coprocessors, accuracy requirements, or inputs programs should change themselves and adapt to work optimally in different each different environment. This provides performance portability, extends the life of software, and saves programmer effort.

To achieve this, this thesis has three main focuses:

- Designing language level solutions for concisely representing the implementation choice spaces of programs in ways that can be searched automatically.
- Developing process and compilation techniques to manage and explore these program search spaces.
- Creating novel autotuning techniques and search spaces representations to efficiently solve these search problems where prior techniques would fail or be inefficient.

The remainder of this subsection will expand on the contributions in each of these areas.

1.1.1 Language

The language level contributions in this work are realized in design and implementation of the PetaBricks programming language compiler.

- We introduce the PetaBricks programming language, which, to best of our knowledge, is the first language that enables programmers to express algorithmic choice at the language level.
- PetaBricks introduces the concept of the *either ... or ...* statement as a means to express recursive poly-algorithm choices.
- PetaBricks uses a declarative outer syntax to express iteration order choices around its imperative inner syntax. Multiple declarations in this declarative outer syntax allow the programmer to cleanly express algorithmic choices in handling boundary cases and iteration order dependency choices in a way that can automatically be searched by an autotuner.
- We have introduced the first language level support for variable accuracy. This includes support for multiple accuracy metrics and accuracy targets, which provide a general-purpose way for users to define arbitrary accuracy requirements in any domain and expands the scope of algorithmic choices that can be represented.
- PetaBricks introduces the concept of the *for_enough* loop, a loop whose input-dependent iteration bounds are determined by the autotuner.
- We introduce language level support for declaring variable accuracy input features, to enable programs that dynamically adapt to each input.

1.1.2 Process and Compilation

- While autotuners have exploited coarse-grained algorithmic choice at a programmatic level, to best of our knowledge PetaBricks is the first compiler that incorporates fine-grained

algorithmic choices in program optimization. PetaBricks includes novel compilation structures to organize and manage this choice space.

- We introduce a new model for online autotuning, called SiblingRivalry, where the processor resources are divided and two candidate configurations compete against each other in parallel. This solves the problem of exploring a volatile configuration space, by dedicating half the resources to a known safe configuration.

- We developed the first system to simultaneously address the interdependent problems of variable accuracy algorithms and input sensitivity.

- We introduce the first system which automatically determines the best mapping of programs in a high level language across a heterogeneous CPU/GPU mix of parallel processing units, including placement of computation, choice of algorithm, and optimization for specialized memory hierarchies. With this, a high-level, architecture independent language can obtain comparable performance to architecture specific, hand-coded programs.

1.1.3 Autotuning Techniques

The autotuning technique contributions of this work can be divided into two main areas. First, a number of novel techniques have been developed in the context of PetaBricks. The remaining contributions in the area of autotuning techniques are in the OpenTuner project, a new open source framework for building domain-specific multi-objective program autotuners.

- We have created a multi-objective, practical evolutionary autotuning algorithm for high-dimensional, multi-modal, and experimentally shown its efficacy on non-linear configuration search spaces up to 10^{3000} possible configurations in size.

- We introduce a bottom-up learning algorithm that drastically improves convergence time for many benchmarks. It does this by using tests conducted on much smaller program instances to bootstrap the learning process.

- We show a novel dynamic programming solution to efficiently build tuned multigrid V-cycle algorithms that combine methods with varying levels of accuracy while providing that a final target accuracy is met. These autotuned V-cycle shapes are non-obvious and perform better than the cycle shapes used in practice.

- We have developed a technique for mapping variable accuracy code so that it can be efficiently autotuned without the search space growing prohibitively large. This is done by simultaneously autotuning for many different accuracy targets.

- We present a multi-armed bandit based algorithm for online autotuning, which, to the best of our knowledge, is the first application of evolutionary algorithms to the problem of online autotuning of computer programs.

- We show a through principled search of the parameter space of our multi-armed bandit algorithm that a single "robust" parameter setting exists, which performs well across a large suite of benchmarks.

- To solve the problem of input sensitive programs, we present a novel two level approach, which first clusters the input feature space into input classes and then builds adaptive overhead-aware classifiers to select between different input optimized algorithms for these classes, solves the problem of input sensitivity for much larger algorithmic search spaces than would be tractable using prior techniques.

- We offer a principled understanding of the influence of program inputs on algorithmic autotuning, and the relations among the spaces of inputs, algorithmic configurations, performance, and accuracy. We identify a key disparity between input properties, configuration, and execution behavior which makes it impractical to produce a direct mapping from input properties to configurations and motivates our two level approach.

- We show both empirically and theoretically that for many types of search spaces there are rapidly diminishing returns to adding more and more input adaption to a program. A little bit of input adaptation goes a long way, while a large amount is often unnecessary.

- We present OpenTuner, which is a general autotuning framework to describe complex search spaces which contain parameters such as schedules and permutations.
- OpenTuner introduces the concept of ensembles of search techniques to program autotuning, which allow many search techniques to work together to find an optimal solution.
- OpenTuner provides more sophisticated search techniques than typical program auotuners. This enables expanded uses of program autotuning to solve more complex search problems and pushes the state of the art forward in program autotuning in a way that can easily be adopted by other projects.

Chapter 2

The PetaBricks Language

This chapter will provide an overview of the base PetaBricks language and all extensions made to it over the course of this thesis work. The key features of the language, such as algorithmic choice and variable accuracy, will be introduced with example programs.

2.1 Sorting as an Example of Algorithmic Choice

As a motivation example, consider the problem of sorting. There are a huge number of ways to sort a list [31]. For example: insertion sort, quick sort, merge sort, bubble sort, heap sort, radix sort, and bucket sort. Most of these sorting algorithms are recursive, thus, one can switch between algorithms at any recursive level. This leads to an exponential number of possible algorithmic compositions that make use of more than one primitive sorting algorithm.

Since sorting is a well known problem, most readers will have some intuition about the optimal algorithm: for very small inputs, insertion sort is faster; for medium sized inputs, quick sort is faster (in the average case); and for very large inputs radix sort becomes fastest. Thus, the optimal algorithm might be a composition of the three, using quick sort and radix sort to recursively decompose the problem until the subproblem is small enough for insertion sort to take over. Once parallelism is introduced, the optimal algorithm gets more complicated. It often makes sense to use merge sort at large sizes because it contains more parallelism than quick sort (the merging performed at each recursive level can also be parallelized). Another aspect that impacts performance beyond

available parallelism is the locality of different memory access patterns of each algorithm, which can impact the cache utilization.

Even with this detailed intuition (which one may not have for other algorithms), the problem of writing an optimized sorting algorithm is nontrivial. Using popular languages today, the programmer would still need to find the right cutoffs between algorithms. This has to be done through manually tuning or using existing autotuning techniques that would require additional code to integrate (as discussed in Chapter 1). If the programmer puts too much control flow in the inner loop for choosing between a wide set of choices, the cost of control flow may become prohibitive. The original simple code for sorting will be completely obscured by this glue, thus making the code hard to comprehend, extend, debug, port and maintain.

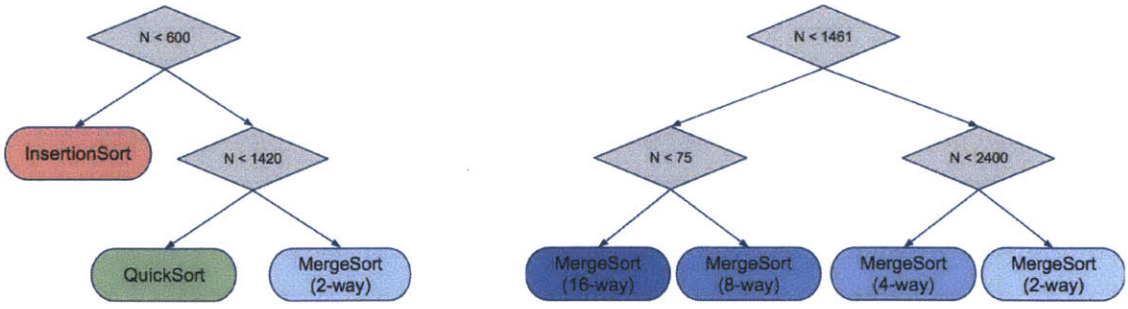
PetaBricks solves this problem by automating both algorithm selection and autotuning in the compiler. The programmer specifies the different sorting algorithms in PetaBricks and how they fit together, but does *not* specify when each one should be used. The compiler and autotuner will experimentally determine the best composition of algorithms to use and the respective cutoffs between algorithms. This has added benefits in portability. On a different architecture, the optimal cutoffs and algorithms may change. The PetaBricks program can adapt to this by merely retuning.

Figure 2.1 shows a partial implementation of Sort in PetaBricks. The *Sort* function is defined taking an input array named *in* and an output array named *out*. The *either...or* primitive implies a space of possible polyalgorithms. The semantics are that when the *ether...or* statement is executed, exactly one of the clauses will be executed, and the choice of which sub block to execute is left up to the autotuner. In our example, many of the sorting routines (QuickSort, MergeSort, and RadixSort) will recursively call Sort again, thus, the *either...or* statement will be executed many times dynamically when sorting a single list. The autotuner uses evolutionary search to construct polyalgorithms which make one decision at some calls to the *either...or* statement, then different decisions in the recursive calls.

These polyalgorithms are realized through *selectors* (sometimes called *decision trees*) which efficiently select which algorithm to use at each recursive invocation of the *either...or* statement.

```
1 function Sort
2 to out[n]
3 from in[n]
4 {
5   either {
6     InsertionSort(out, in);
7   } or {
8     QuickSort(out, in);
9   } or {
10    MergeSort(out, in);
11  } or {
12    RadixSort(out, in);
13  } or {
14    BitonicSort(out, in);
15  }
16 }
```

Figure 2.1: PetaBricks code for *Sort*



(a) Optimized for an Intel Xeon X5460

(b) Optimized for a Sun UltraSPARC T1

Figure 2.2: Example synthesized selectors for Sort.

Figure 2.2 shows two examples of selectors that might be synthesized by the autotuner for some specific input and architecture.

2.2 Iteration Order Choices

Current language constructs are too prescriptive in specifying an exact iteration order for loop nests. In order to take advantage of architectural features such as vectorization and cache hierarchies, compilers are forced to perform heroic analysis to find better iteration orders. This not only hinders the compilers ability to get the best performance but also requires the programmer to specify additional constraints beyond what is required to describe many algorithms. Natural mathematical notations such as sets and tensors do not provide an iteration order.

To expose iteration order choices to the compiler, PetaBricks does not require an outer sequential control flow, it uses a declarative outer syntax combined with an imperative inner syntax. The outer syntax defines parameterized data dependencies, which bring data into the local scope of each imperative block. The compiler must find a legal execution order that computes inputs before outputs and every output exactly once. The outer syntax can define multiple ways of computing the same value as another way of expressing algorithmic choice. This is done by defining multiple rules to compute the same region of data.

The language is built around two major constructs, *transforms* and *rules*. The *transform*, analogous to a function, defines an algorithm that can be called from other transforms, code written in other languages, or invoked from the command line. The header for a transform defines *to* and *from* arguments, which represent inputs and outputs, and *through* intermediate data used within the transform. The size in each dimension of these arguments is expressed symbolically in terms of free variables, the values of which must be determined by the PetaBricks compiler.

The user encodes choice by defining multiple *rules* in each transform. Each rule defines how to compute a region of data in order to make progress towards a final goal state. Rules have explicit dependencies parametrized by free variables set by the compiler. Rules can have different granularities and intermediate state. The compiler is required to find a sequence of rule applications that will compute all outputs of the program. The explicit rule dependencies allow automatic

parallelization and automatic detection and handling of corner cases by the compiler. The rule header references *to* and *from* regions which are the inputs and outputs for the rule. The compiler may apply rules repeatedly, with different bindings to free variables, in order to compute larger data regions. Additionally, the header of a rule can specify a *where* clause to limit where a rule can be applied. The body of a rule consists of C++-like code to perform the actual work.

Instead of outer sequential control flow, Petabricks users specify which transform to apply, but not how to apply it. The decision of when and which rules to apply is left up the compiler and runtime system to determine. This has the dual advantages of both exposing algorithmic choices to the compiler and enabling automatic parallelization. It also gives the compiler a large degree of freedom to autotune iteration order and storage.

Figure 2.3 shows an example PetaBricks transform, that performs matrix multiplication. The transform header is on lines 1 to 5, which defines the inputs outputs and 7 intermediate matrices which may or may not be used. The first rule (line 8 to 10) is the straightforward way of computing a single matrix element. With the first rule alone the transform would be correct, the remaining rules add choices. Rule 2 (lines 14 to 16) operated on a different granularity, a 4x4 tile, and could contain a vectorized version. Next come rules to compute the intermediate $M1$ though $M7$ defined in Strassen's algorithm. This intermediate data will only be used in the final 4 rules are chosen, which compute the quadrents of the output from $M1$ to $M7$.

The PetaBricks transform is a side effect free function call as in any common procedural language. The major difference with PetaBricks is that we allow the programmer to specify multiple rules to convert the inputs to the outputs for each transform. Each rule converts parts or all of the inputs to parts or all of the outputs and may be implemented using multiple different algorithms. It is up to the PetaBricks compiler to determine which rules are necessary to compute the whole output, and the autotuner to decide which of these rules are most computationally efficient for a given architecture and input.

Taken together, the set of possible pathways through the graph of rules in a transform may be thought of as forming a directed acyclic dependency graph with various paths leading from the transform's inputs (represented as the graph's source) to the transform's outputs (represented as

```
1 transform StrassenMatrixMultiply
2 from A[n, n], B[n, n]
3 to AB[n, n]
4 using M1[n/2, n/2], M2[n/2, n/2], M3[n/2, n/2], M4[n/2, n/2],
5       M5[n/2, n/2], M6[n/2, n/2], M7[n/2, n/2]
6 {
7   // Base case 1: Compute a single element
8   to(AB.cell(x,y) out)
9   from(A.row(y) a, B.column(x) b) {
10    out = dot(a,b);
11  }
12
13  // Base case 2: Vectorized version to compute 16 elements at a time
14  to(AB.region(x, y, x + 4, y + 4) out)
15  from(A.region(0, y, n, y + 4) a,
16       B.region(x, 0, x + 4, n) b){ ... }
17
18  // Compute intermediate data
19  to(M1 m1)
20  from(A.region(0, 0, n/2, n/2) a11,
21       A.region(n/2, n/2, n, n) a22,
22       B.region(0, 0, n/2, n/2) b11,
23       B.region(n/2, n/2, n, n) b22)
24  using(t1[n / 2, n / 2], t2[n/2, n / 2]) {
25    spawn MatrixAdd(t1, a11, a22);
26    spawn MatrixAdd(t2, b11, b22);
27    sync;
28    StrassenMatrixMultiply(m1, t1, t2);
29  }
30
31  // ... M2 through M7 rules omitted ...
32
33  // Recursive case: Compute the 4 quadrants using M1 through M7
34  to(AB.region(0, 0, n/2, n/2) c11) from(M1 m1, M4 m4, M5 m5, M7 m7){
35    MatrixAddAddSub(c11, m1, m4, m7, m5);
36  }
37  to(AB.region(n/2, 0, n, n/2) c12) from(M3 m3, M5 m5){
38    MatrixAdd(c12, m3, m5);
39  }
40  to(AB.region(0, n/2, n/2, n) c21) from(M2 m2, M4 m4){
41    MatrixAdd(c21, m2, m4);
42  }
43  to(AB.region(n/2, n/2, n, n) c22) from(M1 m1, M2 m2, M3 m3, M6 m6){
44    MatrixAddAddSub(c22, m1, m3, m6, m2);
45  }
46 }
```

Figure 2.3: PetaBricks code for MatrixMultiply

the graph's sink). Intermediate nodes in the graph represent intermediate data structures produced and used during the computation of the output. The user specifies how to get to nodes from other nodes by defining *rules*. The rules correspond to the edges (or sometimes hyperedges) of the graph, which encode both the data dependencies of the algorithm, as well as the code needed to produce the edge's destinations from the sources.

In addition to choices between different algorithms, many algorithms have configurable parameters that change their behavior. A common example of this is the branching factor in recursively algorithms such as merge sort or radix sort. To support this PetaBricks has a *tunable* keyword that allows the user to export custom paramcters to the autotuner. PetaBricks analyzes where these tunable values are used, and autotunes them at an appropriate time in the learning process.

PetaBricks contains many additional language features such as: `%{ ... }%` escapes used to embed raw C++ in the output file; a *generator* keyword for specifing a transform to be used to supply input data during training; matrix *versions*, with a `A<0..n>` syntax, useful when defining iterative algorithms; rule priorities and *where* clauses are used to handle corner cases gracefully; and template transforms, similar to templates in C++, where each template instance is autotuned separately.

2.3 Variable Accuracy

Another issue that PetaBricks addresses is variable accuracy algorithms. This choice over accuracy of the implementation can be viewed as another dimension in the solution space. There are many different classes of variable accuracy algorithms.

One class of variable accuracy algorithms are *approximation algorithms* in the area of soft computing [165]. 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 [50]. 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 [163]. 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. 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.

To help illustrate variable accuracy algorithms in PetaBricks, we will first introduce a new example program, kmeans, which will used as a running example in this Section.

2.3.1 K-Means Example

Figure 2.4 presents an example PetaBricks program, kmeans. This 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 data *Centroids* to keep track of the current center of each cluster. 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 2.5, which indicates the dependencies of each of the three rules.

```
1 transform kmeans
2 from Points[n,2] // Array of points (each column
3                  // stores x and y coordinates)
4 using 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 2.4: PetaBricks code for kmeans.

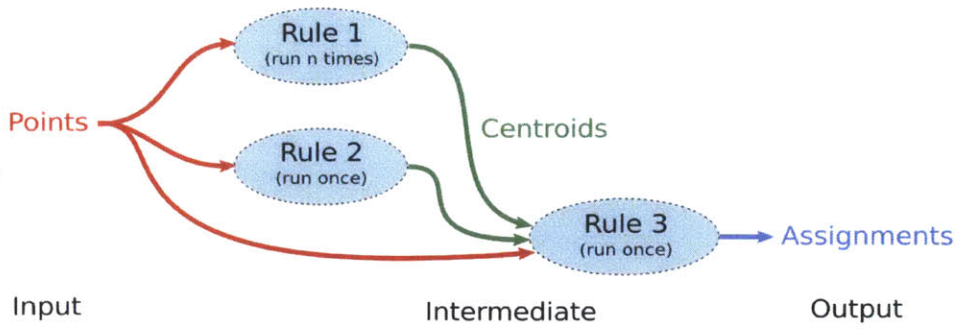


Figure 2.5: Dependency graph for kmeans example. The rules are the vertices while each edge represents the dependencies of each rule. Each edge color corresponds to each named data dependence in the pseudocode.

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 will not be executed until the intermediate data structure Centroids has been computed by one of the first two rules.

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.

2.3.2 Language Support for Variable Accuracy

To realize support for variable accuracy we extend the idea of algorithmic choice to include choices between different accuracies. We provide a mechanism to allow the user to specify how accuracy should be measured. Our 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 (described below) that utilize runtime checking of accuracy.

```
1 transform kmeans
2 accuracy_metric kmeansaccuracy
3 accuracy_variable k
4 from Points[n,2] // Array of points (each column
5                   // stores x and y coordinates)
6 using Centroids[k,2]
7 to Assignments[n]
8 {
9   // Rule 1:
10  // One possible initial condition: Random
11  // set of points
12  to(Centroids.column(i) c) from(Points p) {
13    c=p.column(rand(0,n))
14  }
15
16  // Rule 2:
17  // Another initial condition: Centerplus initial
18  // centers (kmeans++)
19  to(Centroids c) from(Points p) {
20    CenterPlus(c, p);
21  }
22
23  // Rule 3:
24  // The kmeans iterative algorithm
25  to(Assignments a) from(Points p, Centroids c) {
26    for_enough {
27      int change;
28      AssignClusters(a, change, p, c, a);
29      if (change==0) return; // Reached fixed point
30      NewClusterLocations(c, p, a);
31    }
32  }
33 }
35 transform kmeansaccuracy
36 from Assignments[n], Points[n,2]
37 to Accuracy
38 {
39  Accuracy from(Assignments a, Points p){
40    return sqrt(2*n/SumClusterDistanceSquared(a,p));
41  }
42 }
```

Figure 2.6: PetaBricks code for variable accuracy kmeans. (The new variable accuracy code is highlighted in light blue.)

Figure 2.6 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.

2.3.3 Variable Accuracy Language Features

PetaBricks contains the following language features used to support variable accuracy algorithms:

- 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. An accuracy variable is the same as a *tunable* except that it provides some hints to the autotuner about its effect.

- 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. This field is optional and 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 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 making each function polymorphic on accuracy.

  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 runtime check for the level of accuracy attained. If this check fails the programmer can insert calls to retry with the next higher level of accuracy or the programmer 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.

2.3.4 Accuracy Guarantees

PetaBricks supports the following three 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 of program inputs to determine statistical bounds on an accuracy metric to within a desired level of confidence. Confidence bounds are calculated by assuming accuracy is normally distributed and measuring the mean and standard deviation. This implicitly assumes that accuracy distributions are the same across training and production inputs.

- *Runtime checking* can provide a hard guarantee of accuracy by testing accuracy at runtime and performing additional work if accuracy requirements are not met. Runtime 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. In these cases the *accuracy_metric* can just return a constant based on the program configuration.

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

2.4 Input Features

For a large class of problems, the best optimization to use depends on the input data being processed. We broadly call this feature of algorithms input sensitivity and will discuss our handling of them in detail in Chapter 7.

```
1 function Sort
2 to out[n]
3 from in[n]
4 input_feature Sortedness, Duplication
5 {
6   either {
7     InsertionSort(out, in);
8   } or {
9     QuickSort(out, in);
10  } or {
11    MergeSort(out, in);
12  } or {
13    RadixSort(out, in);
14  } or {
15    BitonicSort(out, in);
16  }
17 }
18
19 function Sortedness
20 from in[n]
21 to sortedness
22 tunable double level (0.0, 1.0)
23 {
24   int sortedcount = 0;
25   int count = 0;
26   int step = (int)(level*n);
27   for(int i=0; i+step<n; i+=step) {
28     if(in[i] <= in[i+step]) {
29       // increment for correctly ordered
30       // pairs of elements
31       sortedcount += 1;
32     }
33     count += 1;
34   }
35   if(count > 0)
36     sortedness = sortedcount / (double) count;
37   else
38     sortedness = 0.0;
39 }
40
41 function Duplication
42 from in[n]
43 to duplication
44 ...
```

Figure 2.7: PetaBricks code for Sort with input features. (The new input feature code is highlighted in light blue.)

PetaBricks provides language support for input sensitivity by adding the keyword *input_feature*, shown on lines 4 and 5 of Figure 2.7. The *input_feature* keyword specifies a programmer-defined function, a *feature extractor*, that will measure some domain specific property of the input to the function. A feature extractor must have no side effects, take the same inputs as the function, and output a single scalar value. The autotuner will call this function as necessary.

Feature extractors may have tunable parameters which control their behavior. For example, the *level* tunable on line 23 of Figure 2.7, is a value that controls the sampling rate of the sortedness feature extractor. Higher values will result in a faster, but less accurate measure of sortedness. *Tunable* is a general language keyword that specifies a variable to be set by the autotuner and two values indicating the allowable range of the tunable (in this example between 0.0 and 1.0).

2.5 A More Complex Example

To end the chapter we show a more complex example PetaBricks program, Convolution, and its performance characteristics when mapped to different GPU/CPU targets. We will show the power of algorithmic choice previewing the results for convolution. The full set of results an be found in Chapter 4.

Figure 2.8 shows the PetaBricks code for *SeparableConvolution*. The top-level transform computes *Out* from *In* and *Kernel* by *either* computing a single-pass 2D convolution, *or* computing two separate 1D convolutions and storing intermediate results in *buffer*. Our compiler maps this program to multiple different executables which perform different *algorithms* (separable vs. 2D blur), and map to the GPU in different ways. This program computes the convolution of a 2D matrix with a separable 2D kernel. The main transform (starting on line 1) maps from input matrix *In* to output matrix *Out* by convolving the input with the given *Kernel*. It does this using one of two rules:

- *Choice 1* maps from *In* to *Out* in a single pass, by directly applying the *Convolve2D* transform.

```
1 transform SeparableConvolution
2 from In[w, h], Kernel[KWIDTH]
3 to Out[w-KWIDTH+1, h-KWIDTH+1]
4 {
5   // Choice 1: single pass 2D convolution
6   to(Out out) from(In in, Kernel kernel) {
7     Convolve2D(out, in, kernel);
8   }
9
10  // Choice 2: two-pass separable convolution
11  to(Out out) from(In in, Kernel kernel)
12  using(buffer[w-KWIDTH+1, h]) {
13    ConvolveRows(buffer, in, kernel);
14    ConvolveColumns(out, buffer, kernel);
15  }
16 }
17
18 transform Convolve2D
19 from In[w, h], Kernel[KWIDTH]
20 to Out[w-KWIDTH+1, h-KWIDTH+1]
21 {
22  Out.cell(x,y)
23  from(In.region(x, y, x+KWIDTH+1, y+KWIDTH+1) in, Kernel kernel)
24  {
25    ElementT sum = 0;
26    for(int x = 0; x < KWIDTH; x++)
27      for(int y = 0; y < KWIDTH; y++)
28        sum += in.cell(x,y) * kernel.cell(x) * kernel.cell(y);
29    return sum;
30  }
31 }
```

(continued in Figure 2.9).

Figure 2.8: PetaBricks code for SeparableConvolution.

(continued from Figure 2.8).

```
32
33 transform ConvolveRows
34 from In[w, h], Kernel[KWIDTH]
35 to Out[w-KWIDTH+1, h]
36 {
37   Out.cell(x,y)
38   from(In.region(x, y, x+KWIDTH, y+1) in, Kernel kernel)
39   {
40     ElementT sum = 0;
41     for(int i = 0; i < KWIDTH; i++)
42       sum += in.cell(i,0) * kernel.cell(i);
43     return sum;
44   }
45 }
46
47 transform ConvolveColumns
48 from In[w, h], Kernel[KWIDTH]
49 to Out[w, h-KWIDTH+1]
50 {
51   Out.cell(x,y)
52   from(In.region(x, y, x+1, y+KWIDTH) in, Kernel kernel)
53   {
54     ElementT sum = 0;
55     for(int i = 0; i < KWIDTH; i++)
56       sum += in.cell(0,i) * kernel.cell(i);
57     return sum;
58   }
59 }
```

Figure 2.9: PetaBricks code for SeparableConvolution

Choice Type	Parameters	Possible Configurations
Parallelism	3	10^{12}
Work Stealing Scheduler	6	6
Iteration Order	9	10^{43}
Algorithmic	25	10^{72}
Total	43	10^{128}

Figure 2.10: Counts of different types of choices generated by PetaBricks for SeparableConvolution.

- *Choice 2* maps from *In* to *Out* in two passes, by first performing the *ConvolveRows* transform, storing its result in *buffer*, and then performing the *ConvolveColumns* transform on this intermediate result.

The three transforms are each defined in terms of a single data parallel rule which, for each point in *Out*, computes a sum of the points in the corresponding *KWIDTH*-sized region of *In* weighted by the corresponding points in the *Kernel*. *ConvolveRows* and *ConvolveColumns* apply only in the horizontal and vertical directions, respectively, while *Convolve2D* applies both dimensions simultaneously, iterating over a 2D window of *In* for each output point. Based on the experimental data, the autotuner chooses which rules to run (a choice of algorithm), and how to map them onto the machine, including runtime scheduling strategy, parallelism, data placement, and the choice of processors on which to run.

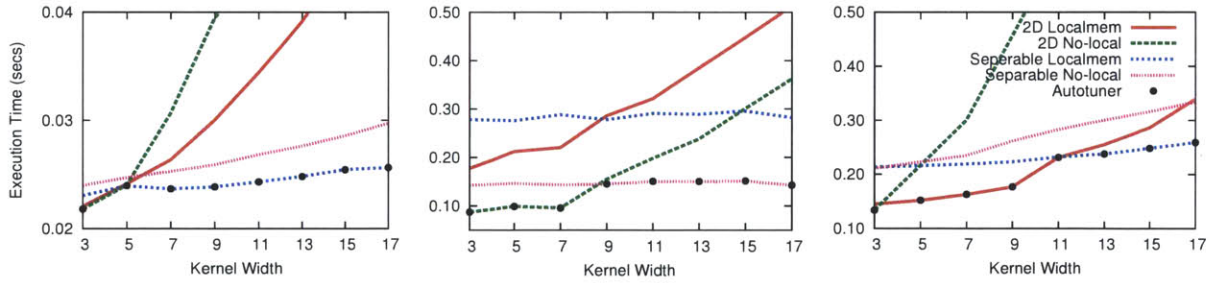
2.5.1 The Choice Space for SeparableConvolution

Figure 2.10 summarizes the number of choices generated by the PetaBricks compiler for SeparableConvolution. The top-level *SeparableConvolution* transform compiles into two simple rules, each of which computes the entire *Out* matrix from the entire *In* matrix. Internally, these rules apply the *Convolve** transforms which are defined elementwise, without sequential dependencies, and so can be executed in a data parallel fashion over the entire output space. As one choice, these data parallel rules can be compiled directly into equivalent OpenCL kernels, with each *work-item* computing a single cell of *Out*. Automatic compilation into OpenCL is discussed in Section 3.5.

Because of its simple data parallel pattern, this example runs most efficiently when all *Convolve** transform are executed entirely in the OpenCL runtime, but the ideal choices of algorithm and

mapping to the memory system vary across machines, and for each machine across different kernel sizes. However, even though not optimal for this benchmark, the choice to run some fraction of the computation on the CPU is also available to the autotuner.

Intuitively, when blur size (kernel's width) is large, implementing the separable convolution as a pair of 1D passes over the data, first in the horizontal and then in the vertical direction, performs asymptotically less work than a single 2D convolution pass. Starting each block of work with an extra phase which prefetches a block of *In* into scratchpad memory shared by all processors in the block saves memory bandwidth compared to separately loading many overlapping inputs through the (slower) main memory hierarchy. But when the kernel size is small, the overhead (in bandwidth for intermediate results, and in kernel invocations) of multiple passes overtakes the computational complexity advantage of separable computation, making a single 2D pass faster, and the overhead of explicit prefetching into OpenCL local memory cannot be offset by the smaller number of redundant loads saved across a block of work. Where these tradeoffs dominate varies between machines. And the underlying machines vary further: on an OpenCL target where the shared memory maps to the same caches and buses used for all loads and stores, the explicit prefetch phase nearly always represents wasted work.



(a) Desktop (b) Server (c) Laptop

Figure 2.11: Execution time (lower is better) of different possible mappings of `SeparableConvolution` to OpenCL with varying kernel widths on three test systems when the input size is 3520×3520. The three test systems are described in Section 4.6.1. Note that each mapping is optimal for at least one machine and kernel width. In handwritten OpenCL, these would correspond to four distinct programs the programmer would need to write.

Figure 2.11 shows the running times of four distinct OpenCL choices (2D Convolution and Separable Convolution, each with and without local memory prefetching) generated by PetaBricks

for *SeparableConvolution*, on three different machines (described in Section 4.6.1), over kernel sizes from 3-17. The ideal choice (the lowest line at a given point) varies among the four choices, depending on both the targeted machine and the kernel size. As expected, the execution times of the single-pass 2D algorithms increase faster than those of the two-pass separable algorithms as the kernel size increases, and the use of scratchpad memory helps little when the kernel size is small and worsens the performance on the CPU OpenCL runtime. But how exactly these effects interact, and the relative behavior of each choice, varies substantially across machines.

In short, simply mapping this program to the OpenCL runtime is not sufficient. How it is mapped (whether or not to prefetch shared inputs into scratchpad memory), and which algorithms to use, is a complex choice, highly dependent on the specific workload and target architecture.

2.6 Language Specification

This section provides a exhaustive list of the syntax of the PetaBricks language. Figures 2.12, 2.13, and 2.14 show a slightly simplified grammar for the PetaBricks language, with the exception of the inner *EmbeddedCPlusPlusSuset* in rule bodies. This is *C++* code (suitable to go in a function body) with the addition of the *either...or* statement. *Ident* is a variable name literal and *Integer* and *Float* are numeric literals.

2.6.1 Transform Header Flags

The following options can appear in the header for a transform or a function. Note that a function is syntactic sugar for a transform with a single rule that takes the same inputs/outputs.

- *'accuracy_bins' FloatList*

  Specific additional accuracy targets which should be trained for.

- *'accuracy_metric' Ident*

  The name of another PetaBricks transform to measure the quality of an output.

- *'accuracy_variable' ConfigItem*

  'config' ConfigItem

```
PetaBricksProgram → Nil
PetaBricksProgram → Function PetaBricksProgram
PetaBricksProgram → Transform PetaBricksProgram

Function → 'function' Ident TransformHeaderList RuleBody

Transform → 'transform' Ident TransformHeaderList TransformBody

TransformHeaderList → Nil
TransformHeaderList → TransformHeader TransformHeaderList

TransformHeader → 'main'
TransformHeader → 'memoized'
TransformHeader → 'param' Ident
TransformHeader → 'from' MatrixDefList
TransformHeader → 'through' MatrixDefList
TransformHeader → 'using' MatrixDefList
TransformHeader → 'to' MatrixDefList
TransformHeader → 'template' Ident '(' Integer ',' Integer ')'
TransformHeader → 'accuracy_metric' Ident
TransformHeader → 'accuracy_bins' FloatList
TransformHeader → 'generator' Ident
TransformHeader → 'input_feature' Ident
TransformHeader → 'config' ConfigItem
TransformHeader → 'tunable' ConfigItem
TransformHeader → 'accuracy_variable' ConfigItem

ConfigItem → ConfigItemFlagList Ident ConfigItemBounds

ConfigItemFlagList → Nil
ConfigItemFlagList → ConfigItemFlag ConfigItemFlagList

ConfigItemFlag → 'int'
ConfigItemFlag → 'double'
ConfigItemFlag → 'float'
ConfigItemFlag → 'sizespecific'
ConfigItemFlag → 'accuracyhint'

ConfigItemBounds → Nil
ConfigItemBounds → '(' ')'
ConfigItemBounds → '(' Float ')'
ConfigItemBounds → '(' Float ',' Float ')'
ConfigItemBounds → '(' Float ',' Float ',' Float ')'

MatrixDefList → MatrixDef
MatrixDefList → MatrixDef ',' MatrixDefList

MatrixDef → Ident OptVersion OptSize
```

Figure 2.12: Grammar for the PetaBricks language. Page 1 of 3 (Figures 2.12, 2.13, 2.14).

```
OptVersion → Nil
OptVersion → '<' Formula '>'
OptVersion → '<' Formula '...' Formula '>'

OptSize → Nil
OptSize → '[' FormulaList ']'

TransformBody → '{' RuleList '}'

RuleList → Nil
RuleList → Rule RuleList

Rule → RuleFlagList RuleHeader RuleBody

RuleFlagList → Nil
RuleFlagList → RuleFlag RuleFlagList

RuleFlag → 'primary'
RuleFlag → 'secondary'
RuleFlag → 'priority' '(' Integer ')'
RuleFlag → 'rotatable'
RuleFlag → 'recursive'
RuleFlag → 'recursive' '(' Formula ')'
RuleFlag → 'duplicate' '(' Ident ',' Integer ',' Integer ')'
RuleFlag → 'rule' Ident

RuleHeader → Region RuleHeaderFrom RuleHeaderThrough OptWhere
RuleHeader → RuleHeaderTo RuleHeaderFrom RuleHeaderThrough OptWhere

RuleHeaderFrom → 'from' '(' NamedRegionList ')'

RuleHeaderTo → 'to' '(' NamedRegionList ')'

NamedRegionList → Nil
NamedRegionList → NamedRegion
NamedRegionList → NamedRegion ',' NamedRegionList

NamedRegion → Region Ident
NamedRegion → Region Ident '=' Formula

RuleHeaderThrough → Nil
RuleHeaderThrough → 'through' '(' MatrixDefList ')'
RuleHeaderThrough → 'using' '(' MatrixDefList ')'
```

Figure 2.13: Grammar for the PetaBricks language. Page 2 of 3 (Figures 2.12, 2.13, 2.14).

```
OptWhere →  Nil
OptWhere →  'where' FormulaRelation

RuleBody →  '{' EmbeddedCPlusPlusSubset '}'

Formula →  Ident
Formula →  Integer
Formula →  Float
Formula →  '(' Formula ')'
Formula →  Formula '+' Formula
Formula →  Formula '-' Formula
Formula →  Formula '*' Formula
Formula →  Formula '/' Formula
Formula →  Formula '^' Formula
Formula →  '-' Formula

FormulaRelation →  Formula '==' Formula
FormulaRelation →  Formula '<' Formula
FormulaRelation →  Formula '>' Formula
FormulaRelation →  Formula '<=' Formula
FormulaRelation →  Formula '>=' Formula
FormulaRelation →  FormulaRelation 'or' FormulaRelation
FormulaRelation →  FormulaRelation 'and' FormulaRelation

Region →  Ident OptVersion RegionAccessor

RegionAccessor →  Nil
RegionAccessor →  '.' all '(' ')'
RegionAccessor →  '.' cell '(' FormulaList ')'
RegionAccessor →  '.' region '(' FormulaList ')'
RegionAccessor →  '.' row '(' Formula ')'
RegionAccessor →  '.' col '(' Formula ')'
RegionAccessor →  '.' column '(' Formula ')'
RegionAccessor →  '.' slice '(' Integer ',' Formula ')'

FormulaList →  Nil
FormulaList →  Formula
FormulaList →  Formula ',' FormulaList

FloatList →  Float
FloatList →  Float ',' FloatList

Nil →
```

Figure 2.14: Grammar for the PetaBricks language. Page 3 of 3 (Figures 2.12, 2.13, 2.14).

'tunable' ConfigItem

Variables stored in the configuration file. Unless the keyword `config` is used, the autotuner will tune the value. Additional keywords *double*, *int*, or *float* can be added to change the type. If the keyword *accuracy_variable* or *accuracyhint* is used it will notify the autotuner that the variable is likely to effect accuracy. If the keyword *sizespecific* is given, the variable may hold a different value for each input size, and will generate a synthezied function in the program configuration. Optional default, min, and max bounds can be added.

- *'from' MatrixDefList*

  List of inputs to the transform.

- *'generator' Ident*

  The name of another PetaBricks transform to generate training inputs.

- *'input_feature' Ident*

  The name of another PetaBricks transform to extract a feature from the input.

- *'main'*

  Marks a single transform as the entry point for the PetaBricks program.

- *'memoized'*

  Enables automatic memoization for results of the given function. Outputs to the function will be cached in a hash table and re-used when the function is called with the same inputs.

- *'param' Ident*

  A constant integer input parameter, allowed to be used in size formulas.

- *'template' Ident '(' Integer ',' Integer ')'*

  Define many copies of the transform, with the value of Ident set to all values between the two given Integers.

- *'through' MatrixDefList*

  'using' MatrixDefList

  Intermediate data (semantically equivalent).

- *'to' MatrixDefList*

  List of outputs from the transform.

- *'transform' Ident*

  The name of the transform to be defined.

2.6.2 Rule Header Flags

The following flags can go in the header before a rule:

- *'duplicate' '(' Ident ',' Integer ',' Integer ')'*

  Define many copies of the rule, with Ident set to each value between the two Integers.

- *'from' '(' NamedRegionList ')'*

  Define the symbolic input regions from the inputs and outputs of the transform.

- *'primary'*

  'secondary'

  'priority' '(' Integer ')'

  Set the priority for the rule. If multiple rules apply to the same region only the rules with lowest integer priority can be used.

  priority(0) is the same as *primary*. *priority(1)* is the default. *priority(2)* is the same as *secondary*.

- *'recursive'*

  For backwards compatibility, does nothing.

- *'recursive' '(' Formula ')'*

  Specify the decision trees generated should be conditioned on the given formula instead of the size of the largest input dimension.

- *'rotatable'*

  Generate 4 copies of the given rule, each with the coordinate space offset by 90 degrees for 2D inputs.

- *'rule' Ident*

  Provide a optional name for the rule (used in debugging / visualization).

- *'through' '(' MatrixDefList ')'*

  'using' '(' MatrixDefList ')'

  Intermediate data used in the scope of the rule.

- *'to' '(' NamedRegionList ')'*

  Define the symbolic output regions from the outputs of the transform.

- *'where' FormulaRelation*

  A condition to limit the region where the rule can be applied. The rule may only be used when FormulaRelation evaluates to true. This is evaluated statically at compile time .

2.6.3 Matrix Definitions

Input, output, and intermediate matrices can be defined in the *to*, *from*, *using*, and *through* clauses of the transform header. They can also be declared in the *using* and *through* clauses of the rule. They define a symbolic region, bases on a set of variables they share a transform scope and whose values are automatically inferred from the sizes of the regions a transform is called with.

A matrix definition can either be a single double, for example *A*, or can be array, *B[n]*, or have an arbitrary number of dimensions, *C[n, n]*. The sizes of each dimension can be a formula, for example *D[n, n, 2*n, n + m + 10]*. In this example the first two dimensions of matrix D must be the same size (n), the next dimension must be twice that size, and the third dimension can be any size and m will be bound to $|D_4| - n - 10$, where $|D_4|$ is the size of the forth dimension of D. (This is determined by solving for m.) If m and n were used in any other matrix definitions in the same transform they are required to have the same values.

Matrix definitions can also have versions, for example $E < 1...10 > [n]$, which can be used for iterative computations. This is syntactic sugar for adding an extra dimension.

2.6.4 Matrix Regions

Rules operate on regions of a matrix. These regions are bound to local variables in the scope of the rule. Regions can be specified using the following methods on the matrix:

- *'.' all '(' ')'*

  The entire matrix. This is the same as writing just the name of the matrix.

- *'.' cell '(' FormulaList ')'*

  A specific element in the matrix. FormulaList must contain a number of formulas equal to the number of dimensions.

- *'.' region '(' FormulaList ')'*

  A region of the matrix specified by begin coordinate (inclusive) and end coordinate (exclusive). FormulaList must contain a number of formulas equal to twice the number of dimensions.

- *'.' row '(' Formula ')'*

  Alias for *.slice(1, Formula)*

- *'.' col '(' Formula ')'*

  '.' column '(' Formula ')'

  Alias for *.slice(0, Formula)*

- *'.' slice '(' Integer ',' Formula ')'*

  Return a region with one fewer dimensions with a given Integer dimension replaced by Formula.

  For example: *C.cell(x, y)* could be rewritten as either: *C.slice(1, y).cell(x)*, *C.slice(0, x).cell(y)*, or *C.slice(0, x).slice(0, y).cell()*.

Chapter 3

The PetaBricks Compiler

The traditional goal of a compiler is to map a program to a specific piece of hardware. Most compiler optimizations reason about program transformations using a simplified model of a specific processor target. This simple model of the performance of the machine is often implied by the heuristics that guide optimization. The PetaBricks compiler takes a different approach. It exposes the choice space in its internal representations, and to the autotuner, so that it can be search automatically. The PetaBricks compiler manages complex, interdependent choices of *algorithm:* which algorithm to use; *placement:* on which resources to place computations and data; and *mapping:* how much parallelism to exploit and how to use specialized memories.

The PetaBricks compiler implementation consists of three main components:

- a source-to-source compiler from the PetaBricks language to C++;
- an autotuning system and choice framework to find optimal choices and set parameters; and
- a runtime library used by the generated code.

The relationship between these components is depicted in Figure 3.1. First, the source-to-source compiler executes and performs static analysis. The compiler encodes choices and tunable parameters in the output code so that autotuning can be performed. When autotuning is performed (either at compile time or at installation time), it outputs an application configuration file that controls when different choices are made. This configuration file can be tweaked by hand to force

specific choices. Optionally, this configuration file can be fed back into the compiler and applied statically to eliminate unused choices and allow additional optimizations.

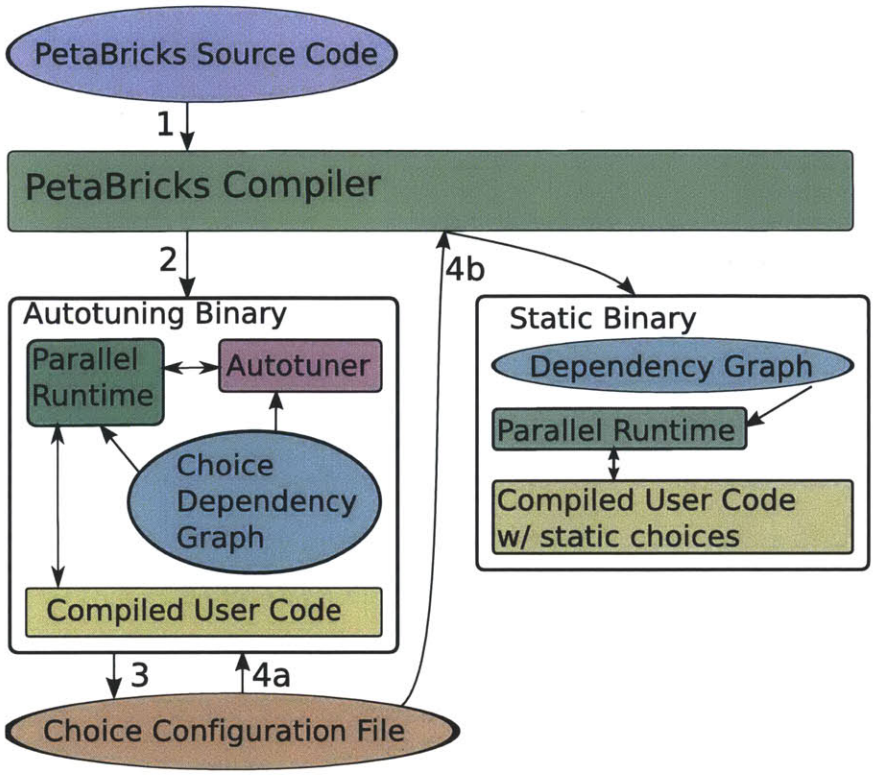


Figure 3.1: Interactions between the compiler and output binaries. First, in Steps 1 and 2, the compiler reads the source code and generates an autotuning binary. Next, in Step 3, autotuning is run to generate a choice configuration file. Finally, either the autotuning binary is used with the configuration file (Step 4a), or the configuration file is fed back into a new run of the compiler to generate a statically chosen binary (Step 4b).

3.1 PetaBricks Compiler

Figure 3.2 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.

A *region* is a core concept in the compiler. A region is a rectilinear subset of a single matrix defined at the transform level. A region is defined by a symbolic begin coordinate (inclusive) and

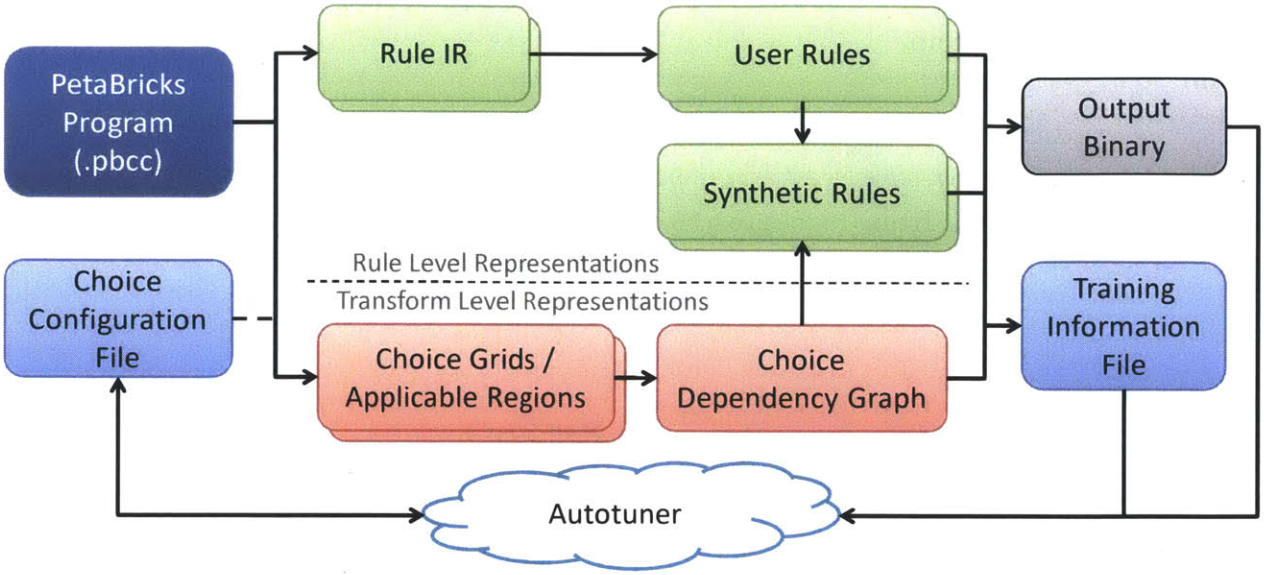


Figure 3.2: Flow for the compilation of a PetaBricks program with a *single* transform. (Additional transforms would cause the center part of the diagram to be duplicated.)

end coordinate (exclusive). We use the Maxima symbolic algebra library [122] to reason about region intersections and relations, and to guarantee all points of every matrix are covered.

To help illustrate the compilation process we will use the example transform `RollingSum`, shown in Figure 3.3. RollingSum computes an incremental (sometimes known as a cumulative) sum of an input list. It includes two rules: rule 0 computes an output directly, by iterating all input elements to the left; and rule 1 computes a value using a previously computed value to the left. An algorithm using only rule 0 is slower ($\Theta(n^2)$ operations), but can be executed in a data parallel way. An algorithm using only rule 1 is faster ($\Theta(n)$ operations), but has no parallelism and must be run sequentially.

Compilation consists of the following main phases. Figure 3.4 shows an overview the intermediate representation built up as the phases proceed. It starts as an abstract syntax tree and ends as a choice dependency graph. All compilation is done on symbolic regions of an unknown size and is general to any number of dimensions. The compilation steps are as follows:

Parsing and normalization. First, the input language is parsed into an abstract syntax tree. Rule dependencies are normalized by converting all dependencies into region syntax, assigning each

```
1 transform RollingSum
2 from A[n]
3 to B[n]
4 {
5   //rule0: sum all elements to the left
6   to(B.cell(i) b) from(A.region(0, i) in) {
7     b=sum(in);
8   }
9
10  //rule1: use the previously computed value
11  to(B.cell(i) b) from(A.cell(i) a,
12                       B.cell(i-1) leftSum) {
13    b=a+leftSum;
14  }
15 }
```

Figure 3.3: PetaBricks code for RollingSum. A simple example used to demonstrate the compilation process. The output element B_x is the sum of the input elements $A_0..A_x$.

Intermediate Representation	**Scope**	**Coverage**	**Data Type**
Applicable Regions	Rule	Where rule can be used	One region per rule / matrix
Choice Grids	Transform	All points of all matrices (non-overlapping)	List of regions per matrix
Choice Dependency Graph	Transform	All points of all matrices (non-overlapping)	Choice grids, plus annotated data dependency edges

Figure 3.4: Overview of major compiler intermediate representations.

rule a symbolic *center* (a arbitrary symbolic coordinate used to specify where a rule executes) and rewriting all dependencies to be relative to this center. (This is done using the Maxima symbolic algebra library [122].) In our `RollingSum` example, the center of both rules is equal to i, and the dependency normalization does not do anything other than replace variable names. For other inputs, this transformation would simplify the dependencies. For example, if 1 were added to every coordinate containing i in the input to rule 0 (leaving the meaning of the rule unchanged), the compiler would then assign the center to be $i+1$ and the dependencies would be been automatically rewritten to remove the added 1.

Applicable regions. Next, the region where each rule can legally be applied, called an *applicable*, is calculated. Applicable regions are non-overlapping regions of the output/intermediate symbolic space that cover all points. There is one set of applicable regions for each rule.

Applicable regions are first calculated for each dependency and then propagated upwards with intersections (this is again done by the linear equations solver and inference system). In rule 0 of our `RollingSum` example, both `b` and `in` (and thus the entire rule) have an applicable region of $[0, n)$. In rule 1 `a` and `b` have applicable regions of $[0, n)$ and leftSum has an applicable region of $[1, n)$ because it would read off the array for $i = 0$. These applicable regions are intersected to get an applicable region for rule 1 of $[1, n)$. Applicable regions can also be constrained with user defined *where* clauses, which are handled similarly.

Choice grid analysis. Next, we construct a *choice grid* for each symbolic matrix. The choice grid divides each matrix into rectilinear regions where a uniform set of rules are applicable. These regions must not overlap and must cover all points in every symbolic matrix.

Choice grids are constructed using an inference system to sort the applicable regions and divide them into smaller regions where a uniform set of choices apply. In our `RollingSum` example, the choice grid for B is:

$$\begin{aligned} [0, 1) &= \{\text{rule } 0\} \\ [1, n) &= \{\text{rule } 0, \text{ rule } 1\} \end{aligned}$$

and A is not assigned a choice grid because it is an input. For analysis and scheduling these two regions are treated independently.

It is in the choice grid phase that rule priorities are applied. If many rules are applicable to the same region, only those with the high priority can be used. In each region, all rules of non-minimal priority are removed. This feature is not used in our example code, but if the user had only provided rule 1, he could have added special handler for $[0, 1)$ by specifying a `secondary` rule. This mechanism becomes especially useful in higher dimensions where there are more corner cases.

Where clauses. Where clasues are handled in two ways:

- If the where clause can be statically proven to correspond to a rectilinear region: the where clause is eliminated and the applicable region for the rule is replaced with the intersection of the where clause region and the original applicable region. This is the fast path, and used by most of our benchmarks.

- If the where clause is not rectilinear: in choice grid analysis the rule is replaced by a meta-rule that is constructed by finding sets of rules that cover the entire rectolinear bounding box, and combining them up into a single meta-rule which dynamically executes the where clauses and switches between the sub-rules based on the result.

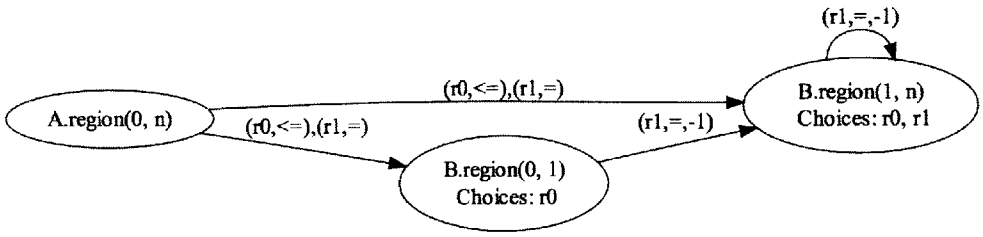


Figure 3.5: *Choice dependency graph* for RollingSum (in Figure 3.3). Arrows point the opposite direction of dependency (the direction data flows). Edges are annotated with rules and directions, offsets of 0 are not shown.

Choice dependency graph analysis. A choice dependency graph is constructed using each regions from the choice grid as nodes. Edges in this graph are created from the data dependencies of each rule in the choice grid. The paths through the graph from the inputs to the outputs represent legal execution orders. Each edge is annotated with the set of choices that require that edge, a direction of the data dependency, and an offset between rule centers for that dependency. The direction and offset information are especially useful for parallel scheduling; in many cases, they eliminate the need for a barrier before beginning the computation of a dependant matrix.

Figure 3.5 shows the choice dependency graph for our example `RollingSum`. The three nodes correspond to the input matrix and the two regions in the choice grid. Each edge is annotated with the rules that require it along with the associated directions and offsets. These annotations allow matrices to be computed in parallel when the rules chosen allow. This high level coarse graph is passed to the dynamic scheduler to execute in parallel at runtime. The dependency edges tell the

scheduler when it can split regions to compute them in parallel. The cost of the dynamic scheduler is negligible because scheduling is done from the top down on large regions of the matrix.

The graph for `RollingSum` does not require simplification, however if the graph were more complicated analysis would be required to simplify it. This simplification process is primarily focused around removing cycles of size greater than one. The input graph can contain cycles (provided union of the directions along the cycle points in towards a single hyper-quadrant), but the output schedule must be a topologically sorted directed acyclic graph (DAG) for other phases of the compiler to work. We remove cycles by merging strongly connected components into larger meta-nodes. This gives us choice dependency graph which is a DAG and is easier to process. The meta-nodes created by this process have a separate scheduler, which will finds an axis and direction for iterating which respects dependencies. For example, a cycle created by dependencies $A[i] \rightarrow B[i-1]$ and $B[i] \rightarrow [A-1]$ with be resolved by merging the A and B regions into a single meta-node and interleaving the execution of the two rules such that $A[i-1]$ and $B[i-1]$ are computed before $A[i]$ and $B[i]$.

The choice dependency graph is encoded in the output program for use by the autotuner and parallel runtime. It contains all information needed to explore choices and execute the program in parallel.

Code generation. Code generation has two modes. In the default mode choices and information for autotuning are embedded in the output code. This binary can be dynamically tuned, which generates a configuration file, and later run using this configuration file. In the second mode for code generation, a previously tuned configuration file is applied statically during code generation. The second mode is included since the C++ compiler can make the final code incrementally more efficient when the choices are eliminated.

3.2 Parallelism in Output Code

The PetaBricks runtime includes a parallel work stealing [67] dynamic scheduler. The scheduler works on *tasks* with a known interface. The generated output code will recursively create these

tasks and feed them to the dynamic scheduler to be executed. Dependency edges between tasks are detected at compile time and encoded in the tasks as they are created. A task may not be executed until all the tasks that it depends on have completed. These dependency edges expose all available parallelism to the dynamic scheduler and allow it to change its behavior based on autotuned parameters.

To expose parallelism and to help the dynamic scheduler schedule tasks in a depth-first search manner (see Section 3.4), the generated code is constructed such that functions suspended due to a call to a spawned task, can be migrated and executed on a different processor. This is difficult to achieve as the function's stack frame and registers need to be migrated. We support this by generating *continuation points*, points at which a partially executed function may be converted back into a task so that it can be rescheduled to a different processor. The continuation points are inserted after any code that spawns a task. This is implemented by storing all needed state to the heap.

The code generated for dynamic scheduling incurs some overhead, despite being heavily optimized. In order to amortize this overhead, the output code that makes use of dynamic scheduling is not used at the leaves of the execution tree where most work is done. The PetaBricks compiler generates two versions of every output function. The first version is the dynamically scheduled task-based code described above, while the second version is entirely sequential and does not use the dynamic scheduler. Each output transform includes a tunable parameter (set during autotuning) to decide when to switch from the dynamically scheduled to the sequential version of the code.

3.3 Autotuning System and Choice Framework

Autotuning is performed on the target system so that optimal choices and cutoffs can be found for that architecture. We have found that the best solution varies both by architecture and number of processors, these results are discussed in Chapter 4. The autotuning library is embedded in the output program whenever choices are not statically compiled in. Autotuning outputs an application

configuration file containing choices. This file can either be used to run the application, or it can be used by the compiler to build a binary with hard-coded choices.

The autotuner uses the *choice dependency graph* encoded in the compiled application. This choice dependency graph is also used by the parallel scheduler discussed in Section 3.4. This choice dependency graph contains the choices for computing each region and also encodes the implications of different choices on dependencies.

The intuition of the autotuning algorithm is that we take a bottom-up approach to tuning. To simplify autotuning, we assume that the optimal solution to smaller sub-problems is independent of the larger problem. In this way we build algorithms incrementally, starting on small inputs and working up to larger inputs.

The autotuner builds a multi-level algorithm. Each level consists of a range of input sizes and a corresponding algorithm and set of parameters. Rules that recursively invoke themselves result in algorithmic compositions. In the spirit of a genetic tuner, a population of candidate algorithms is maintained. This population is seeded with all single-algorithm implementations. The autotuner starts with a small training input and on each iteration doubles the size of the input. At each step, each algorithm in the population is tested. New algorithm candidates are generated by adding levels to the fastest members of the population. Finally, slower candidates in the population are dropped until the population is below a maximum size threshold. Since the best algorithms from the previous input size are used to generate candidates for the next input size, optimal algorithms are iteratively built from the bottom up. The autotuning process will be explained in more detail in Chapter 6.

All choices are represented in a flat configuration space. Dependencies between these configurable parameters are exported to the autotuner so that the autotuner can choose a sensible order to tune different parameters. The autotuner starts by tuning the leaves of the graph and works its way up. In the case of cycles, it tunes all parameters in the cycle in parallel, with progressively larger input sizes. Finally, it repeats the entire training process, using the previous iteration as a starting point, a small number of times to better optimize the result.

3.4 Runtime Library

The runtime library is primarily responsible for managing parallelism, data, and configuration. It includes a runtime scheduler as well as code responsible for reading, writing, and managing inputs, outputs, and configurations. The runtime is described in more detail in Section 3.5.3.

The runtime scheduler dynamically schedules tasks (that have their input dependencies satisfied) across processors to distribute work. When tasks reach a certain tunable cutoff size, they stop calling the scheduler and continue executing sequentially. Conversely, large data parallel tasks are divided up into smaller tasks, to increase the amount of parallelism available to the scheduler.

The scheduler attempts to maximize locality using a greedy algorithm that schedules tasks in a depth-first search order. Following the approach taken by Cilk [67], we distribute work with thread-private deques and a task stealing protocol. A thread operates on the top of its deque as if it were a stack, pushing tasks as their inputs become ready and popping them when a thread needs more work. When a thread runs out of work, it randomly selects a victim and steals a task from the *bottom* of the victim's deque. This strategy allows a thread to steal another thread's most nested continuation, which preserves locality in the recursive algorithms we observed. We use Cilk's THE [67] protocol to allow the victim to pop items of work from its deque without needing to acquire a lock in the common case.

3.5 Code Generation for Heterogeneous Architectures

This Section presents a solution to the problem of efficiently programming diverse heterogeneous systems, such as GPGPUs. Support for heterogeneous architectures consists of three different parts:

- Compiler passes and static analyses to automatically convert subsets of existing PetaBricks programs into OpenCL kernels that can be run on a variety of architectural backends, and coexist as choices available to the autotuner. This includes static analyses to help efficiently manage the memory of these coprocessing devices, and reduce data movement between kernel executions.

- A GPU management runtime that allows coprocessor devices to be efficiently utilized in the PetaBricks workstealing runtime without requiring the calling CPU thread to block on GPU operations, providing automatic memory management, and allowing efficient overlapping of computation and communication.

- Autotuning enhancements to search different divisions of GPU and CPU work, and to allow the exploration of choices on the GPU, such as choices of in which of the many memory spaces to place different data, and choices of how much data to run on the GPU.

3.5.1 OpenCL Kernel Generation

Since the PetaBricks language is more general (and supports calling arbitrary C/C++ code), only a subset of a PetaBricks program can be converted into OpenCL kernels. The process of generating additional choices for OpenCL execution is done early in the compilation process, before scheduling. The conversion process consists of three phases that are applied to each original user defined rule to inject equivalent synthetic OpenCL rules when possible.

In the first phase, a dependency analysis is performed to determine if the execution pattern of the rule fits into the OpenCL execution model. Sequential dependency patterns and data parallel dependency patterns can both be mapped to OpenCL kernels, but more complex parallel patterns, such as wavefront parallelism, can not be in our current implementation. It is possible that some sets of algorithmic choices will permit a mapping while others will not. The choice dependency graph is analyzed to determine if a rule can be mapped to OpenCL. The analysis looks at direction of the computed dependency for the strongly connected component associated with each of the rule's outputs. If there is no dependency, or the dependency is eliminated by selecting the rule choice under consideration, then the outer dependency pattern of the rule can be mapped to OpenCL.

If a rule passes the first phase, it goes to the second phase where the body of the transform is converted into OpenCL code. This phase includes a number of syntactic conversions. It is also responsible for rewriting data accesses to use GPU global memory, detecting a number of language constructs, such as calls to external libraries, that can not be mapped to OpenCL, and disqualifying a rule from being converted to OpenCL. This phase can also detect some language constructs (such

as inline native code), which can not be supported by OpenCL. The majority of these constructs are detected statically; however, there are a few more subtle requirements, sometimes OpenCL-implementation specific, which we detect by attempting to compile the resulting transform and rejecting synthetic OpenCL rules that do not compile.

The third phase attempts to optimize the *basic* version of OpenCL codes generated from the second phase by utilizing GPU local memory, known as OpenCL local memory or CUDA shared memory. For a subset of the mapped OpenCL rules, the *local memory* version can be generated. To do so, we analyze input data access pattern. A *bounding box* is a rectangular region of an input matrix that is used for computing an entry of the output matrix. If the size of the bounding box is a constant greater than one, then the local memory version of the GPU code is created; if the size of the bounding box is one, there is no need to copy the data into local memory because threads that share the same local memory never access the same data. The local memory version consists of two parts. First, all work-items on the GPU cooperate to load the data into local memory that will be accessed by the work-group they belong to. The second part is the actual computation derived from the basic version by replacing global memory accesses with local memory accesses. This local memory version is presented as a choice to the autotuner.

3.5.2 Data Movement Analysis

A second group of OpenCL analyses are performed at scheduling time in the compiler for determining how data is copied in and out of GPU memory. The PetaBricks compiler generates a unique schedule for each assignment of choices in a transform. These schedules represent parallelism as dependency trees that are given to the workstealing runtime. After each schedule is generated, it is analyzed for GPU data movement requirements. This analysis looks at preceding rule applications in the schedule and classifies applicable output regions generated on the GPU into three states:

- *must copy-out* regions are those that are immediately followed by a rule that executes on the CPU in the current schedule. For these regions, data is copied eagerly.

- *reused* regions are those immediately followed by another rule on the GPU. For these regions the data is left in GPU memory between rule applications.

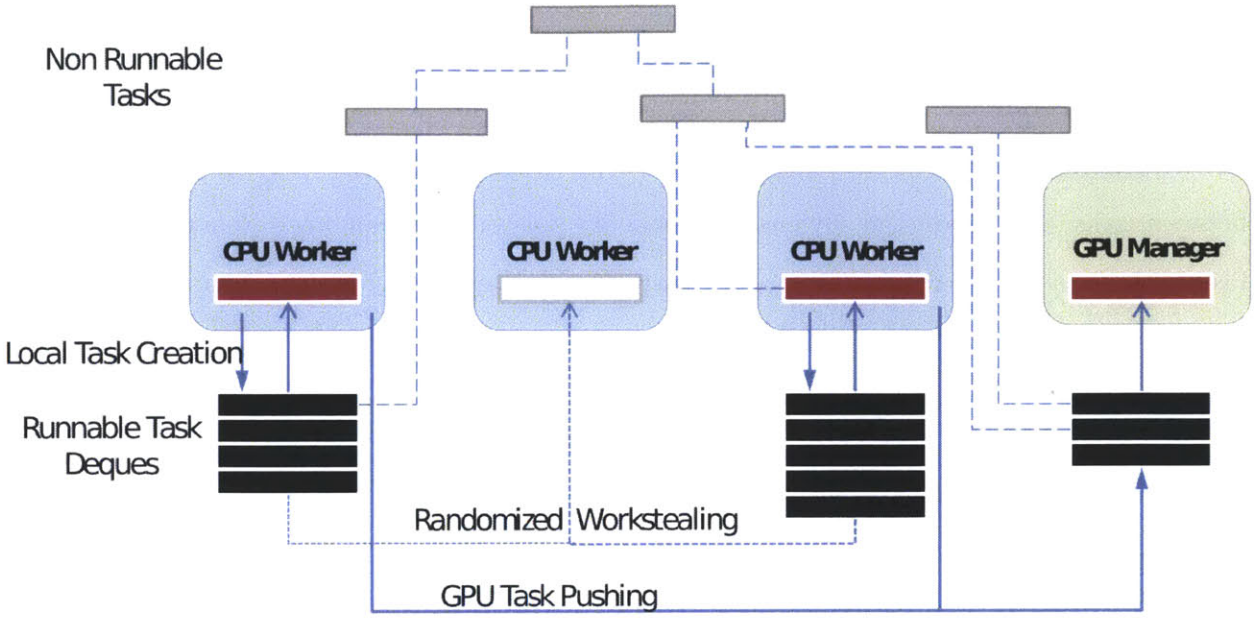


Figure 3.6: Overview of the runtime system. Worker threads use workstealing among themselves, and the GPU management thread only receives works that are pushed by the workers. Tasks exist in both a *runnable* and *non-runnable* state. Runnable tasks exist in deques of different threads, while non-runnable tasks are accessible only through dependency pointers in other tasks.

- *may copy-out* are regions followed by dynamic control flow which we cannot analyze statically. For these regions, we employ a lazy copy-out strategy. A check is inserted before any code that may consume one of these regions to ensure that the required data is in CPU memory. If it is not, then the copy-out is performed when the data is requested.

Depending on the result of this analysis, tasks to prepare, copy-in, execute, and check copy-out status are inserted into the schedule by the compiler.

3.5.3 Runtime System

This section describes the runtime tasks, the *workstealing* model for CPU tasks, the *work-pushing* model for GPU tasks, and the integration of the two models. Figure 3.6 presents an overview of the different parts of the system that will be described in this section.

Task Model for GPUs

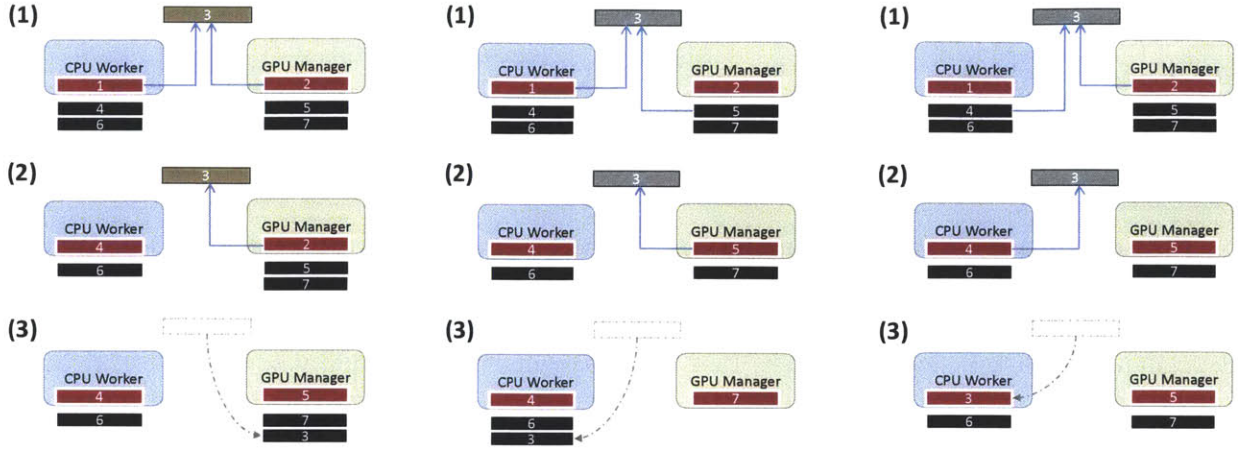
The PetaBricks runtime uses a workstealing mechanism similar to the one introduced in Cilk [67], however, unlike Cilk, the PetaBricks task model supports arbitrary (non-cyclic) dependency graphs between tasks. To support this, dynamic dependency pointers are maintained from each incomplete task to those tasks that depend on it. When a task completes, it may return a continuation task to which all of its dependents are forwarded. Since task dependencies for dependent tasks are created in parallel to a task being executed, some care must be taken when managing tasks. Each task has a state, a count of dependencies, and a list of pointers to dependent tasks. A task can be in one of 5 states:

- *new task* is the initial state for a task. Dependencies may only be added to a task while it is in the new state, and those tasks that the new task depends on have to be tasks not already in the *complete* state. Adding a dependency to a task atomically increments the task's dependency count and adds the task to the appropriate dependents lists, following any needed continuation pointers. When dependency creation is finished, the task transitions to a *runnable task* if its dependency count is zero, otherwise to a *non-runnable task*.

- *non-runnable task* is the state for tasks whose dependency count is greater than zero. These tasks are waiting for other tasks to complete. Non-runnable tasks are stored only in the dependents lists of other tasks and are not stored in any central or thread-specific location. The task completion that eventually decrements the dependency count from one to zero is responsible for enqueuing the task in its thread-local deque.

- *runnable task* is the state for tasks that have zero dependencies and can be executed. These tasks are either being executed or are in exactly one thread-local deque of tasks. Runable tasks can be stolen. When executed, if the task returns a continuation task, it transitions into the *continued* state, otherwise, it transitions to the *complete* state.

- *complete task* is the state for tasks that have already been executed and did not result in a continuation task. When a task becomes complete, it decrements the dependency count of



(a) A GPU task is always pushed to the bottom of the GPU management thread's queue.

(b) The GPU management thread pushes a CPU task to the bottom of a random CPU worker's deque.

(c) A CPU worker pushes a CPU task to the top of its own deque.

Figure 3.7: Three different cases when a task become runnable. The events progress from (1) to (3). Green tasks are GPU non-runnable tasks. Grey tasks are CPU non-runnable tasks.

each of its dependents and enqueues any dependent that becomes runnable. The dependents list is then cleared. Any subsequent attempt to depend on this task results in a no-op.

- *continued task* is the state for tasks that have been executed and returned a continuation. When a task enters the continued state a pointer is stored to the continuation task and the dependents list is transferred to the continuation task. Subsequent attempts to depend on this task instead depend on the continuation task (and possibly, recursively, a continuation of the continuation task).

GPU Management Thread and GPU Tasks

In this highly decentralized model, managing a GPU accelerator presents some challenges. First, for performance, it is important to overlap data transfer operations with computation. Second, one does not want to have many worker threads blocking and waiting for GPU operations to complete.

To address these challenges, we add a dedicated GPU management thread that is responsible for keeping track of all data that resides in GPU memory and schedule operations on the GPU. It operates using the same task representation as CPU threads', allowing dependencies between GPU

and CPU tasks. A tasks is marked as either GPU or CPU task. We use workstealing scheme to manage CPU tasks, but work-pushing scheme to handle GPU tasks. CPU worker threads' deques can only contain CPU tasks, and the GPU management thread's FIFO queue can only contain GPU tasks.

Figure 3.7 depicts what happens when a task becomes runnable in different scenarios. We define the term *cause* as follow. Task A *causes* task B to become runnable when task A satisfies the last dependency of task B; more precisely, task A decrements the dependency count of task B to zero. When a GPU task becomes runnable, it is pushed to the bottom of the GPU management thread's queue as shown in Figure 3.7(a). When a GPU task causes a CPU task to become runnable, the GPU management thread chooses a random CPU worker and pushes the task to the bottom of that thread's deque as shown in Figure 3.7(b). When a CPU task causes another CPU task to become runnable, the CPU worker that runs the former task pushes the newly runnable task to the top of its own local deque as shown in Figure 3.7(c).

There are four classes of GPU tasks that are run by the GPU management thread. For each execution of a GPU kernel, there are one *prepare* task, zero or more *copy-in* tasks, one *execute* task, and zero or more *copy-out completion* tasks.

- *Prepare* tasks allocate buffers on the GPU, and update metadata for GPU execution.

- *Copy-in* tasks copy the required input data to the GPU. One copy-in task is responsible for one input. This task performs a non-blocking write to a GPU buffer and becomes a *complete* immediately after the call, so the GPU manager thread does not have to wait and can execute the next task in its queue right away.

- *Execute* tasks initiate the asynchronous execution of the kernel, perform non-blocking reads from GPU buffers to must copy-out regions, and put may copy-out regions into *pending storage.*

- *Copy-out completion* tasks check the status of the non-blocking reads called by the execute task. If the status of a read is complete, the copy-out completion task corresponding to that

output data transitions to the *complete* state. If the read is not complete, the GPU manager thread pushes the task back to the end of its queue.

There is no dependency between GPU tasks because the GPU management thread only runs one task at a time; the GPU tasks associated to one GPU kernel execution only need to be enqueued by following order: prepare, copy-in, execute, and copy-out completion, to ensure the correctness. However, CPU tasks may depend on GPU copy-out completion tasks.

3.5.4 Memory Management

GPU memory is allocated and managed by the GPU management thread. The GPU management thread keeps a table of information about data stored in the GPU. Each region stored on the GPU can either be a copy of a region of a matrix in main memory or an output buffer for newly computed data that must eventually be copied back to main memory. The GPU management thread is responsible for releasing buffers that become stale because the copy in main memory has been written to and for copying data back to main memory when the data is needed or when it has been flagged for eager copy-out.

The memory management process includes various optimizations to minimize data transfer between the GPU and the CPU. Before we further explain our implementation, the term *matrix* and *region* should be clarified. A matrix is an input or an output of a transform, and is an n-dimensional dense array of elements. A region is a part of a matrix, defined by a start coordinate and size that is an input or an output of a rule.

Copy-in Management Before the GPU management thread executes a *copy-in* task, it will check whether the data is already on the GPU. It can already be on the GPU either if it was copied in for another task execution or if it was the output of another task on the GPU. If all data that will be copied in by the task is already on the GPU, then the GPU management thread will change the status of that *copy-in* task to *complete* without actually executing the task, otherwise it will perform the required copy.

Copy-out Management A single output matrix might be generated by multiple rules in a transform. For example, one task might generate the interior elements of a matrix and other tasks will generate the edges and corners. Instead of creating multiple small buffers on the GPU for these multiple rule outputs, it is often more efficient to create one big buffer for the entire matrix and have the individual tasks output to regions of this buffer. Creating one buffer will take less time than creating multiple buffers because there is an overhead associated with each read from GPU memory, and copying out multiple regions individually requires extra copying to consolidate the data back to the original buffer. It is also possible that some regions of a matrix will be generated on the CPU while others will be generated on the GPU. Our code generation and memory management needs to handle these more complex cases to keep track of which regions of a matrix have been written.

In order to apply this optimization, we use a static analysis for each possible schedule to determine which regions of the matrix are generated on the GPU. The GPU management thread will wait until all of the individual regions have been computed to change the state of the larger matrix.

CPU-GPU Work Balancing If a rule can be run on GPU, choices of how much data should be computed on the GPU are presented to our autotuner. At the extremes, the entire rule could computed on the GPU, or the entire rule could be computed on the CPU. We implement the CPU-GPU work balancing feature by having the autotuner pick a CPU/GPU ratio that defines how much of the data should be computed on each device. This ratio can be set in fixed 1/8th increments. The GPU-CPU ratio shared among all the rules in the same transform because they compute the same matrix but different regions. The system divides the matrix so that the first part is on GPU and the second part is on CPU.

3.5.5 GPU Choice Representation to the Autotuner

The compiler exposes four classes of GPU choices to the autotuner. First, there is the decision of if and when to use the GPU. This is encoded as an algorithmic choice in the selectors constructed by the autotuner. The autotuner can construct selectors that use the GPU for some input sizes and not

others. The autotuner can also construct poly-algorithms that run some parts of the computation on the GPU and other parts on the CPU.

The second type of choice is memory mapping from PetaBricks code to OpenCL. The choice indicates whether or not to use the local memory of the device when possible. This choice exists only if the OpenCL kernel with local memory version of a rule is generated. This is also mapped to an algorithmic choice using a selector constructed by to autotuner.

The third type is the number of work-items in the work-groups (or so called local work size) of each OpenCL kernel, since assigning the right local work size is a common optimization for GPU computing. These parameters are mapped to tunable values that the autotuner can explore independently of algorithmic choices.

The final one is GPU-CPU workload ratio of each transform, which defines what percentage of a region should be computed on the GPU. To limit the search space size, the possible ratios restricted to multiples of 1/8.

In a big picture, every transform provides 12 levels of algorithmic choices for 12 different ranges of input sizes. Note that all levels of the same transform have the same number of algorithmic choices. For example, in SeparableConvolution configuration, each level of each `Convolve*` transform has three possible choices: using CPU backend, using OpenCL backend with global memory only, and using OpenCL backend with local memory optimization. Each `Convolve*` has two OpenCL kernels, so each of them has its own tunable parameters for local work size and GPU-CPU workload ratio. Apart from OpenCL related parameters, the configuration also includes other parameters such as split size for CPU work-stealing model, number of execution threads on CPU, and a sequential/parallel cutoff.

Challenges with Runtime Kernel Compilation

The PetaBricks autotuning approach runs large numbers of tests on small input sizes in order to quickly explore the choice space and seed exploration for larger input sizes. With only the CPU, these tests on small input sizes are very cheap and greatly improve convergence time. However, with our OpenCL backend these small tests take a much larger amount of time, because OpenCL

kernels are compiled dynamically at runtime. These kernel compiles represent a fixed startup cost, often on the order of a few seconds, that dominate execution time for small input sizes. This factor increases autotuning time.

To address this problem we use two techniques. First, we cache the intermediate representation (IR) used by the OpenCL compiler. The first time we run a program, we store the OpenCL runtime-specific IR for each compiled kernel with the hash of the source for that kernel. This IR is reused in subsequent executions in order to skip the parsing and optimization phases of kernel compilation. Second, we adjusted the parameters of the autotuner so that it runs fewer tests at small input sizes. This involved both skipping extremely small input sizes entirely and running fewer tests on the smaller input sizes we do use. The result of these optimizations reduced typical training times from many days for some benchmarks to an average of 5.2 hours. This training time is still heavily influenced by architecture-specific JITing, which OpenCL does not allow to be cached. Full binary caching, as allowed by other languages such as CUDA, would further reduce training times.

3.6 Choice Space Representation

We will now discuss two important building blocks that will be required by the autotuner (described in Chapter 6), *choice configuration files* and *mutator functions*, before discussing the individual phases of the autotuner.

3.6.1 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.

- **Synthesized functions**. A function from a dynamic input size to a value. For example, how many iterations in a `for_enough` loop, which may be different depending on the size of data the loop is operating on. The autotuner sets a fixed number of values of this function and other values are filled in through interpolation.

- **User defined parameters**.

These *choice configuration files* will be used heavily by the PetaBricks autotuner which will be described in Chapter 6.

3.7 Deadlocks and Race Conditions

Another typical problem in hand written parallel code is deadlocks. Deadlocks cannot occur in PetaBricks because the program's dependency graph is fully analyzed at compile time. Potential deadlocks manifest themselves as a cycle in the graph, and the PetaBricks compiler detects this cycle and reports an error to user. This deadlock freedom guarantee, when using only the core PetaBricks language, is a great advantage. When external code, written in other languages, is called from PetaBricks, it is the programmers responsibility to ensure that the program executes without deadlocks.

Similar to deadlocks, race conditions cannot exist in PetaBricks, except when caused by externally called code written in other languages. Since PetaBricks is implicitly parallel, the programmer cannot manually specify that two operations should run in parallel. Instead, analysis is performed by the compiler and tasks that do not depend on each other are automatically parallelized. If a race condition were to exist, then the compiler would see that dependency edge and not run the two tasks in parallel.

3.8 Automated Consistency Checking

A side benefit of having multiple implementations of algorithms for solving the same problem is that the compiler can check these algorithms against each other to make sure they produce consistent results. This helps the user to automatically detect bugs and increase confidence in code correctness. This automated checking makes it advisable to include a slow reference implementation as a choice so that faster choices can be checked against it.

This consistency checking happens during autotuning when a special flag is set. The autotuner, by design, is already exploring the space of possible algorithms to find one that performs the best. The consistency checking merely uses a fixed input during each autotuning round and ensures that the same output is produced by every candidate algorithm. While not provably correct, this technique provides good testing coverage. Notably, this technique also focuses more testing on the candidate algorithms that are actually used as the autotuner hones in on an optimal choice. Some of our benchmarks use iterative approaches that do not produce exact answers. To support such code, our automated checker takes a threshold argument where differences below that threshold are ignored.

Chapter 4

Benchmarks and Experimental Analysis

In this section, we describe a set of benchmarks we implemented to illustrate the capabilities of the PetaBricks compiler. The benchmarks were chosen to be relevant, widely applicable scientific and computing kernels: solving Poisson's equation, the symmetric tridiagonal eigenvalue problem, sorting, and dense matrix multiply. The Poisson's equation solver benchmark is described in detail in Chapter 5.

The benchmarks described in this chapter will demonstrate the benefits of algorithmic choice. We will show how different algorithms are required to get performance on different systems, and show how PetaBricks programs can adapt to fit their environment.

Results were gathered on a 8-way (dual socket, quad core) Intel Xeon E7340 system running at 2.4 GHz. The system was running 64 bit CSAIL Debian 4.0 with Linux kernel 2.6.18 and GCC 4.1.2.

4.1 Fixed Accuracy Benchmarks

4.1.1 Symmetric Eigenproblem

The symmetric eigenproblem is another problem with broad applications in areas such as mechanics, quantum physics and structural engineering. Given a symmetric $n \times n$ matrix, we want to find its eigenvalues and/or eigenvectors. Deciding on which algorithms to use depends on how many eigenvalues to find and whether eigenvectors are needed. Here we study the problem in which all the eigenvalues and eigenvectors are computed.

Algorithms and Choices

To find all the eigenvalues and eigenvectors of a symmetric matrix, we examine the use of three primary algorithms, QR iteration, Bisection and inverse iteration, and Divide-and-conquer. The input matrix A is first reduced to $A = QTQ^T$, where Q is orthogonal and T is symmetric tridiagonal. All the eigenvalues and eigenvectors of T are then computed by the algorithm chosen. The eigenvalues of A and T are equal. The eigenvectors of A are obtained by multiplying Q by the eigenvectors of T. The total work needed is $O(n^3)$ for reduction of the input matrix and transforming the eigenvectors, and the cost associated with each algorithm [52].

The QR iteration applies the QR decomposition iteratively until T converges to a diagonal matrix. It computes all the eigenvalues and eigenvectors in $O(n^3)$ operations.

Bisection, followed by inverse iteration, finds k eigenvalues and the corresponding eigenvectors in $O(nk^2)$ operations, resulting in a complexity of $O(n^3)$ for finding all eigenvalues and eigenvectors. This algorithm is based on a simple formula to count the number of eigenvalues less than a given value. Each eigenvalue and eigenvector thus can be computed independently, making the algorithm "embarrassingly parallel".

The eigenproblem of tridiagonal T can also be solved by a divide-and-conquer approach. The eigenvalues and eigenvectors of T can be computed using the eigenvalues and eigenvectors of two smaller tridiagonal matrices, and this can be done recursively. Divide-and-conquer requires $O(n^3)$ work in the worst case.

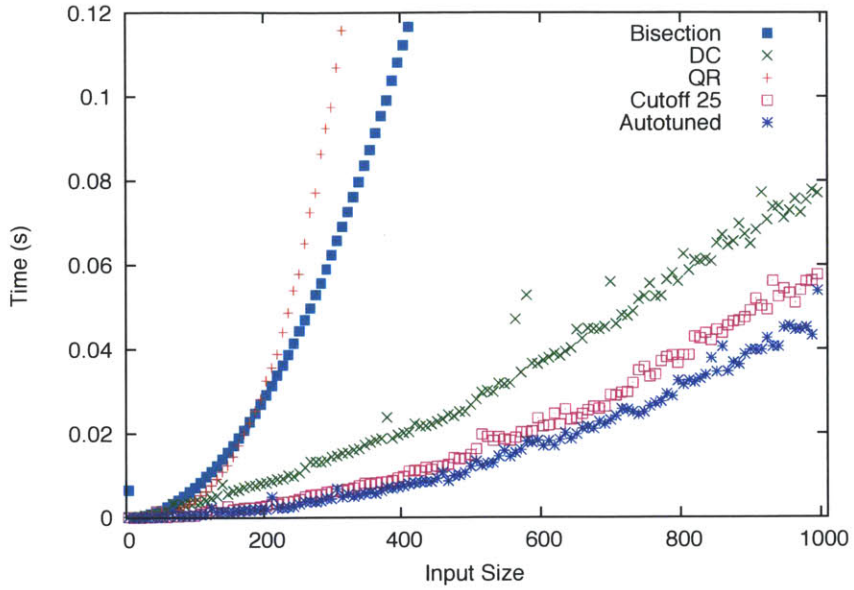


Figure 4.1: Performance for Eigenproblem on 8 cores. "Cutoff 25" corresponds to the hard-coded hybrid algorithm found in LAPACK.

`EIG`(T)

1: **either**
2: Use `QR` to find Λ and X
3: Use `BISECTION` to find Λ and X
4: Recursively call `EIG` on submatrices T_1 and T_2 to get Λ_1, X_1, Λ_2 and X_2. Use results to compute Λ and X.
5: **end either**

Figure 4.2: Pseudo code for eigenvector solve.

The PetaBricks transforms for these three primary algorithms are implemented using LAPACK routines, as is MATLAB polyalgorithm `eig`. Our optimized hybrid PetaBricks algorithm computes the eigenvalues Λ and eigenvectors X by automating choices of these three basic algorithms. The pseudo code for this is shown in Figure 4.2. There are three algorithmic choices, two non-recursive and one recursive. The two non-recursive choices are QR iterations, or bisection followed by inverse iteration. Alternatively, recursive calls can be made. At the recursive call, the PetaBricks compiler will decide the next choices, i.e. whether to continue making recursive calls or switch to one of the non-recursive algorithms. Thus the PetaBricks compiler chooses the optimal cutoff for the base case if the recursive choice is made. After autotuning, the best algorithm choice was found to be

divide-and-conquer for matrices larger than 48, and switching to QR iterations when the size of matrix $n \leq 48$.

Performance

We implemented and compared the performance of five algorithms in PetaBricks: QR iterations, bisection and inverse iteration, divide-and-conquer with base case $n = 1$, divide-and-conquer algorithm with hard-coded cutoff at $n = 25$, and our autotuned hybrid algorithm. In figure 4.1, these are labelled QR, Bisection, DC, Cutoff 25 and Autotuned respectively. The input matrices tested were random symmetric tridiagonal. Our autotuned algorithm runs faster than any of the three primary algorithms alone (QR, Bisection and DC). It is also faster than the divide-and-conquer strategy which switches to QR iteration for $n \leq 25$, which is the underlying algorithm of the LAPACK routine `dstevd` [6].

4.1.2 Sort

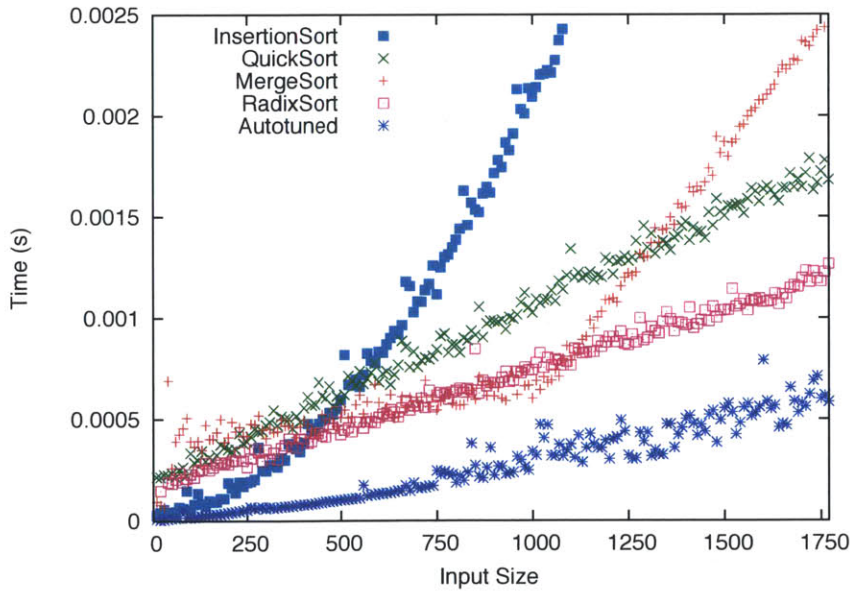


Figure 4.3: Performance for sort on 8 cores.

For the problem of sorting, we implemented the following algorithms in PetaBricks: insertion sort; quick sort; n-way merge sort (when n equals 2, merge sort employs a recursive merge routine

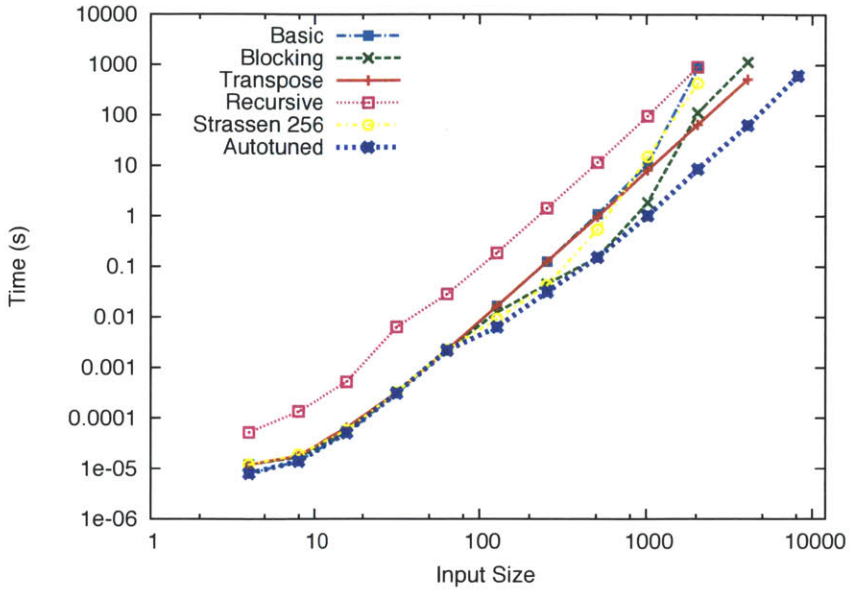


Figure 4.4: Performance for Matrix Multiply on an 8 cores. "Strassen 256" uses strassen algorithm to decompose until n=256 when it switches to basic matrix multiply.

that can also be parallelized), where the compiler can select n; and a 16 bucket radix sort (a MSD variant that can recursively call any sorting algorithm). All of the algorithms are recursive except for insertion sort. Each of these algorithms recursively calls a generalized `sort` transform, which allows the compiler to switch algorithms at any level.

Figure 4.3 shows the performance for sort on 8 cores. Our autotuner was able to achieve significant performance improvements over any single algorithm. Surprisingly, the autotuned composite algorithm did not utilize radix sort, despite it being the second fastest algorithm. Instead, it built a hybrid algorithm using first 2-way merge sort, followed by quicksort, followed by a call to insertion sort for smaller inputs. The sharp bend in performance for merge sort occurs at 1024 where the binary tree of merges grows from 10 to 11 levels. If the graph is extended to larger inputs, merge sort's performance forms a step ladder. When merge sort is used in a autotuned hybrid algorithm this step ladder performance pattern disappears.

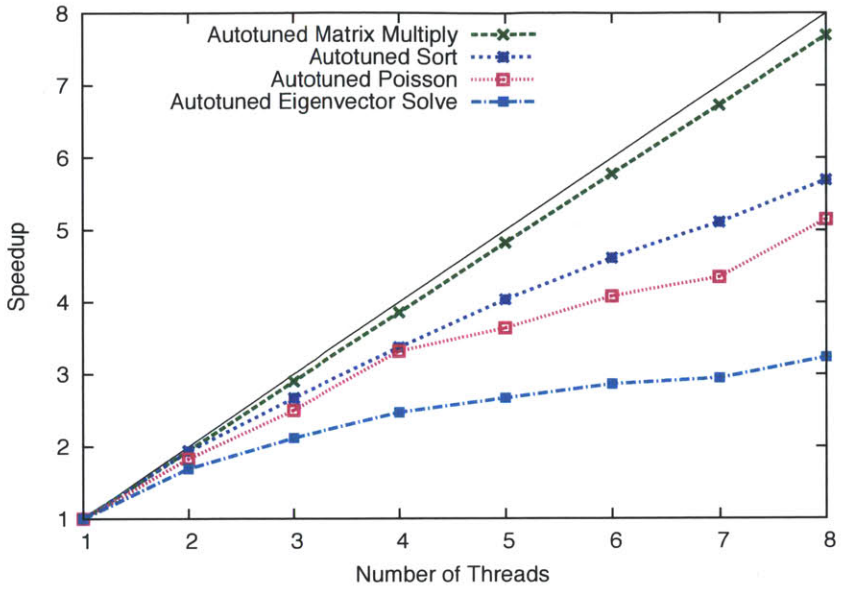


Figure 4.5: **Parallel scalability.** Speedup as more worker threads are added. Run on an 8-way (2 processor × 4 core) x86_64 Intel Xeon System.

4.1.3 Matrix Multiply

The PetaBricks code for matrix multiply can be found in the introduction (Figure 2.3). The compiler also considers non-algorithmic choices including: transposing each of the inputs; various blocking strategies; and various parallelization strategies. For matrix multiply, these non algorithmic choices make a huge impact.

Figure 4.4 shows performance for various versions of matrix multiply. Since the non-algorithmic optimizations (blocking and transposing) made a large difference performance of those optimizations are also shown. The series labeled "Recursive" is the recursive decomposition in the "c". The autotuned algorithm uses a mixture of blocking, transpose, and the recursive decomposition.

4.2 Autotuning Parallel Performance

A great advantage of PetaBricks is that it allows a single program to be optimized for both sequential performance and parallel performance. We have observed our autotuner make different choices when training in parallel. As a general trend we noticed much lower cutoffs to bases cases in sequential

programs. In many cases entirely different algorithms are chosen. Of particular note is the fact that algorithms tuned on 8 cores scale much better than algorithms tuned on 1 core.

As an example, when tuning `sort` on 1 core our autotuner picks radix sort with a cutoff of 98 where it switches to 4-way merge sort after which it finishes with insertion sort at a cutoff of 75. When tuned using 8 cores the autotuner decides to use the 2-way-merge sort (with a parallelizable recursive merge) function until the input is smaller than 1420, after which it switches to quick sort. Finally, at inputs smaller than 600, it switches to insertion sort. When both algorithms are run using 8 cores, the algorithm tuned on 8 cores performs 2.14x faster than the algorithms tuned on 1 core (as seen in Table 4.1).

4.3 Effect of Architecture on Autotuning

		Trained on			
		Mobile	Xeon 1-way	Xeon 8-way	Niagara
Run on	Mobile	-	1.09x	1.67x	1.47x
	Xeon 1-way	1.61x	-	2.08x	2.50x
	Xeon 8-way	1.59x	2.14x	-	2.35x
	Niagara	1.12x	1.51x	1.08x	-

Table 4.1: Slowdown when trained on a setup different than the one run on. Benchmark is sort on an input size of 100,000. Slowdowns are relative to training natively. Descriptions of abbreviated system names can be found in Table 4.2.

Abbreviation	System	Frequency	Cores used	Scalability	Algorithm Choices (w/ switching points)
Mobile	Core 2 Duo Mobile	1.6 GHz	2 of 2	1.92	IS(150) 8MS(600) 4MS(1295) 2MS(38400) QS(∞)
Xeon 1-way	Xeon E7340 (2 x 4 core)	2.4 GHz	1 of 8	-	IS(75) 4MS(98) RS(∞)
Xeon 8-way	Xeon E7340 (2 x 4 core)	2.4 GHz	8 of 8	5.69	IS(600) QS(1420) 2MS(∞)
Niagara	Sun Fire T200 Niagara	1.2 GHz	8 of 8	7.79	16MS(75) 8MS(1461) 4MS(2400) 2MS(∞)

Table 4.2: Automatically tuned configuration settings for the sort benchmark on various architectures. We use the following abbreviations for algorithm choices: IS = insertion sort; QS = quick sort; RS = radix sort; 16MS = 16-way merge sort; 8MS = 8-way merge sort; 4MS = 4-way merge sort; and 2MS = 2-way merge sort, with recursive merge that can be parallelized. The cutoff point to switch to each algorithm is shown within parentheses.

Multicore architectures have drastically increased the processor design space resulting in a large variety of processor designs currently on the market. Such variance significantly hinders porting efforts of performance critical code. In this section, we present the results of PetaBricks

autotuner when optimizing our sort benchmark on three parallel architectures designed for a variety of purposes: Intel Core 2 Due mobile processor, Intel Xeon E7340 server processor, and the Sun Fire T200 Niagara low power high throughput server processor.

Table 4.1 illustrates the necessity of tuning your program for the architecture that you plan to run on. When autotuning our sort benchmark, we found that configurations trained on a different setup than they are run on exhibit significant slowdowns. For example, even though they have the same number of cores, the autotuned configuration file from the Niagara machine results in a 2.35x loss of performance when used on the Xeon processor. On average we observed a slowdown of 1.68x across all of the systems we tested.

Table 4.2 displays the optimal configurations for the sort benchmark after running the same autotuning process on the three architectures. It is interesting to note the dramatic differences between the choice of algorithms, composition switching points, and scalability. The Intel architectures (with larger computation to communication ratios) appear to perform better when PetaBricks produces code with less parallelism, suggesting that the cost of communication often outweighs any benefits from running code containing fine-grained parallelism. On the other hand, the Sun Niagara processor performs best when executing code with lots of parallelism as shown by the exclusive use of recursive algorithms.

4.4 Variable Accuracy Benchmarks

In this section, we describe a number variable accuracy benchmarks that we implemented to test the efficacy variable accuracy algorithms in PetaBricks. The Poisson and Helmholtz benchmarks are described in detail in Chapter 5.

Many variable accuracy benchmarks are input sensitive. For these experiments we test and train on (different) random inputs taken from a single distribution. Chapter 7 will revisit many of these benchmarks in the context of input sensitivity and discuss how to classify and adapt to different types of program inputs.

4.4.1 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 provides 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 13 well known bin packing algorithms:

- **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 [83].
- **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**.

For more information on bin packing see [50].

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.

4.4.2 Clustering

Clustering divides a set of data into clusters based on similarity. The problem of k-clustering is NP-hard when k is fixed. Clustering is a common technique for statistical data analysis in areas including machine learning, pattern recognition, image segmentation and computational biology. We implemented a variant of Lloyd's algorithm for k-means clustering. Our implementation gives the autotuner multiple choices for creating initial assignments of clusters, including random and k-means++ [19].

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 [19], 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.

Rather than assigning a fixed k through a heuristic (such as the commonly used $k = \sqrt{n}$), we define k as an accuracy variable and allow the autotuner to set it. This allows the number of clusters to change based on how compact clusters the user of the algorithm requests through the accuracy requirement. This implicitly ties the tuned algorithm to the distribution of inputs being run on. We explore adapting to specific program inputs in Chapter 7.

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.

4.4.3 Image Compression

Our image compression benchmark performs Singular Value Decomposition (SVD) on an $m \times n$ matrix. SVD is major component in some image compression algorithms [152]. For any $m \times n$ real matrix A with $m \geq n$, the SVD of A is $A = U\Sigma V^T$. The columns u_i of the matrix U, the columns v_i of V, and the diagonal values σ_i of Σ (singular values) form the best rank-k approximation of A, given by $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$. 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.

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. The transform for matrix approximation consists of a hybrid algorithm for finding all eigenvalues and eigenvectors, which

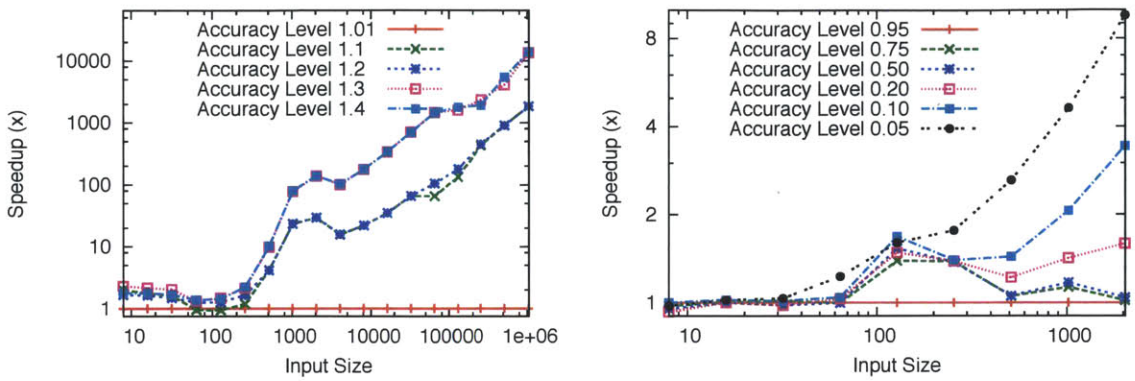
combines Divide and Conquer, QR Iteration, and Bisection method. Another algorithmic choice exposed is using the Bisection method for only k eigenvalues and eigenvectors. The accuracy metric used is the ratio between the RMS error of the initial guess (the zero matrix) to the RMS error of the output compared with the input matrix A, converted to log-scale.

4.4.4 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.

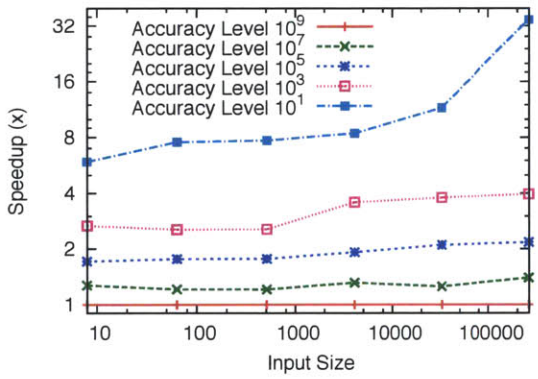
The convergence of a matrix iteration depends on the properties of the matrix A, one of which is called the condition number. A preconditioner P of a matrix A is a matrix that if well chosen, the condition number of $P^{-1}A$ is smaller than that of A. Solving the preconditioned system $P^{-1}Ax = P^{-1}b$ gives the same solution, but the rate of convergence improves. 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. 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.

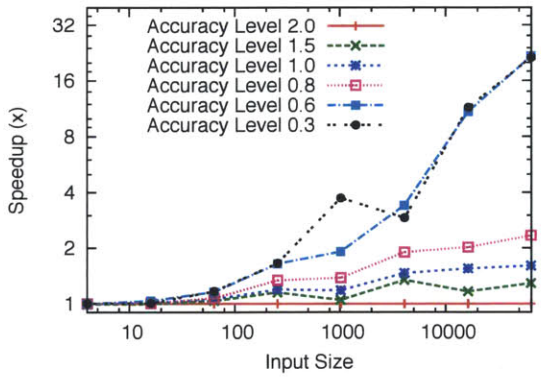


(a) Bin Packing

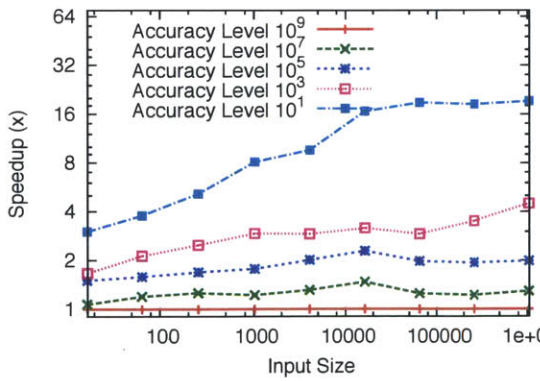
(b) Clustering



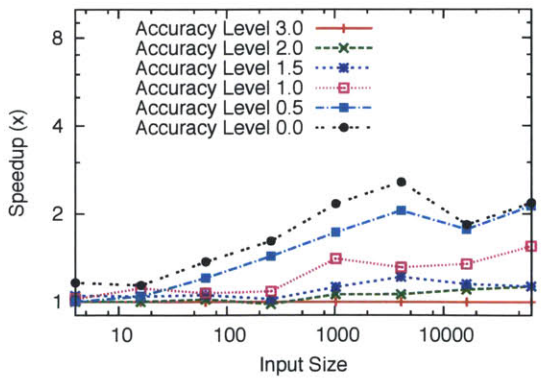
(c) Helmholtz



(d) Image Compression



(e) Poisson



(f) Preconditioner

Figure 4.6: 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.

4.5 Experimental Results

Figures 4.6(a)-4.6(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.

4.5.1 Analysis

This section provides more detailed analysis of the impact of accuracy on algorithmic choice and of programmability. We observed similar behaviours for the other benchmarks. Further analysis of many of these benchmarks is provided in Chapter 7 in the context of adapting these benchmarks to different inputs.

Bin Packing Figure 4.7 depicts the results of autotuning the Bin Packing benchmark for various desired accuracy levels (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

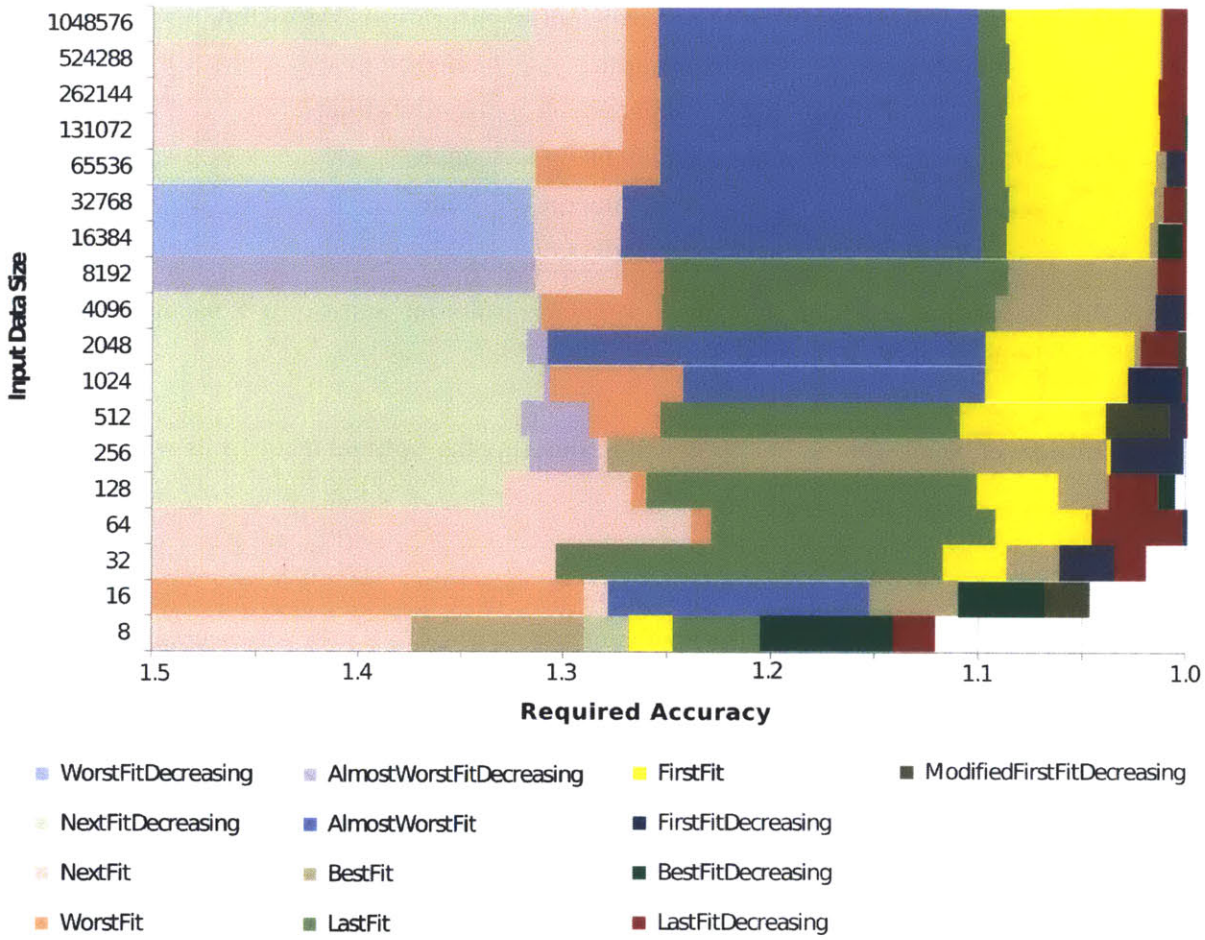


Figure 4.7: Best algorithm for each accuracy and input size in the Bin Packing benchmark. By best we mean on the optimal frontier (there exists no algorithm with better performance and accuracy for a given input size on average). Accuracy is defined as the number of bins over the optimal number of bins achievable. Lower numbers represents a higher accuracy.

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.

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×) of optimal, it is never the best performing algorithm when a probabilistic bound of worse than 1.07× accuracy is desired. This result highlights

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

Figure 4.8: Algorithm selection and initial k value results for autotuned k-means benchmark for various accuracy levels with n=2048 and k_source= 45

the advantages of using a empirical approach to determining optimal algorithms when probabilistic guarantees on accuracy are permissible.

Clustering Figure 4.8 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 $(\sqrt{\frac{2n}{\sum D_i^2}})$ 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,

4.5.2 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.

The amount of time spent training is heavily dependent on both the configuration used for the autotuner and the time complexity of the application being tuned. The configuration used for the autotuner determines the number of times the program will be run to measure performance and accuracy metrics for different parameter values. A larger number of runs will be chosen automatically in cases where the timing or accuracy of the result has larger variance – this is highly benchmark dependent. Total training times for the various benchmark configurations ranged from

fifteen minutes to two hours per benchmark, where the longest training times resulted from running our most complex benchmarks on very large inputs.

4.6 Heterogeneous Architectures Experimental Results

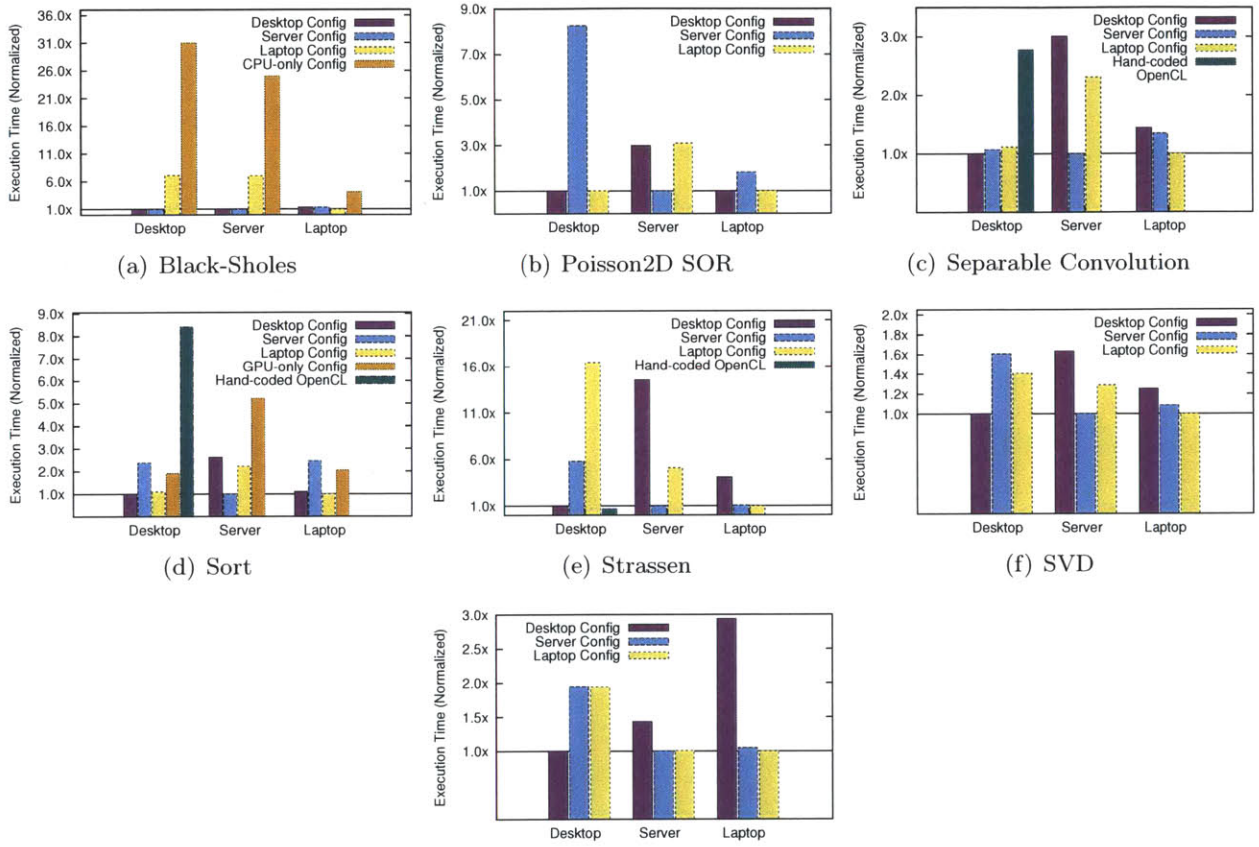
This section of experimental results explores the extent to which different heterogeneous systems require different configurations to obtain optimal performance. To this end, our experiments test how configurations tuned for one heterogeneous system perform when run on a different system. We examine these differences both by testing relative performance and by examining the configurations and algorithmic choices of these tuned configurations.

	Desktop Config	**Server Config**	**Laptop Config**
Black-Sholes	100% on GPU	100% on OpenCL	Concurrently 25% on CPU and 75% on GPU
Poisson2D SOR	Split on CPU followed by compute on GPU	Split some parts on OpenCL followed by compute on CPU	Split on CPU followed by compute on GPU
SeparableConv.	1D kernel+local memory on GPU	1D kernel on OpenCL	2D kernel+local memory on GPU
Sort	Polyalgorithm: above 174762 2MS (PM), then QS until 64294, then 4MS until 341, then IS on CPU[1]	Polyalgorithm: above 7622 4MS, then 2MS until 2730, then IS on CPU[1]	Polyalgorithm: above 76830 4MS (PM), then 2MS until 8801 (above 34266 PM), then MS4 until 226, then IS on CPU[1]
Strassen	Data parallel on GPU	8-way parallel recursive decomposition on CPU, call LAPACK when $< 682 \times 682$	Directly call LAPACK on CPU
SVD	First phase: task parallism between CPU/GPU; matrix multiply: 8-way parallel recursive decomposition on CPU, call LAPACK when $< 42 \times 42$	First phase: all on CPU; matrix multiply: 8-way parallel recursive decomposition on CPU, call LAPACK when $< 170 \times 170$	First phase: all on CPU; matrix multiply: 4-way parallel recursive decomposition on CPU, call LAPACK when $< 85 \times 85$
Tridiagonal Solver	Cyclic reduction on GPU	Direct solve on CPU	Direct solve on CPU

Figure 4.9: Summary of the different autotuned configurations for each benchmark, focusing on the primary differences between the configurations. [1]For sort we use the abbreviations: IS = insertion sort, 2MS = 2-way mergesort, 4MS = 4-way mergesort, QS = quicksort, PM = with parallel merge.

4.6.1 Methodology

Figure 4.12 lists the representative test systems used in our experiments and assigns code names that will be used in the remainder of the section. We chose these machines to represent three diverse



(a) Black-Sholes

(b) Poisson2D SOR

(c) Separable Convolution

(d) Sort

(e) Strassen

(f) SVD

(g) Tridiagonal Solver

Figure 4.10: Benchmark performance when varying the machine and the program configuration. Execution time on each machine is normalized to the natively autotuned configuration. Lower is better. 4.10(c), 4.10(d), and 4.10(e) include *Hand-coded OpenCL* as a baseline taken from the NVIDIA SDK sample code. This baseline uses NVIDIA-specific constructs and only runs on our Desktop system. These hand-coded OpenCL baselines implement 1D separable convolution, radix sort, and matrix multiply respectively. As additional baselines, 4.10(b) includes a *CPU-only Config* which uses a configuration autotuned with OpenCL choices disabled and 4.10(d) includes *GPU-only Config* which uses PetaBricks bitonic sort on the GPU.

Name	# Possible Configs without OpenCL	# Possible Configs	Generated OpenCL Kernels	Mean Autotuning Time	Testing Input Size
Black-Sholes	10^{21}	10^{130}	1	3.09 hours	500000
Poisson2D SOR	10^{133}	10^{1358}	25	15.37 hours	2048^2
SeparableConv.	10^{156}	10^{1358}	9	3.82 hours	3520^2
Sort	10^{267}	10^{920}	7	3.56 hours	2^{20}
Strassen	10^{645}	10^{1509}	9	3.05 hours	1024^2
SVD	10^{1016}	10^{2435}	8	1.79 hours	256^2
Tridiagonal Solver	10^{178}	10^{1040}	8	5.56 hours	1024^2

Figure 4.11: Properties of the benchmarks. The number of configurations are calculated from the parameters described in Section 3.5.5.

Codename	CPU(s)	Cores	GPU	OS	OpenCL Runtime
Desktop	Core i7 920 @2.67GHz	4	NVIDIA Tesla C2070	Debian 5.0 GNU/Linux	CUDA Toolkit 4.2[1]
Server	4× Xeon X7550 @2GHz	32	None	Debian 5.0 GNU/Linux	AMD Accelerated Parallel Processing SDK 2.5[2]
Laptop	Core i5 2520M @2.5GHz	2	AMD Radeon HD 6630M	Mac OS X Lion (10.7.2)	Xcode 4.2[1]

Figure 4.12: Properties of the representative test systems and the code names used to identify them in results. The OpenCL runtimes marked [1] targets the GPUs of the machines, while [2] targets the CPU by generating optimized SSE code.

systems a program may run on. Desktop represents a high-end gaming system with a powerful processor and a graphics card. Server represents a large throughput-oriented multicore system one might find in a data center. It does not have a graphics card, we instead use a CPU OpenCL runtime that creates optimized parallel SSE code from OpenCL kernels. Laptop represents a laptop (it is actually a Mac Mini), with a low-power mobile processor and a graphics card. Together, our test systems cover graphics cards from both AMD and NVIDIA, use three different OpenCL runtimes, have two different operating systems, and range in cores from 2 to 32.

In our experiments we first create three program configurations by autotuning: *Desktop Config* is the configuration tuned on Desktop; *Server Config* is the configuration tuned on Server; and *Laptop Config* is the configuration tuned on Laptop.

Next, we run each of these three configurations on each of our three machines. Since we are migrating configurations between machines with different numbers of processors, for fairness, we remove the thread count parameter from the search space and set the number of threads equal to the number of processors on the current machine being tested. (On Server, the number of threads is set to 16 which provides better performance on every benchmark.)

Finally, for some benchmarks, we also include baseline results for comparison. We do this either by writing a PetaBricks program configuration by hand, by running OpenCL programs included as sample code in the NVIDIA OpenCL SDK, or by running CUDPP applications. We use the label *Hand-coded OpenCL* to indicate the performance of a standalone OpenCL program not using our system. These baselines are described in detail for each benchmark.

Figure 4.11 lists properties of each of our benchmarks from the PetaBricks benchmark suite. The configuration space is large, ranging from 10^{130} to 10^{2435}. Our system automatically creates up to 25 OpenCL kernels per benchmark. Average autotuning time was 5.2 hours. This training time is larger as a result of overheads from OpenCL kernel compilation, which dominate tuning time for many benchmarks.

4.6.2 Benchmark Results and Analysis

Figure 4.9 summarizes the auto tuned configurations for each benchmark and Figure 4.10 shows the performance of each of these configurations on each system. Detailed analysis for each benchmark is provided in this section.

Black-Sholes The Black-Scholes benchmark results (Figure 4.10(a)) show that on some machines it is fastest to divide the data and compute part of the output on the CPU and another part concurrently on the GPU. Black-Scholes implements a mathematical model of a financial market to solve for the price of European options. Each entry of the output matrix can be computed from the input matrix by applying the Black-Scholes formula.

The autotuned Black-Scholes configurations on the Desktop and Server both perform all computation using the GPU/OpenCL backend, while the configuration on the Laptop divides the work such that 25% of the computation is performed on the CPU, and 75% is performed on the

GPU. This 25/75 split provides a 1.3x speedup over using only the GPU on the Laptop, while such a split produces a 7x slowdown on the other two systems.

The reason why these configurations perform this way is the OpenCL performance for Black-Sholes is an order of magnitude better than the CPU performance on the Desktop and Server. However, on the laptop the relative performance of the GPU is only about 4x the performance of the CPU. This can be seen in the *CPU-only Config* bar in Figure 4.10(a), which is a baseline configuration that does not use the GPU/OpenCL backend. On the laptop, where the relative performance of the two processors is close, exploiting heterogeneous parallelism between the CPU and the GPU results in performance gains.

Poisson2D SOR The Poisson2D SOR benchmark (Figure 4.10(b)) shows that the choice of which backend is best for each phase of the computation can change between machines. This benchmark solves Poisson's equation using Red-Black Successive Over-Relaxation (SOR). Before main iteration, the algorithm splits the input matrix into separate buffers of red and black cells for cache efficiency.

In the Desktop and Laptop tuned configuration, this splitting is performed on the CPU and then actual iterations are performed using the OpenCL GPU backed. In the Server configuration, nearly the opposite happens; the OpenCL backend is used for splitting a large area of the matrix, and the CPU backend is used for the main iterations. The reason for this switch is the very different performance profiles of the OpenCL backends. The Desktop and Laptop utilize actual GPUs while the Server OpenCL backend shares the CPU.

Separable Convolution The Separable Convolution benchmark (Figure 4.10(c)) highights the effect of algorithmic choices on the GPU. Three configurations, all using only OpenCL for computation, provide very different performance on each machine. This benchmark is used as a driving example in Section 2.5, which describes the choices benchmark in more detail. At width 7, shown here, Desktop performs best using 1D separable convolution on the GPU with the GPU local memory. Laptop, with the less powerful GPU, performs best using local memory but with the single-pass 2D convolution algorithm, because the overhead of synchronization and creating an

extra buffer to store the intermediate results between the two passes dominates the computational savings of the separable algorithm. The best configuration on Server uses the OpenCL version of separable convolution, but without local memory prefetching, since the CPUs' caches perform best here without explicit prefetches.

The results also show 2.3x better performance than OpenCL baseline implementation taken from the OpenCL samples in the NVIDIA SDK (Figure 4.10(c)). Our implementation differs from this sample code in that in our generated code each work-item computes exactly one entry of the output, while in the sample code each work-item computes multiple entries of the output. This optimization in the sample code not only increases code complexity, but also results in worse performance than our PetaBricks implementation on the Tesla C2070. Performance on these machines is complex and unpredictable, making hard-coded choices often lose to our automatically inferred results.

Sort The Sort benchmark (Figure 4.10(d)) shows that for some tasks it makes sense to run on the CPU. The benchmark includes 7 sorting algorithms: merge sort, parallel merge sort, quick sort, insertion sort, selection sort, radix sort, and bitonic sort; in addition, merge sort and parallel merge sort have choices of dividing a region into two or four subregions. The configuration defines a poly-algorithm that combines these sort building blocks together into a hybrid sorting algorithm

None of the tuned configurations choose to use OpenCL in the main sorting routine (although some helper functions, such as copy, are mapped to OpenCL). The choices in CPU code are complex, and result in up to a 2.6x difference in performance between autotuned configurations. Each configuration is a poly-algorithm that dynamically changes techniques at recursive call sites. Desktop uses 2-way merge sort with parallel merge at the top level, switches to quick sort when sizes of sub-arrays are smaller, switches to 4-way merge sort with sequential merge when sizes are even smaller, and finally ends with insertion sort as a base case when size is less than 341. Server uses 4-way merge sort with sequential merge, switches to 2-way merge sort when sizes are smaller, and finally switches to insertion sort when size is less than 2730. Laptop alternatively switches between 4-way and 2-way merge sort, uses parallel merge until size is less than 34266, and switches to insertion sort when size is less than 226.

For comparison, we wrote a configuration by hand that uses bitonic sort in OpenCL using our system (*GPU-only Config* in Figure 4.10(d)). This configuration is between 1.9 and 5.2x slower than the native autotuned configuration. Interestingly, this configuration does beat the Server configuration on both of the other machines that have GPUs. This means that, if one had the Server configuration, using the GPU instead would give a speedup, but not as much of a speedup as re-tuning on the CPU.

As a second baseline, we include the radix sort sample program from the NVIDIA SDK (*Hand-coded OpenCL* in Figure 4.10(d)). This OpenCL implementation of Sort performs 8.4x worse than our autotuned configuration and 4.4x worse than our bitonic sort configuration. The poor performance of both of these GPU Sort baselines, relative to our autotuned Sort, speak to the difficulty writing a high performance Sort on the GPU. Researchers have developed faster methods for GPU sorting, however, these methods require both an autotuning system and heroic programmer effort [72], and their performance generally does not account for overhead in copying data to and from the GPU, which our benchmarks do.

Strassen The Strassen benchmark (Figure 4.10(e)) shows that choosing the right configuration for each architecture results in very large performance gains. The Strassen benchmark performs a dense matrix-matrix multiplication. The choices include: transposing any combination of the inputs; four different recursive decompositions, including Strassen's algorithm; various blocking methods; naive matrix multiplication; and calling the LAPACK external library.

Figure 4.10(e) shows the performance of our Strassen benchmark with different configurations. Laptop configuration gives a 16.5x slowdown on Desktop. OpenCL is used in the Desktop configuration, and C++/Fortran (through a call to LAPACK) is used in the Server and Laptop configurations. The large computation to communication ratio in matrix multiplication stresses the difference in relative CPU/GPU computational power between Desktop, with a high performance graphics card, and Laptop, with a mobile graphics card. This results in the Desktop GPU producing a speedup compared to its CPU, and the Laptop GPU producing a slowdown.

Altough Server and Laptop both use CPUs, their optimal configurations are different. On Server, the best algorithm is to recursively decompose the matrices in 8-way parallel fashion

until the regions are smaller than a certain size, call LAPACK on the small regions, and finally combine the data. On Laptop, the best algorithm is to make a direct call to LAPACK without any decomposition.

As a baseline, we include the matrix multiplication OpenCL sample from the NVIDIA SDK (*Hand-coded OpenCL* in Figure 4.10(e)). This baseline runs 1.4x faster than our autotuned configuration on Desktop. The reason for this difference is the hand-coded OpenCL uses a number complex manual local memory optimizations that accumulate partially computed outputs in local memory shared between work-items. We have not implemented a similar optimization in our system; however, it would be possible to automatically perform a similar optimization.

Singular Value Decomposition (SVD) The results for SVD (Figure 4.10(f)) are particularly interesting because on some systems the autotuner constructs a poly-algorithm with task parallel divisions between the GPU and the CPU, and the complexity of the benchmark provides a large number of choices in the search space. This benchmark approximates a matrix through a factorization that consumes less space. SVD is a variable accuracy benchmark where many of the choices available to the autotuner, such as how many eigenvalues to use, impact the quality of of the approximation. The autotuner must produce an algorithm which meets a given accuracy target. These variable accuracy features are described in more detail in Chapters 2 and 3.

On Desktop, the autotuned configuration divides the work by using the GPU to compute one matrix and the CPU to concurrently compute another matrix. Since the computations of the two matrices are very similar, and the CPU and the GPU have relatively similar performance for these computations on Desktop, overall throughput is increased by dividing the work in this way.

This benchmark also demonstrates that the best configurations of a sub-program might be different when the sub-program is a part of different applications even running on the same machine. The SVD benchmark uses the Strassen code to perform matrix multiplication. However SVD uses matrix multiply on sub-regions of multiple larger arrays (resulting in different data locality) and possibly making multiple calls concurrently. Due to the different data-accessing patterns, the cache behaviors are not the same on the CPU, the bank conflicts on GPU memory vary, and the interactions between subsystems change. This makes the best configurations differ for Strassen

inside SVD and in isolation. While the best matrix multiplication configuration on Desktop for Strassen in isolation always uses the GPU, the best one for this benchmark is 8-way parallel recursive decomposition and then calling LAPACK. The autotuned configurations on Server and Laptop for this benchmark are also different from Strassen in isolation.

Tridiagonal Solver The Tridiagonal Solver benchmark (Figure 4.10(g)) shows that often algorithmic changes are required to utilize the GPU. The benchmark solves a system of a equations where the matrix contains non-zero elements only on cells neighboring and on the diagonal and includes a variety of techniques including polyalgorithms. We implement a subset of the algorithmic choices described in [48, 167].

Similar to the Strassen benchmark, the GPU is only used on the Desktop machine; however, in order to utilize the GPU, an algorithmic change is required. Cyclic reduction is the best algorithm for Desktop when using the GPU. If a machine does not use OpenCL, it is better to run the sequential algorithm as demonstrated used on Server and Laptop.

Our best configuration on Desktop is 3.5x slower than CUDPP [167] on input size 512. Some of this slowdown is a result of OpenCL being generally slower than CUDA when using the NVIDIA toolchain. Additionally, our automatically generated OpenCL kernel is not as highly optimized as CUDPP's kernel, which guarantees the efficient use of shared memory without bank conflicts.

4.6.3 Heterogeneous Results Summary

In all of our benchmarks, algorithmic choices—which traditional languages and compilers do not expose—play a large role in performance on heterogeneous systems since the best algorithms vary not only between machines but also between processors within a machine. Ultimately, the complex and interdependent space of best mapping choices seen in these benchmarks would be very difficult to predict from first principles, alone, but our empirical exploration effectively and automatically accounts for many interacting effects on actual machine performance, in each program on each system.

Taken together, these seven benchmarks demonstrate even more of the complexity in mapping programs to heterogeneous machines than any one benchmark alone.

- *Strassen* shows that choosing the right configuration for each specific architecture can provide a huge performance improvement. In this benchmark, choosing to use a data parallel accelerator can yield large speedups (16.5x) on some machines, and large slowdowns (4.1x) on others, for the same problem, depending on the exact characteristics of all heterogeneous processors in the system.

- *Poisson 2D SOR* further supports the previous argument by showing that the optimal placement of computations across the processors in one system is almost the opposite of another.

- *Tridiagonal Solver* demonstrates that not only where computations run, but also which algorithms to use on that particular resource, can dramatically affect performance.

- *Separable Convolution* shows that, even when a program is best run entirely on the data-parallel compute resources, the best algorithm to use and the best strategy for mapping to the complex heterogeneous memory hierarchy vary widely both across machines and across program parameters (kernel size).

- *Sort*, on the other hand, shows that even parallel problems may not always be best solved by data parallel accelerators, and that complex machine-specific algorithmic choice is still essential to performance on a smaller array of conventional processors.

- *SVD* shows that the best performance can require mapping portions of a program across all different processor types available, and together with Strassen, it shows that the best configurations of the same sub-program in different applications vary on the same system.

- Finally, while SVD confirms that sometimes it is best to run different portions of a program on different subsystems concurrently, *Black-Scholes* illustrates that sometimes it is best to run the same portion of the program but different regions of data across multiple heterogeneous subsystems simultaneously; therefore, considering the workload balance between processor types is important to achieve the optimal performance on a heterogeneous system.

4.7 Summary

This chapter has shown three sets of results. First, we saw that for fixed accuracy benchmarks, PetaBricks can provide significant speedups over using a single algorithm or a hard coded heuristic. Next, our variable accuracy results showed the by trading accuracy for performance one can achieve large (sometimes asymptoticly better) speedups. Finally, we explored heterogeneous results that show there is no one size fits all solution for all execution targets. One often needs to do fundamentally different things on different machines for the best performance.

Chapter 5

Multigrid Benchmarks

Our multigrid benchmarks (Helmholz3D and Poisson2D) are particularly interesting, and represent distinct contribution in their domain. Multigrid is a popular techniques for efficiently solving partial differential equations over a grid. This chapter explores these two benchmarks in greater detail.

In some cases, tuning algorithmic choice could simply mean choosing the appropriate top-level technique during the initial function invocation; however, for many problems including multigrid, it is better to be able to utilize multiple techniques within a single function call or solve. For example, in the C++ Standard Template Library's stable_sort routine (Shown in Figure 1.1 in Chapter 1), the algorithm switches from using the divide-and-conquer $O(n \log n)$ merge sort to $O(n^2)$ insertion sort once the working array size falls below a set cutoff. In multigrid, an analogous strategy might switch from recursive multigrid calls to a direct method such as Cholesky factorization and triangular solve once the problem size falls below a threshold.

This chapter analyzes the optimizations of algorithmic choice in multigrid. When confronted with the problem of training the autotuner to choose between a recursive multigrid call and a call to an iterative or direct solver, one quickly realizes that no comparison between methods can be fair without considering the relative accuracies of each. Indeed, we found that in some cases sacrificing accuracy at lower levels of recursion has little impact on the accuracy of the final result, while in other cases improving accuracy at a lower level reduces the number of (more expensive) iterations needed at a higher level.

Algorithm	Direct	SOR	Multigrid
Complexity	n^2 (N^4)	$n^{1.5}$ (N^3)	n (N^2)

Figure 5.1: Complexity of different algorithmic choices for multigrid.

5.1 Autotuning Multigrid

Although multigrid is a versatile technique that can be used to solve many different types of problems, we will use the 2D Poisson's equation as an example and benchmark to guide our discussion. The techniques presented here are generalizable to higher dimensions and the broader set of multigrid problems.

Poisson's equation is a partial differential equation that describes many processes in physics, electrostatics, fluid dynamics, and various other engineering disciplines. The continuous and discrete versions are

$$\nabla^2 \phi = f \quad \text{and} \quad Tx = b, \tag{5.1}$$

where T, x, and b are the finite difference discretizations 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). Figure 5.1 shows the computational complexity of using any single algorithmic choice to compute a solution. From left to right, each of the methods has a larger overhead, but yields a better asymptotic serial complexity [52]. N is the size of the grid on a side, and $n = N^2$ is the number of cells in the grid.

5.1.1 Algorithmic choice in multigrid

Multigrid is a recursive algorithm that uses the solution to a coarser grid resolution as part of the algorithm. We will first address tuning symmetric "V-type" cycles. An extension to full multigrid will be presented in Section 5.1.4.

```
MULTIGRID-V-SIMPLE(x, b)
1: if N = 3 then
2:     Solve directly
3: else
4:     Relax using some iterative method
5:     Compute the residual and restrict to half resolution
6:     Recursively call MULTIGRID-V-SIMPLE on coarser grid
7:     Interpolate result and add correction term to current solution
8:     Relax using some iterative method
9: end if
```

Figure 5.2: Pseudocode for standard multigrid v-cycle.

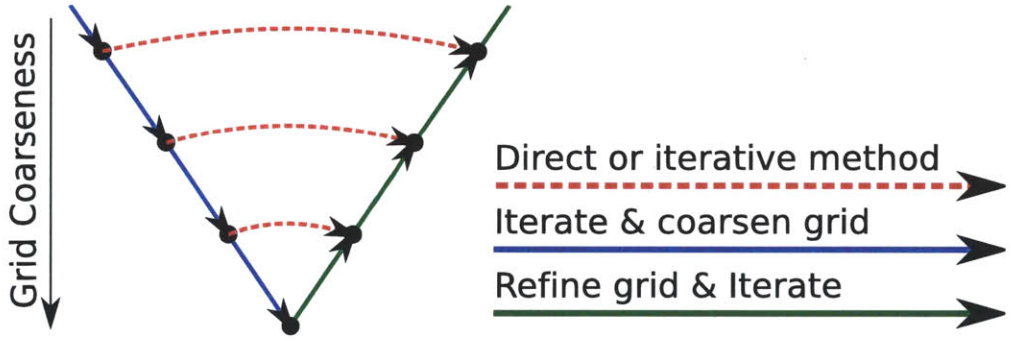


Figure 5.3: Simplified illustration of choices in the multigrid algorithm. The diagonal arrows represent the recursive case, while the dotted horizontal arrows represent the shortcut case where a direct or iterative solution may be substituted. Depending on the desired level of accuracy a different choice may be optimal at each decision point. This figure does not illustrate the autotuner's capability of using multiple iterations at different levels of recursion; it shows a single iteration at each level.

For simplicity, we assume all inputs are of size $N = 2^k + 1$ for some positive integer k. Let x be the initial state of the grid, and b be the right hand side of Equation (5.1).

Figure 5.2 shows pseudocode for a standard multigrid v-cycle. It is at the recursive call on line 6 that our autotuning compiler can make a choice of whether to continue making recursive calls to multigrid or take a shortcut by using the direct solver or one of the iterative solvers at the current resolution. Figure 5.3 shows these possible paths of the multigrid algorithm.

Figure 5.4 shows the idea of algorithmic choice can be implemented by defining a top level function `MULTIGRID-V`, which makes calls to either the direct, iterative, or recursive solution. The function `RECURSE` implements the recursive solution. Making the choice on line 1 of `MULTIGRID-V`

```
MULTIGRID-V(x, b)
1: either
2:    Solve directly
3:    Use an iterative method
4:    Call RECURSE for some number of iterations
5: end either

RECURSE(x, b)
1: if N = 3 then
2:    Solve directly
3: else
4:    Relax using some iterative method
5:    Compute the residual and restrict to half resolution
6:    On the coarser grid, call MULTIGRID-V
7:    Interpolate result and add correction term to current solution
8:    Relax using some iterative method
9: end if
```

Figure 5.4: Pseudocode for multigrid with algorithmic choices

has two implications. First, the time to complete the algorithm is choice dependent. Second, the accuracy of the result is also dependent on choice since the various methods have different abilities to reduce error (depending on paramcters such as number of iterations or weights). To make a fair comparison between choices, we must take both performance and accuracy of each choice into account. To this end, during the tuning process, we keep track of not just a single optimal algorithm at every recursion level, but a *set* of such optimal algorithms for varying levels of desired accuracy.

5.1.2 Full dynamic programming solution

We will first describe a full dynamic programming solution to handling variable accuracy, then restrict it to a discrete set of accuracies. We define an algorithm's *accuracy level* to be the ratio between the error norm of its input x_{in} versus the error norm of its output x_{out} compared to the optimal solution x_{opt}:

$$\frac{||x_{in} - x_{opt}||_2}{||x_{out} - x_{opt}||_2}.$$

We choose this ratio instead of its reciprocal so that a higher accuracy level is better, which is more intuitive. In order to measure the accuracy level of a potential tuned algorithm, we assume

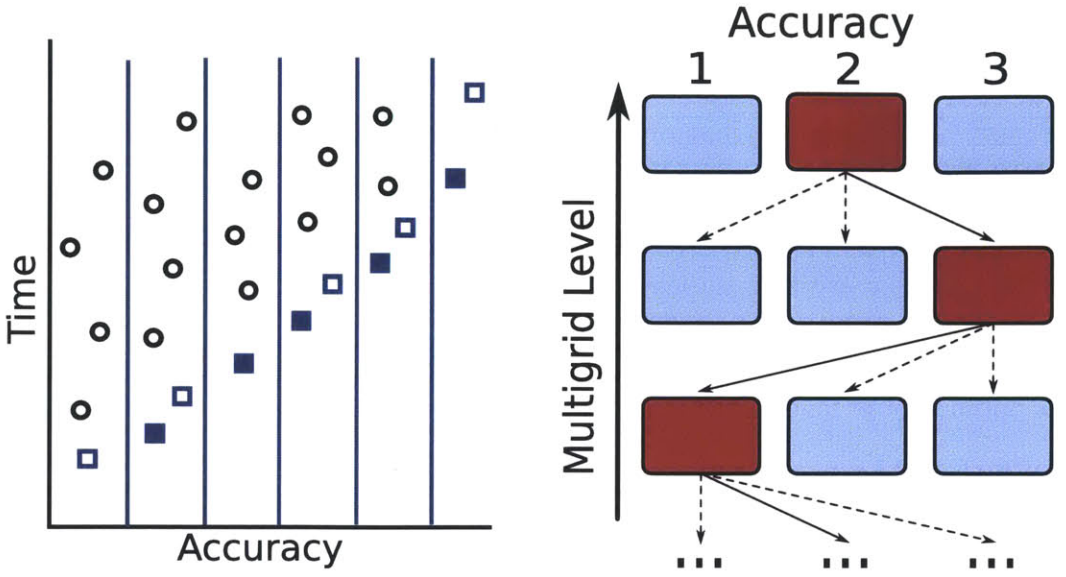


Figure 5.5: (a) Possible algorithmic choices with optimal set designated by squares (both hollow and solid). The choices designated by solid squares are the ones remembered by the PetaBricks compiler, being the fastest algorithms better than each accuracy cutoff line. (b) Choices across different accuracies in multigrid. At each level, the autotuner picks the best algorithm one level down to make a recursive call. The path highlighted in red is an example of a possible path for accuracy level p_2

we have access to representative training data so that the accuracy level of our algorithms during tuning closely reflects their accuracy level during use.

Let level k refer to an input size of $N = 2^k + 1$. Suppose that for level $k-1$, we have solved for some set A_{k-1} of optimal algorithms, where optimality is defined such that no optimal algorithm is dominated by any other algorithm in both accuracy and compute time.

In order to construct the optimal set A_k, we try substituting all algorithms in A_{k-1} for step 6 of `RECURSE`. We also try varying parameters in the other steps of the algorithm, including the choice of iterative methods and the number of iterations (possibly zero) in steps 4 and 8 of `RECURSE` and steps 3 and 4 of `MULTIGRID-V`.

Trying all of these possibilities will yield many algorithms that can be plotted as in Figure 5.5(a) according to their accuracy and compute time. The optimal algorithms we add to A_k are the dominant ones designated by square markers.

The reason to remember algorithms of multiple accuracies on this Pareto frontier for use in step 6 of RECURSE is that it may be better to use a less accurate, fast algorithm and then iterate multiple times, rather than use a more accurate, slow algorithm. Note that even if we use a direct solver in step 6, the interpolation in step 7 will invariably introduce error at the higher resolution.

5.1.3 Discrete dynamic programming solution

Since the optimal set of tuned algorithms can grow to be very large, the PetaBricks autotuner offers an approximate version of the above solution. Instead of remembering the full optimal set A_k, the compiler remembers the fastest algorithm yielding an accuracy of at least p_i for each p_i in some set $\{p_1, p_2, \ldots, p_m\}$. The vertical lines in Figure 5.5(a) indicate the discrete accuracy levels p_i, and the optimal algorithms (designated by solid squares) are the ones remembered by PetaBricks. Each highlighted algorithm is associated with a function MULTIGRID-V$_i$, which achieves accuracy p_i on all input sizes.

Due to restricted time and computational resources, to further narrow the search space, we only use SOR as the iteration function since we found experimentally that it performed better than weighted Jacobi on our particular training data for similar computation cost per iteration. In MULTIGRID-V$_i$, we fix the weight parameter of SOR to ω_{opt}, the optimal value for the 2D discrete Poisson's equation with fixed boundaries [52]. In RECURSE$_i$, we fix SOR's weight parameter to 1.15 (chosen by experimentation to be a good parameter when used in multigrid). We also fix the number of iterations of SOR in steps 4 and 8 in RECURSE$_i$ to one. As more powerful computational resources become available over time, the restrictions on the algorithmic search space presented here may be relaxed to find a more optimal solution.

The resulting accuracy-aware Poisson solver, shown in Figure 5.6, is a family of functions, where i is the accuracy parameter. The autotuning process determines what choices to make in MULTIGRID-V$_i$ for each i and for each input size. Since the optimal choice for any single accuracy for an input of size $2^k + 1$ depends on the optimal algorithms for *all* accuracies for inputs of size $2^{k-1} + 1$, the PetaBricks autotuner tunes all accuracies at a given level before moving to a higher level. In this way, the autotuner builds optimal algorithms for every specified accuracy level and

```
MULTIGRID-V_i(x, b)
1: either
2:    Solve directly
3:    Iterate using SOR_{ω_opt} until accuracy p_i is achieved
4:    For some j, iterate with RECURSE_j until accuracy p_i is achieved
5: end either

RECURSE_i(x, b)
1: if N = 3 then
2:    Solve directly
3: else
4:    Compute one iteration of SOR_{1.15}
5:    Compute the residual and restrict to half resolution
6:    On the coarser grid, call MULTIGRID-V_i
7:    Interpolate result and add correction term to current solution
8:    Compute one iteration of SOR_{1.15}
9: end if
```

Figure 5.6: Pseudocode for *accuracy aware* multigrid

for each input size up to a user specified maximum, making use of the tuned sub-algorithms as it goes.

The final set of multigrid algorithms produced by the autotuner can be visualized as in Figure 5.5(b). Each of the versions has the flexibility to choose any of the other versions during its recursive calls, and the optimal path may switch between accuracies many times as we recurse down towards either the base case or a shortcut case.

5.1.4 Extension to Autotuning Full Multigrid

Full multigrid methods have been shown to exhibit better convergence behavior than traditional symmetric cycle shapes such as the V and W cycles by utilizing an estimation phase before the solve phase (see Figure 5.7). The estimation phase of the full multigrid algorithm can be thought of as just a recursive call to itself at a coarser grid resolution. We extend the autotuning ideas presented thus far to leverage this structure and produce autotuned full multigrid cycles.

Figure 5.8 shows pseudocode for `ESTIMATE` and `FULL-MULTIGRID` and illustrates how to construct an autotuned full multigrid cycle. Here we take advantage of the discrete dynamic programming

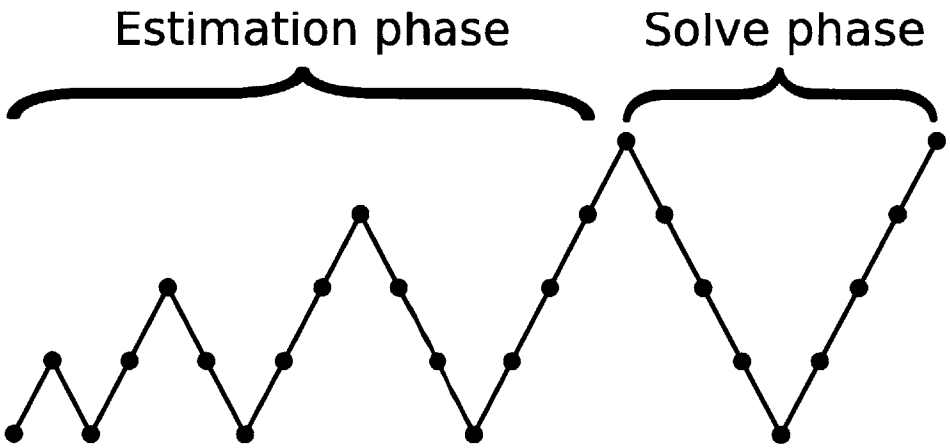


Figure 5.7: Conceptual breakdown of full multigrid into an estimation phase and a solve phase. The estimation phase can be thought of as just a recursive call to full multigrid up to a coarser grid resolution. We make use of this recursive structure, in addition to our autotuned "V-type" multigrid cycles, in constructing tuned full multigrid cycles.

analogue presented in Section 5.1.3 where we maintain only finite sets of optimized functions FULL-MULTIGRID$_j$ and MULTIGRID-V$_k$ to use in recursive calls. In FULL-MULTIGRID$_i$, there are three choices: the first is just a direct solve (line 2), while the latter two choices (lines 4 and 5) are similar to those given in MULTIGRID-V$_i$ except an estimate is first calculated and then used as a starting point for iteration. Note that this structure is descriptive enough to include the standard full multigrid V or W cycle shapes, just as the MULTIGRID-V$_i$ algorithm can produce standard regular V or W cycles.

The parameters j and k in FULL-MULTIGRID can be chosen independently, providing a great deal of flexibility in the construction of the optimized full multigrid cycle shape. In cases where the user does not require much accuracy in the final output, it may make sense to invest more heavily in the estimation phase, while in cases where very high precision is needed, a high precision estimate may not be as helpful as most of the computation would be done in relaxations at the highest resolution. Indeed, we found patterns of this type during our experiments.

ESTIMATE$_i(x, b)$

1: Compute residual and restrict to half resolution
2: Call FULL-MULTIGRID$_i$ on restricted problem
3: Interpolate result and add correction to x

FULL-MULTIGRID$_i(x, b)$

1: **either**
2: Solve directly
3: For some j, compute estimate by calling ESTIMATE$_j(x, b)$, then **either**:
4: Iterate using SOR$_{\omega_{\text{opt}}}$ until accuracy p_i is achieved
5: For some k, iterate with RECURSE$_k$ until accuracy p_i is achieved
6: **end either**

Figure 5.8: Pseudocode for full multigrid

5.1.5 Limitations

Multigrid benchmarks are sensitive to input variations. Results in this chapter examine a single distribution of inputs. Chapter 7 presents additional multigrid results that take into account input sensitivity.

It should be clear that the algorithms produced by the autotuner are not meant to be optimal in any theoretical sense. Because of the compromises made in the name of efficiency, the resulting autotuning algorithm merely strives to discover near-optimal algorithms from within the restricted space of cycle shapes reachable during the search. There are many cycle shapes that fall outside the space of searched algorithms; for example, our approach does not check algorithms that utilize different choices in succession at the same recursion depth instead of choosing a single choice and iterating. Future work may examine the extent to which this restriction impacts performance.

Additionally, the scalar accuracy metric is an imperfect measure of the effectiveness of a multigrid cycle. Each cycle may have different effects on the various error modes (frequencies) of the current guess, all of which would be impossible to capture in a single number. Future work may expand the notion of an "optimal" set of sub-algorithms to include separate classes of algorithms that work best to reduce different types of error. Though such an approach could lead to a better final tuned algorithm, this extension would obviously make the auto-tuning process more complex.

We will demonstrate in our results that although our methodology is not exhaustive, it can be quite descriptive, discovering cycle shapes that are both unconventional and efficient. That section will present actual cycle shapes produced by our multigrid autotuner and show their performance compared to less sophisticated heuristics.

5.2 Results

In this section, we present the results of the PetaBricks autotuner when optimizing our multigrid algorithm on three parallel architectures designed for a variety of purposes: Intel Xeon E7340 server processor, AMD Opteron 2356 Barcelona server processor, and the Sun Fire T200 Niagara low power, high throughput server processor. These machines provided architectural diversity, allowing us to show not only how autotuned multigrid cycles outperform reference multigrid algorithms, but also how the shape of optimal autotuned cycles can be dependent on the underlying machine architecture.

To the best of our knowledge, there are no standard data distributions currently in wide use for benchmarking multigrid solvers, so it was not clear what the best choice is for training and benchmarking our tuned solvers. We decided to use matrices with entries drawn from two different random distributions: 1) uniform over $[-2^{32}, 2^{32}]$ (unbiased), and 2) the same distribution shifted in the positive direction by 2^{31} (biased). The random entries were used to generate right-hand sides (b in Equation 5.1) and boundary conditions (boundaries of x) for the problem. We also experimented with specifying a finite number of random point sources/sinks in the right-hand side, but since the observed results were similar to those found with the unbiased random distribution, we did not include them in interest of space. If one wishes to obtain tuned multigrid cycles for a different input distribution, the training should be done using that data distribution.

5.2.1 Autotuned multigrid cycle shapes

During the tuning process for the `MULTIGRID-V`$_i$ algorithm presented in Section 5.1.3, the autotuner first computes the number of iterations needed for the SOR and `RECURSE`$_j$ choices before determining which is the fastest option to attain accuracy p_i for each input size. Representative training data

is required to make this determination. Once the number of required iterations of each choice is known, the autotuner times each choice and chooses the fastest option.

Figures 5.9(a) and 5.9(b) show the traces of calls to the tuned MULTIGRID-V_4 algorithms for unbiased and biased uniform random inputs of size $N = 4097$, on the Intel machine. As you can see, the algorithm utilizes multiple accuracy levels throughout the call stack. In general, whenever greater accuracy is required by our tuned algorithm, it is achieved through some repetition of optimal substructures determined by the dynamic programming method. This may be easier to visualize by examining the resulting tuned cycles corresponding to the autotuned multigrid calls.

Figures 5.10(a) and 5.10(b) show some tuned "V-type" cycles created by the autotuner for unbiased and biased uniform random inputs of size $N = 2049$ on the AMD Opteron machine. The cycles are shown using standard multigrid notation with some extensions: The path of the algorithm progresses from left to right through time. As the path moves down, it represents a restriction to a coarser resolution, while paths up represent interpolations. Dots represent red-black SOR relaxations, solid horizontal arrows represent calls to the direct solver, and dashed horizontal arrows represent calls to the iterative solver.

As seen in the figure, a different cycle shape is used depending on what level of accuracy is required by the user. Cycles shown are tuned to produce final accuracy levels of $10, 10^3, 10^5$, and 10^7. The leverage of optimal subproblems is clearly seen in the common patterns that appear across cycles. Note that in Figure 5.10(b), the call to the direct solver in cycle i) occurs at level 4, while for the other three cycles, the direct call occurs at level 5. This is an example of the autotuner trading accuracy for performance while accounting for the accuracy requirements of the user.

Figures 5.10(c) and 5.10(d) show autotuned full multigrid cycles for unbiased and biased uniform random inputs of size $N = 2049$ on the AMD Opteron machine. Although similar substructures are shared between these cycles and the "V-type" cycles in 5.10(a) and 5.10(b), some of the expensive higher resolution relaxations are avoided by allowing work to occur at the coarser grids during the estimation phase of the full multigrid algorithm. The tuned full multigrid cycle in Figure 5.10(d)-iv) shows how the additional flexibility of using an estimation phase can dramatically alter the tuned cycle shape when compared to Figure 5.10(b)-iv).

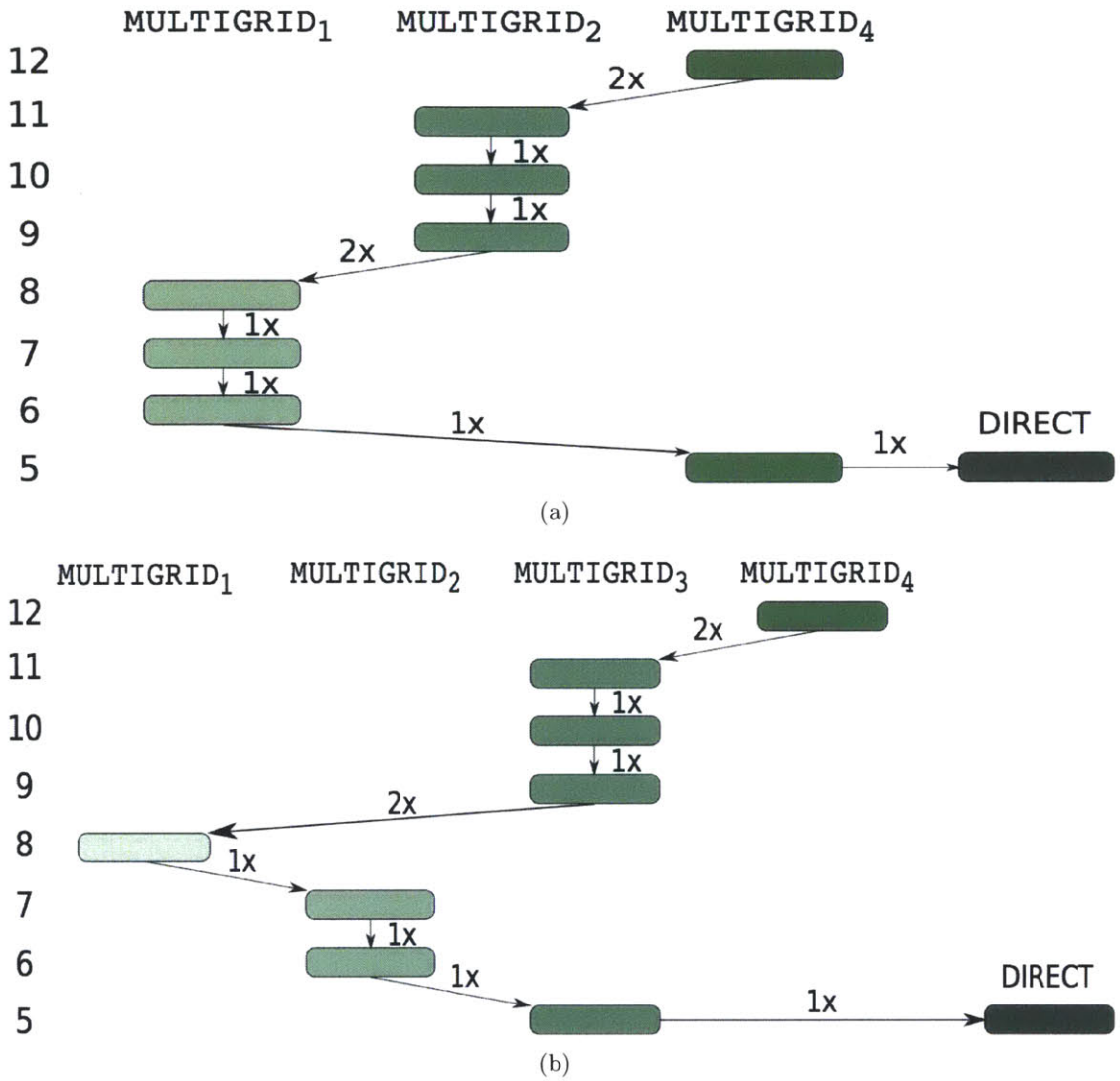


Figure 5.9: Call stacks generated by calls to autotuned MULTIGRID-V_4 for a) unbiased and b) biased random inputs of size $N = 4097$ on an Intel Xeon server. Discrete accuracies used during autotuning were $(p_i)_{i=1..5} = (10, 10^3, 10^5, 10^7, 10^9)$. The recursion level is displayed on the left, where the size of the grid at level k is $2^k + 1$. Note that each arrow connecting to a lower recursion level actually represents a call to RECURSE$_i$, which handles grid coarsening, followed by a call to MULTIGRID-V_i.

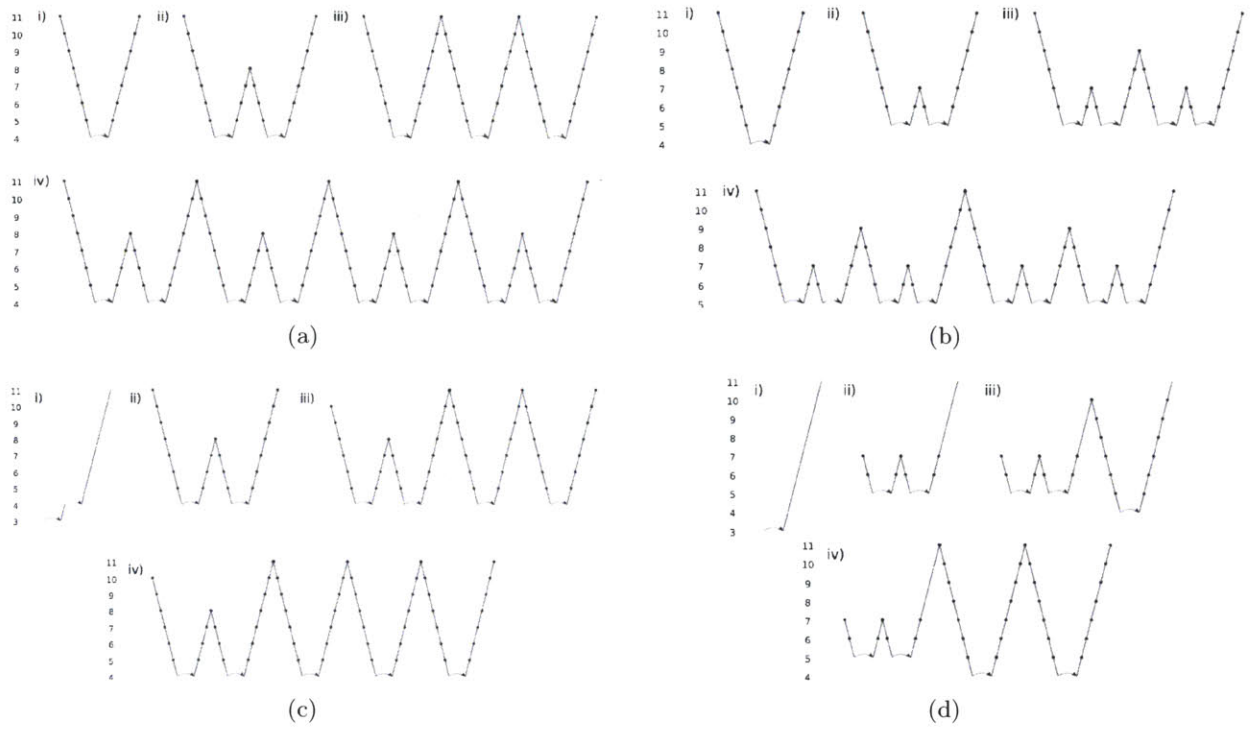


Figure 5.10: Optimized multigrid V (a and b) and full multigrid (c and d) cycles created by the autotuner for solving the 2D Poisson's equation on an input if size $N = 2049$. Subfigures a) and c) were trained on unbiased uniform random data, while b) and d) were trained on biased uniform random data. Cycles i), ii), iii), and iv), correspond to algorithms that yield accuracy levels of $10, 10^3, 10^5$, and 10^7, respectively. The solid arrows at the bottom of the cycles represent shortcut calls to the direct solver, while the dashed arrow in c)-i) 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 recursion level is displayed on the left, where the size of the grid at level k is $2^k + 1$.

It is important to realize that the call stacks in Figure 5.9 and the cycle shapes in Figure 5.10 are all dependent on the specific situation at hand. They would all likely change were the autotuner run on other architectures, using different training data, or solving other multigrid problems. The flexibility to adapt to any of these changing variables by tuning over algorithmic choice is the autotuner's greatest strength.

5.2.2 Performance

This section will provide data showing the performance of our tuned multigrid Poisson's equation solver versus reference algorithms and heuristics. Test data was produced from the same distributions used for training. Section 5.2.2 describes performance of the autotuned `MULTIGRID-V` algorithm, and Section 5.2.2 describes the performance of the autotuned `FULL-MULTIGRID` algorithm.

Autotuned multigrid V algorithm

To demonstrate the effectiveness of our dynamic programming methodology, we compare the autotuned `MULTIGRID-V` algorithm against more basic approaches to solving the 2D Poisson's equation to an accuracy of 10^9, including several multigrid variations. Results presented in the section were collected on the Intel Xeon server testbed machine.

Figure 5.11 shows the performance of our autotuned multigrid algorithm for accuracy 10^9 on unbiased uniform random inputs of different sizes. The autotuned algorithm uses internal accuracy levels of $\{10, 10^3, 10^5, 10^7, 10^9\}$ during its recursive calls. The figure compares the autotuned algorithm with the direct solver, iterated calls to SOR, and iterated calls to `MULTIGRID-V-SIMPLE` (labeled Multigrid). Each of the iterative methods is run until an accuracy of at least 10^9 is achieved.

As to be expected, the autotuned algorithm outperforms all of the simple algorithms shown in Figure 5.11. At sizes greater than $N = 65$, the autotuned algorithm performs slightly better than `MULTIGRID-V-SIMPLE` because it utilizes a more complex tuned strategy.

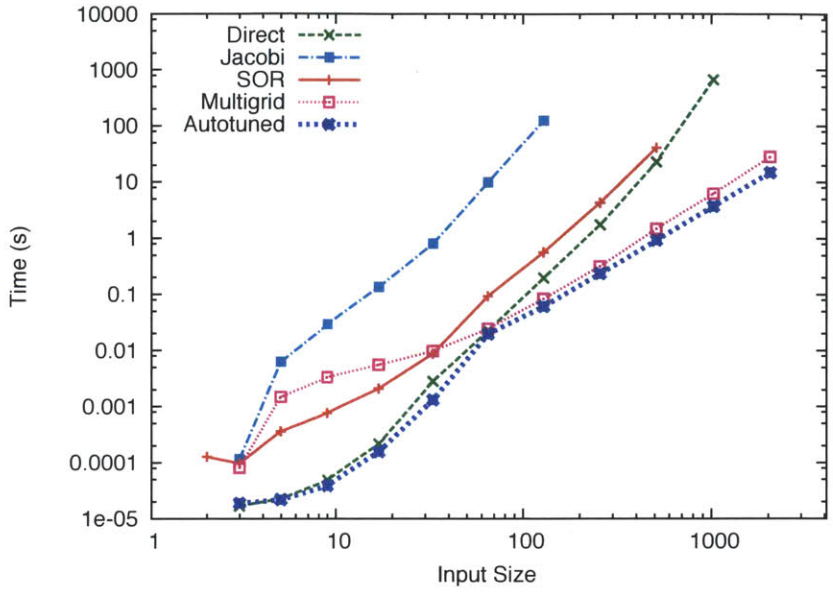


Figure 5.11: Performance for algorithms to solve Poisson's equation on unbiased uniform random data up to an accuracy of 10^9 using 8 cores. The basic direct and SOR algorithms as well as the standard V-cycle multigrid algorithm are all compared to our tuned multigrid algorithm. The iterated SOR algorithm uses the corresponding optimal weight ω_{opt} for each of the different input sizes

Figure 5.12 compares the tuned algorithm with various heuristics more complex than `MULTIGRID-V-SIMPLE`. The training data used in this graph was drawn from the biased uniform distribution. Strategy 10^9 refers to requiring an accuracy of 10^9 at each recursive level of multigrid until the base case direct method is called at $N = 65$. Strategies of the form $10^x/10^9$ refer to requiring an accuracy of 10^x at each recursive level below that of the input size, which requires an accuracy of 10^9. Thus, all strategies presented result in a final accuracy of 10^9; they differ only in what accuracies are required at lower recursion levels. All heuristic strategies call the direct method for smaller input sizes whenever it is more efficient to meet the accuracy requirement.

The lines in Figure 5.12 are somewhat close together and difficult to see on the logarithmic time scale, so Figure 5.13 presents the same data but showing the ratio of times taken versus the autotuned algorithm. We can more clearly see in this figure that as the input size increases, the most efficient heuristic changes from Strategy $10^1/10^9$ to $10^3/10^9$ to $10^5/10^9$. The autotuner does better than just choosing the best from among these heuristics, since it can also tune the desired accuracy at each recursion level independently, allowing greater flexibility. This figure highlights

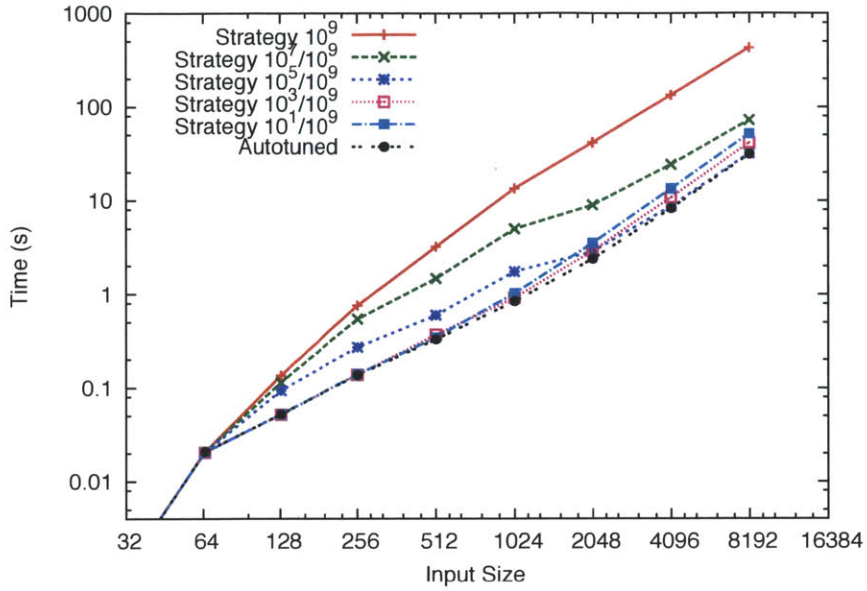


Figure 5.12: Performance for algorithms to solve Poisson's equation up to an accuracy of 10^9 using 8 cores. The autotuned multigrid algorithm is presented alongside various possible heuristics. The graph omits sizes less than $N = 65$ since all cases call the direct method for those inputs. To see the trends more clearly, Figure 5.13 shows the same data as this figure, but as ratios of times taken versus the autotuned algorithm.

the complexity of finding an optimal strategy and showcases the utility of an autotuner that can efficiently find this optimum.

Another big advantage of using PetaBricks for autotuning is that it allows a single program to be optimized for both sequential performance and parallel performance. We have observed our autotuner make different choices when running on different numbers of cores.

Autotuned full multigrid algorithm

In order to evaluate the performance of our autotuned `MULTIGRID-V` and `FULL-MULTIGRID` algorithms on multiple architectures, we ran them for problem sizes up to $N = 4097$ (up to 2049 on the Sun Niagara) for target accuracy levels of 10^5 and 10^9 alongside two reference algorithms: an iterated V cycle and a full multigrid algorithm. The reference V cycle algorithm runs standard V cycles until the accuracy target is reached, while the reference full multigrid algorithm runs a

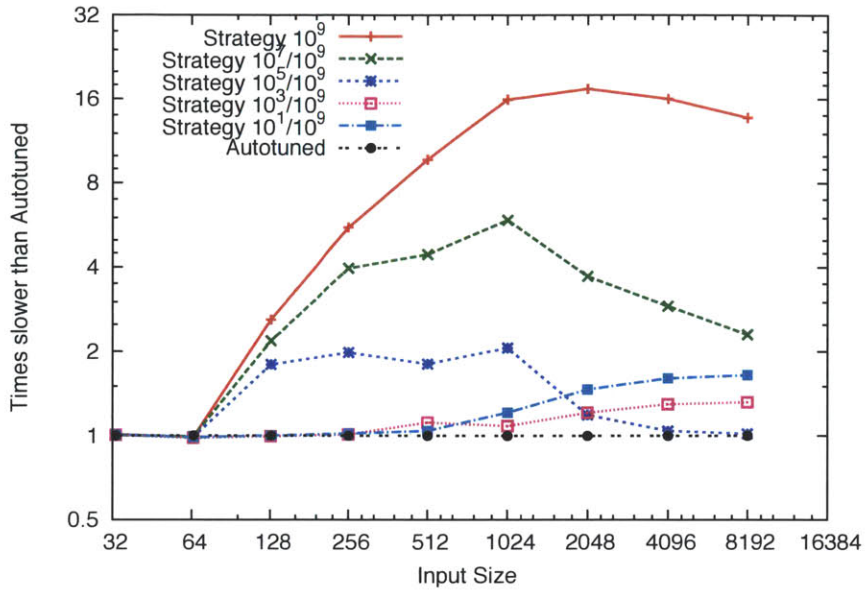


Figure 5.13: Speedup of tuned algorithm compared to various simple heuristics to solve Poisson's equation up to an accuracy of 10^9 using 8 cores. The data presented in this graph is the same as in Figure 5.12 except that the ratio of time taken versus the autotuned algorithm is plotted. Notice that as the problem size increases, the higher accuracy heuristics become more favored since they require fewer iterations at high resolution grid sizes.

standard full multigrid cycle (as in Figure 5.7), then standard V cycles until the accuracy target is reached.

We chose these two reference algorithms since they are generally deemed good starting points for those interested in implementing multigrid for the first time. Since they are easy to understand and commonly implemented, we felt they were a reasonable point of reference for our results. From these starting points, performance tweaks can be manually applied to tailor the solver to each user's specific application domain. The goal of our autotuner is to discover and make these tweaks automatically.

Figure 5.14 shows the performance of both reference and autotuned multigrid algorithms for unbiased uniform random data relative to the reference iterated V-cycle algorithm on all three testbed machines. Figure 5.15 shows similar comparisons for biased uniform random data. The relative time (lower is better) to compute the solution up to an accuracy level of 10^5 is plotted against problem size.

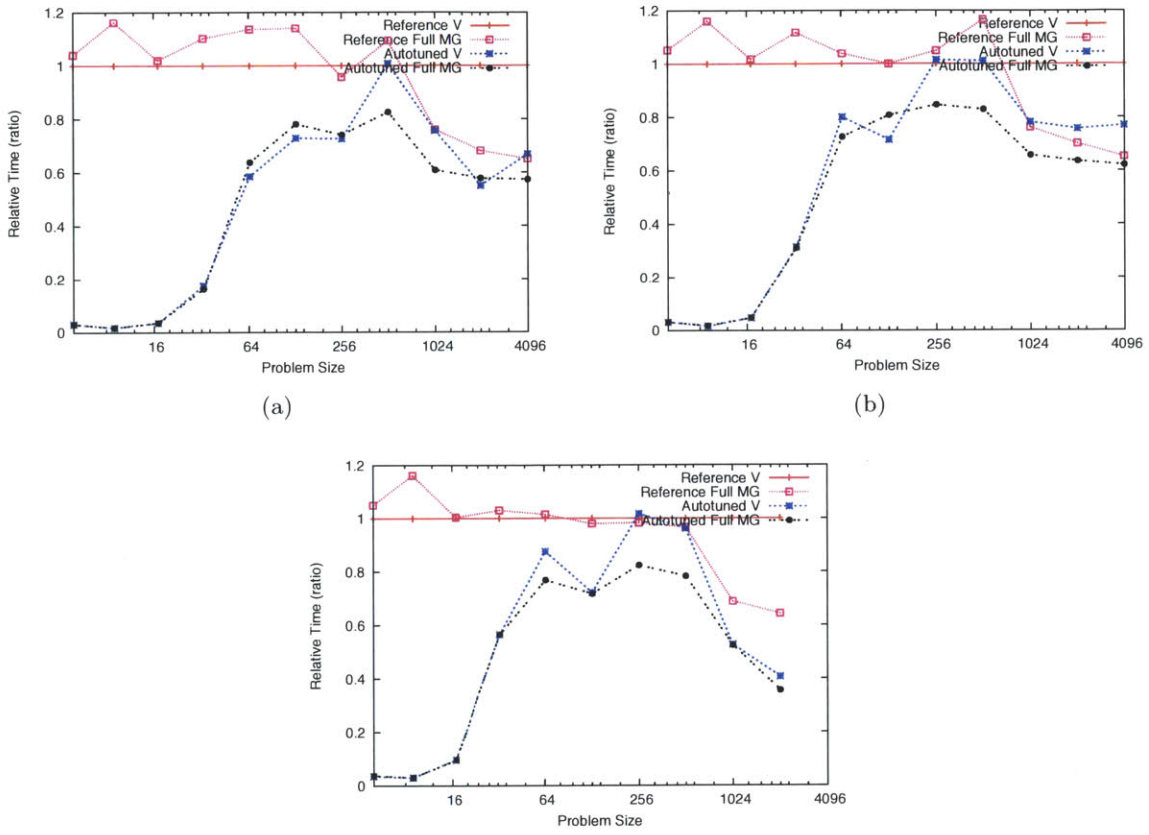


Figure 5.14: Relative performance of multigrid algorithms versus reference V cycle algorithm for solving the 2D Poisson's equation on unbiased, uniform random data to an accuracy level of 10^5 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

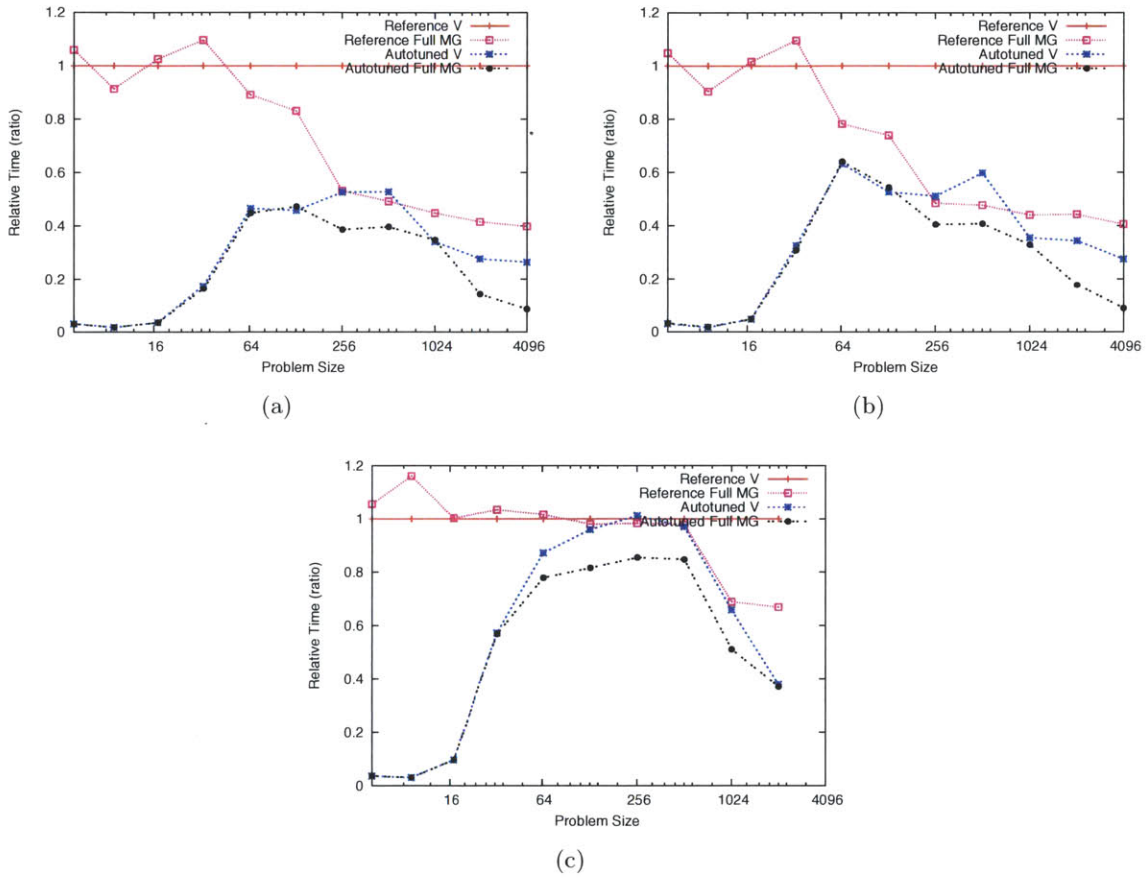


Figure 5.15: Relative performance of multigrid algorithms versus reference V cycle algorithm for solving the 2D Poisson's equation on biased uniform random data to an accuracy level of 10^5 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

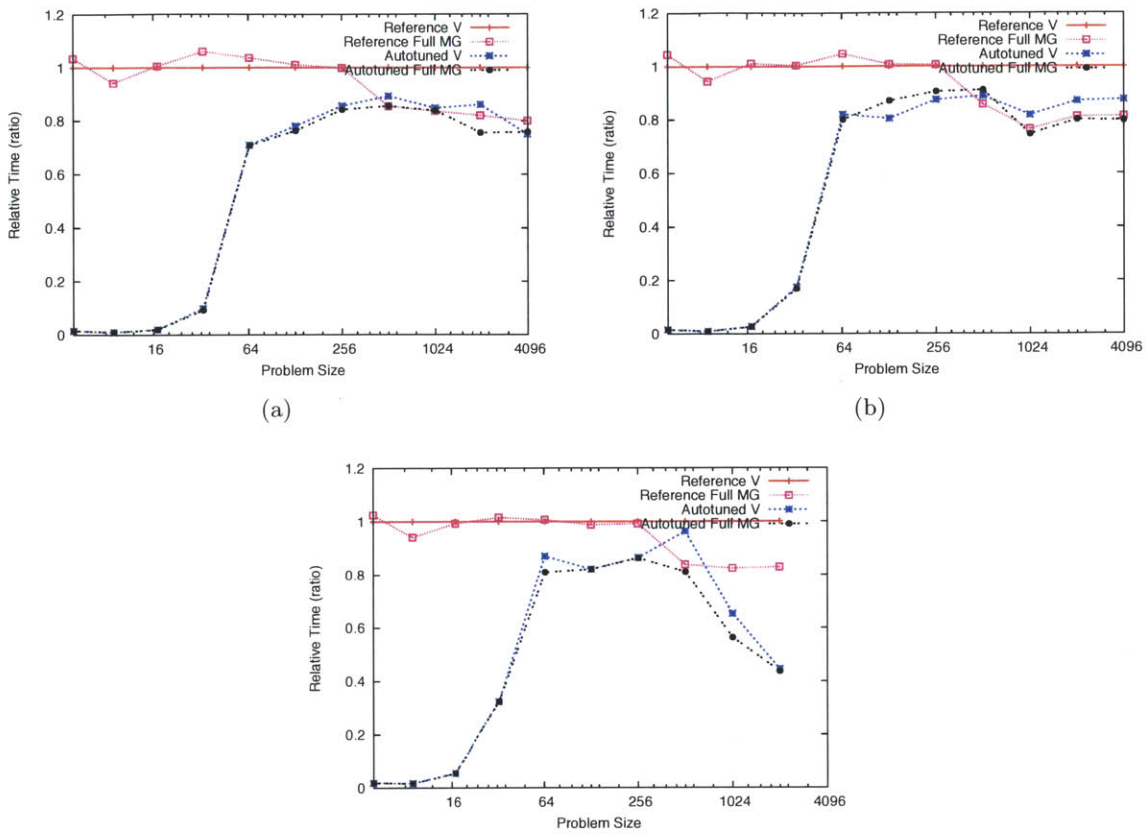


Figure 5.16: Relative performance of multigrid algorithms versus reference V cycle algorithm for solving the 2D Poisson's equation on unbiased, uniform random data to an accuracy level of 10^9 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

On all three architectures, we see that the autotuned algorithms provide an improvement over the reference algorithms' performances. There is an especially marked difference for small problem sizes due to the autotuned algorithms' use of the direct solve without incurring the overhead of recursion. Speedups relative to the reference full multigrid algorithm are also observed at higher problem sizes: e.g., for problem size $N = 2049$, we observed speedups of 1.2x, 1.1x, and 1.8x on the unbiased uniform test inputs, and 2.9x, 2.5x, and 1.8x on the biased uniform test inputs for the Intel, AMD, and Sun machines, respectively.

Figures 5.16 and 5.17 show similar performance comparisons, except to an accuracy level of 10^9. The autotuner had a more difficult time beating the reference full multigrid algorithm when

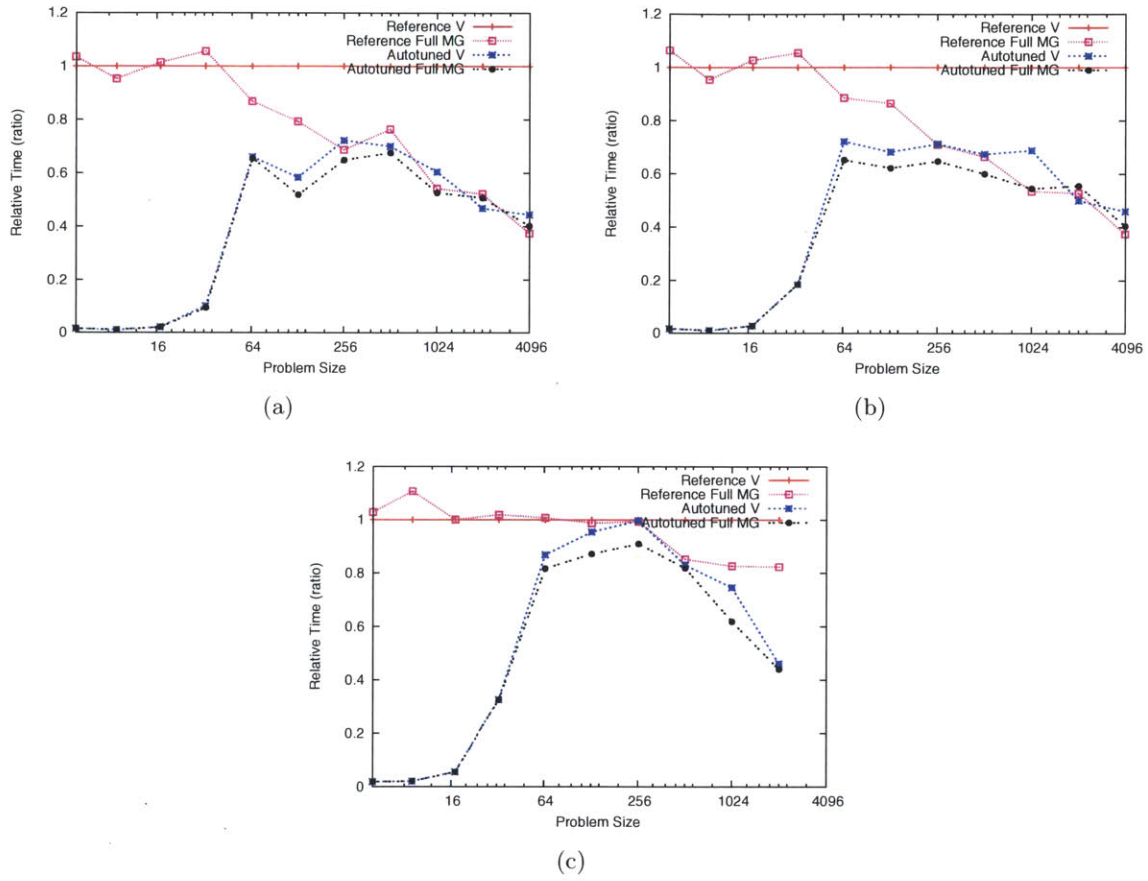


Figure 5.17: Relative performance of multigrid algorithms versus reference V cycle algorithm for solving the 2D Poisson's equation on biased, uniform random data to an accuracy level of 10^9 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

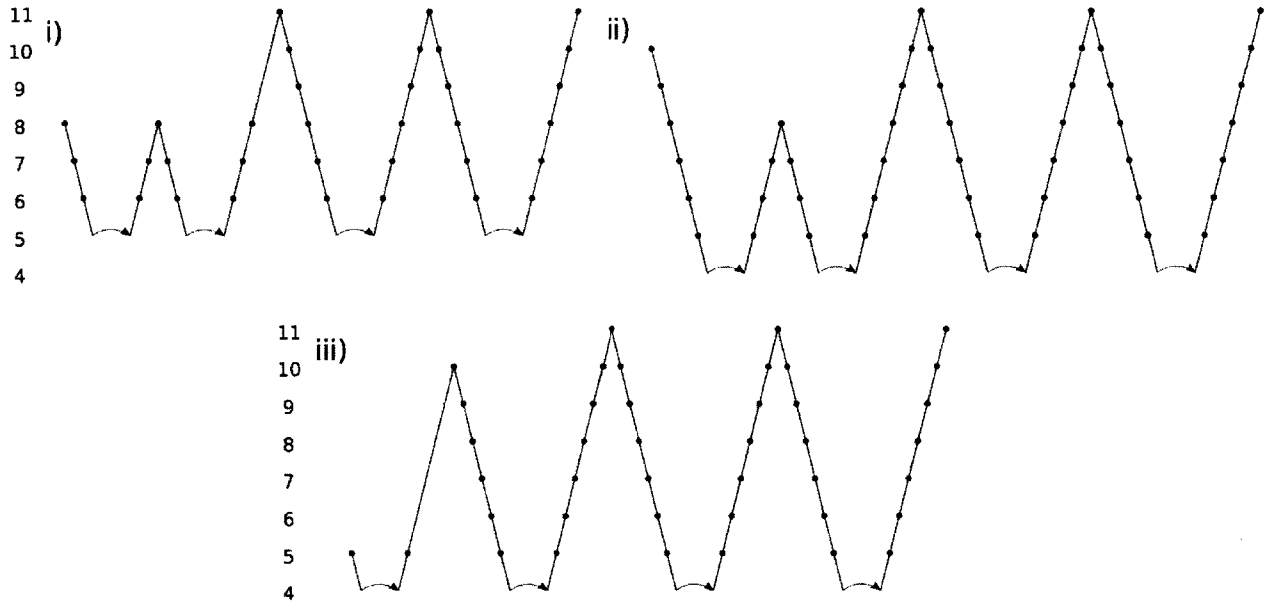


Figure 5.18: Comparison of tuned full multigrid cycles across machine architectures: i) Intel Harpertown, ii) AMD Barcelona, iii) Sun Niagara. All cycles solve the 2D Poisson's equation on unbiased uniform random input to an accuracy of 10^5 for an initial grid size of 2^{11}.

training for both high accuracy and large size (greater than $N = 257$). For sizes greater than 257, autotuned performance is essentially tied with the reference full multigrid algorithm on the Intel and AMD machines, while improvements were still possible on the Sun machine. For input size $N = 2049$, a speedup of 1.9x relative to the reference full multigrid algorithm was observed on the Niagara for both input distributions. We suspect that performance gains are more difficult to achieve when solving for both high accuracy and size in some part due to a greater percentage of compute time being spent on unavoidable relaxations at the finest grid resolution.

5.2.3 Effect of Architecture on Autotuning

		Trained on	
		Xeon 8-way	Niagara
Run on	Xeon 8-way	-	1.29x
	Niagara	1.79x	-

Figure 5.19: Slowdown when trained on a setup different than the one run on for a 1.2 GHz Sun Fire T200 Niagara and a 2.4 GHz Xeon E7340 (2 x 4 core) system. Slowdowns are relative to training natively. Autotuned full Poisson 2D multigrid cycle for unbiased uniform inputs of size $N = 2049$.

Figure 5.18 shows the different optimized cycles chosen by the autotuner on the three testbed architectures. Though all cycles were tuned to yield the same accuracy level of 10^5, the autotuner found a different optimized cycle shape on each architecture. These differences take advantage of the specific characteristics of each machine. For example, the AMD and Sun machines recurse down to a coarse grid level of 2^4 versus 2^5 on the Intel machine. The AMD and Sun's cycles appear to make up for the reduced accuracy of the coarser direct solve by doing more relaxations at medium grid resolutions (levels 9 and 10).

We found that the performance of tuned multigrid cycles can be quite sensitive to the type of system the autotuning is performed on. For example, Figure 5.19 shows slowdowns when trained/running on two different systems: the Sun Niagara 1 and an Intel Xeon. The table shows up to a 79% slowdown for using a V-cycle shape generated on a different machine.

Chapter 7 will provide further analysis of our multigrid benchmarks in the context of adapting to different inputs. We find that different multigrid cycle shapes are best for different inputs.

Chapter 6

The PetaBricks Autotuner

Petabricks not only relies on the autotuner to create hybrid algorithms from user provided algorithmic choice, but also moves most of the compiler optimization decisions to the autotuner. Thus, PetaBricks relies heavily on having an autotuner that is capable of searching extremely large spaces in an efficient manner. This chapter will describe how PetaBricks autotuner is able to achieve these goals. The PetaBricks autotuner uses at its core a novel evolutionary algorithm, INCREA, to produce new candidate algorithms and guide the search.

In this chapter, we present an evolutionary algorithm, INCREA, which is designed to incrementally solve a large, noisy, computationally expensive problem by deriving its initial population through recursively running itself on problem instances of smaller sizes. The INCREA algorithm also expands and shrinks its population each generation and cuts off work that does not appear to promise a fruitful result. For further efficiency, it addresses noisy solution quality efficiently by focusing on resolving it for small, potentially reusable solutions which have a much lower cost of evaluation. We compare INCREA to a general purpose evolutionary algorithm and find that in most cases INCREA arrives at the same solution in significantly less time.

This evolutionary algorithm is at the core of the PetaBricks autotuner. The genome for the evolutionary algorithm is the search space encoded by the PetaBricks compiler and the PetaBricks autotuner evaluates fitness by running candidate programs on representative inputs.

6.1 The Autotuning Problem

The autotuner must identify selectors that will determine which choice of an algorithm will be used during a program execution so that the program executes as fast as possible. 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

$$c_{s,0} = min(size(input)) \text{ and } c_{s,m} = max(size(input)).$$

The components of $\vec{A_s}$ are indices into a discrete set of applicable algorithms available to s, which we denote $Algorithms_s$. The maximum number of intervals is fixed by the PetaBricks compiler. An example of a selector for a sample sorting algorithm is shown in Figure 6.1.

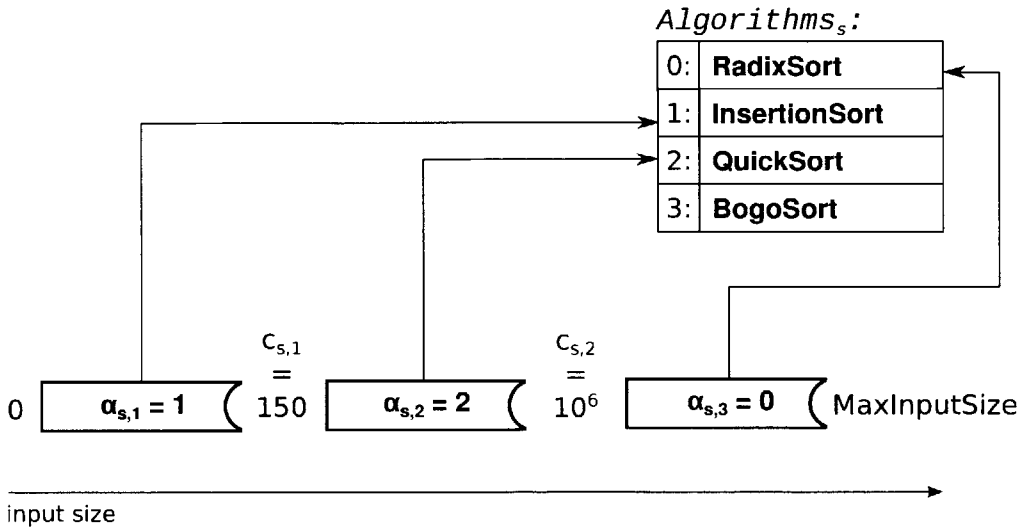


Figure 6.1: A selector for a sample sorting algorithm where $\vec{C_s} = [150, 10^6]$ and $\vec{A_s} = [1, 2, 0]$. The selector selects the *InsertionSort* algorithm for input sizes in the range $[0; 150)$, *QuickSort* for input sizes in the range $[150, 10^6)$ and *RadixSort* for $[10^6, MAXINT)$. *BogoSort* was suboptimal for all input ranges and is not used.

In addition to algorithmic choices, the autotuner tunes parameters such as blocking sizes, sequential/parallel cutoffs and the number of worker threads. Each tunable is either a discrete value of a small set indexed by an integer or a integer in some positive bounded range.

Formally, given a program P, hardware H and input size n, the autotuner must identify the vector of selectors and vector of tunables such that the following objective function *executionTime* is satisfied:

$$\arg\min_{\bar{s},t} \text{executionTime}(P, H, n)$$

6.1.1 Properties of the Autotuning Problem

Three properties of autotuning influence the design of an autotuner. First, the cost of fitness evaluation depends heavily on on the input data size used when testing the candidate solution. The autotuner does not necessarily have to use the target input size. For efficiency it could use smaller sizes to help it find a solution to the target size because is generally true that smaller input sizes are cheaper to test on than larger sizes, though exactly how much cheaper depends on the algorithm. For example, when tuning matrix multiply one would expect testing on a 1024×1024 matrix to be about 8 times more expensive than a 512×512 matrix because the underlying algorithm has $O(n^3)$ performance. While solutions on input sizes smaller than the target size sometimes are different from what they would be when they are evolved on the target input size, it can generally be expected that relative rankings are robust to relatively small changes in input size. This naturally points to "bottom-up" tuning methods that incrementally reuse smaller input size tests or seed them into the initial population for larger input sizes.

Second, in autotuning the fitness of a solution is its fitness evaluation cost. Therefore the cost of fitness evaluation is dependant on the quality of a candidate algorithm. A highly tuned and optimized program will run more quickly than a randomly generated one and it will thus be fitter. This implies that fitness evaluations become cheaper as the overall fitness of the population improves.

Third, significant to autotuning well is recognizing the fact that fitness evaluation is noisy due to details of the parallel micro-architecture being run on and artifacts of concurrent activity in the operating system. The noise can come from many sources, including: caches and branch prediction; races between dependant threads to complete work; operating system artifacts such as scheduling, paging, and I/O; and, finally, other competing load on the system. This leads to a design conflict: an autotuner can run fewer tests, risking incorrectly evaluating relative performance but finishing quickly, or it can run many tests, likely be more accurate but finish too slowly. An appropriate strategy is to run more tests on less expensive (i.e. smaller) input sizes.

The INCREA exploits incremental structure and handles the noise exemplified in autotuning. We now proceed to describe a INCREA for autotuning.

6.2 A Bottom Up EA for Autotuning

Representation

The INCREA genome, see Figure 6.2, encodes a list of selectors and tunables as integers each in the range $[0, MaxVal)$ where $MaxVal$ is the cardinality of each algorithm choice set for algorithms and $MaxInputSize$ for cutoffs. Each tunable has a $MaxVal$ which is the cardinality of its value set or a bounded integer depending on what it represents.

In order to tune programs of different input sizes the genome represents a solution for maximum input size and throughout the run increases the "active" portion of it starting from the selectors and tunables relevant to the smallest input size. It has length $(2m + 1)k + n$, where k is the number of selectors, m the number of interval cutoffs within each selector and n the number of other tunables defined for the PetaBricks program. As the algorithm progresses the number of "active" cutoff and algorithm pairs, which we call "choices" for each selector in the genome starts at 1 and then is incremented in step with the algorithm doubling the current input size each generation.

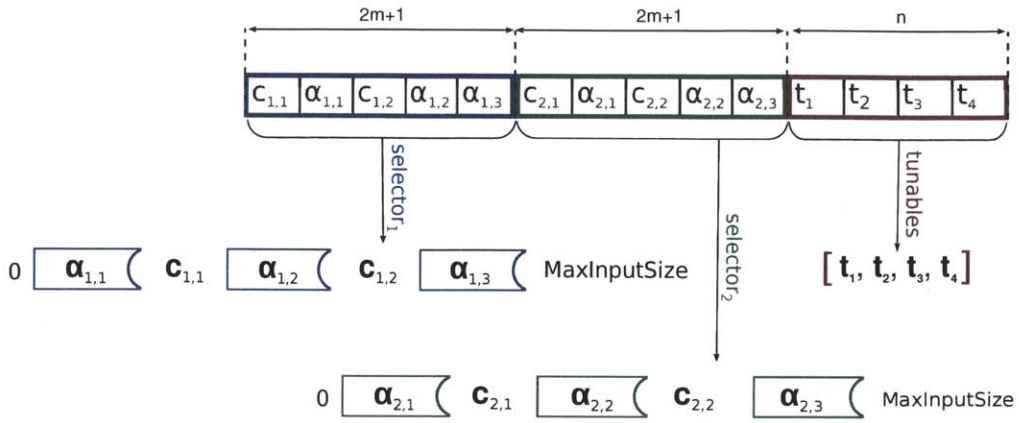


Figure 6.2: A sample genome for $m = 2$, $k = 2$ and $n = 4$. Each gene stores either a cutoff $c_{s,i}$, an algorithm $\alpha_{s,i}$ or a tunable value t_i.

Fitness evaluation

The fitness of a genome is the inverse of the corresponding program's execution time. The execution time is obtained by timing the PetaBricks program for a specified input size.

Top level Strategy

Figure 6.3 shows top level pseudocode for INCREA. The algorithm starts with a "parent" population and an input size of 1 for testing each candidate solution. All choices and tunables are initially set to algorithm 0 and cutoff of MAX_INT. The choice set is grown through mutation on a per candidate basis. The input size used for fitness evaluation doubles each generation.

A generation consists of 2 phases: exploration, and downsizing. During exploration, a random parent is used to generate a child via mutation. Only active choices and tunables are mutated in this process. The choice set may be enlarged. The child is added to the population only if it is determined to be fitter than its parent. The function "fitter" which tests for this condition increases trials of the parent or child to improve confidence in their relative fitnesses. Exploration repeatedly generates a child and tests it against its parent for some fixed number of MutationAttempts or until the population growth reaches some hard limit.

During downsizing, the population, which has potentially grown during exploration, is pruned down to its original size once it is ranked. The "rankThenPrune" function efficiently performs additional fitness tests only as necessary to determine a ranking of which it is reasonably certain.

This strategy is reminiscent but somewhat different from a $(\mu + \lambda)$ES [23]. The $(\mu + \lambda)$ES creates a pool of λ offspring each generation by random draws from the parent population of size μ. Then both offspring and parents are combined and ranked for selection into the next generation. The subtle differences in INCREA are that 1) in a "steady state" manner, INCREA inserts any child which is better than its parent immediately into the population while parents are still being drawn, and 2) a child must be fitter than its parent before it gains entry into the population. The subsequent ranking and pruning of the population matches the selection strategy of $(\mu + \lambda)$ES.

Doubling the input size used for fitness evaluation at each generation allows the algorithm to learn good selectors for smaller ranges before it has to find ones for bigger ranges. It supports subsolution reuse and going forward from a potentially good, non-random starting point. Applying mutation to only the active choice set and tunables while input size is doubling brings additional efficiency because this narrows down the search space while concurrently saving on the cost of fitness evaluations because testing solutions on smaller inputs sizes is cheaper.

Mutation Operators

The mutators perform different operations based on the type of value being mutated. For an algorithmic choice, the new value is drawn from a uniform probability distribution $[0, ||Algorithms_s|| - 1]$. For a cutoff, the existing value is scaled by a random value drawn from a log-normal distribution, i.e. doubling and halving the existing value are equally likely. The intuition for a log-normal distribution is that small changes have larger effects on small values than large values in autotuning. We have confirmed this intuition experimentally by observing much faster convergence times with this type of scaling.

The INCREA mutation operator is only applied to choices that are in the active choice list for the genome. INCREA has one specialized mutation operator that adds another choice to the active choice list of the genome and sets the cutoff to 0.75 times the current input size while choosing

```
populationSize = popLowSize
inputSizes = [1, 2, 4, 8, 16, ...., maxInputSize]
initialize population(maxGenomeLength)
for gen = 1 to log(maxInputSize)
  /* exploration phase: population and active
     choices may increase */
  inputSize = inputSizes[gen]
  for j = 1 to mutationAttempts
    parent = random draw from population
    activeChoices = getActiveChoices(parent)
    /* active choices could grow */
    child = mutate(parent, activeChoices)
    /* requires fitness evaluations */
    if fitter(child, parent, inputSize)
      population = add(population, child)
      if length(population) >= popHighSize
        exit exploration phase
  end /* exploration phase */
  /* more testing */
  population = rankThenPrune(population,
                             popLowSize,
                             inputSize)
  /* discard all past fitness evaluations */
  clearResults(population)
end /* generation loop*/
return fittest population member
```

Figure 6.3: Top level strategy of INCREA.

the algorithm randomly. This leaves the behavior for smaller inputs the same, while changing the behavior for the current set of inputs being tested. It also does not allow a new algorithm to be the same as the one for the next lower cutoff.

Noisy Fitness Strategies

Because INCREA must also contend with noisy feedback on program execution times, it is bolstered to evaluate candidate solutions multiple times when it is ranking any pair. Because care must be taken not to test too frequently, especially if the input data size is large, it uses an adaptive sampling strategy [3, 34, 36, 138] . The boolean function "fitter", see Figure 6.4, takes care of this concern by running more fitness trials for candidates $s1$ and $s2$ under two criteria. The first criterion is a t-test [99]. When the t-test result has a confidence, i.e. p-value less than 0.05, $s1$ and $s2$ are considered different and trials are halted. If the t-test cannot confirm difference, least squares is used to fit a normal distribution to the percentage difference in the mean execution time of the two algorithms. If this distribution estimates there is a 95% probability of less than a 1% difference, the two candidates' fitnesses are considered to be the same. There is also a parameterized hard upper limit on trials.

The parent ranking before pruning, in function "rankThenPrune", is optimized to minimize the number of additional fitness evaluations. First, it ranks the entire population by mean performance without running any additional trials. It then splits the ranking at the *populationLowSize* element into a $KEEP$ list and a $DISCARD$ list. Next, it sorts the $KEEP$ list by calling the "fitter" function (which may execute more fitness trials). Next, it compares each candidate in the $DISCARD$ list to the *populationLowSize* element in the $KEEP$ list by calling the "fitter" function. If any of these candidates are faster, they are moved to the $KEEP$ list. Finally, the $KEEP$ list is sorted again by calling "fitter" and the first *populationLowSize* candidates are the result of the pruning.

This strategy avoids completely testing the elements of the population that will be discarded. It allocates more testing time to the candidate that will be kept in the population. It also exploits the

```
1 function fitter(s1, s2, inputSize)
2   while s1.evalCount < evalsLowerLimit
3     evaluateFitness(s1, inputSize)
4   end
5   while s2.evalCount < evalsLowerLimit
6     evaluateFitness(s2, inputSize)
7   end
8   while true
9     /* Single tailed T-test assumes each sample's mean is
10       normally distributed.  It reports probability that
11       sample means are same under this assumption. */
12    if ttest(s1.evalsResults, s2.evalResults) < PvalueLimit
13      /* statistically different */
14      return mean(s1.evalResults) > mean(s2.evalResults)
15    end
16    /* Test2Equality: Use least squares to fit a normal
17       distribution to the percentage difference in the
18       mean performance of the two algorithms.  If this
19       distribution estimates there is a 95% probability
20       of less than a 1% difference in true means, consider
21       the two algorithms the same. */
22    if Test2Equality(s1.evalResults, s2.evalResults)
23      return false
24    end
25    /* need more information, choose s1 or s2 based on
26       the highest expected reduction in standard error */
27    whoToTest = mostInformative(s1, s2);
28    if whoToTest == s1 and s1.testCount < evalsUpperLimit
29      evaluateFitness(s1, inputSize)
30    elif s2.testCount < evalsUpperLimit
31      evaluateFitness(s2, inputSize)
32    else
33      /* inconclusive result, no more evals left */
34      return false
35    end
36  end /* while */
37 end /* fitter */
```

Figure 6.4: Pseudocode of function "fitter".

fact that comparing algorithms with larger differences in performance is cheaper than comparing algorithms with similar performance.

6.3 Experimental Evaluation

We now compare INCREA to a general purpose EA we call GPEA on 4 Petabricks benchmarks described in Chapter 4: sort (for 2 target input sizes), matmult which is dense matrix multiply, and eig which solves for symmetric eigenvalues.

6.3.1 GPEA

The GPEA uses the same genome representation and operators of INCREA. All selector choices are always active. It initializes all population members with values drawn from the distributions used by the mutation operator. It then loops evaluating the fitness of each member once, performing tournament selection and applying crossover with $p_{xo} = 1.0$ then mutation with probability p_{μ}. Crossover swaps algorithms while cutoffs and tunables are swapped or changed to a random value in between those of the two parents' genes. Extrapolating from [18,33], GPEA's significant population size (100 in our experiments) should provide some robustness to fitness case noise.

6.3.2 Experimental Setup

Parameter	Value
confidence required	70%
max trials	5
min trials	1
population high size	10
population low size	2
mutationAttempts	6
standard deviation prior	15%

(a) INCREA

Parameter	Value
mutation rate	0.5
crossover rate	1.0
population size	100
tournament size	10
generations	100
evaluations per candidate	1

(b) GPEA

Figure 6.5: INCREA and GPEA Parameter Settings.

We performed all tests on multiple identical 8-core, dual-Xeon X5460, systems clocked at 3.16 GHz with 8 GB of RAM. The systems were running Debian GNU/Linux 5.0.3 with kernel version 2.6.26. For each test, we chose a target input size large enough to allow parallelism, and small enough to converge on a solution within a reasonable amount of time. Parameters such as the mutation rate, population size and the number of generations were determined experimentally and kept constant between benchmarks. Parameter values we used are listed in Figure 6.5.

6.3.3 INCREA vs GPEA

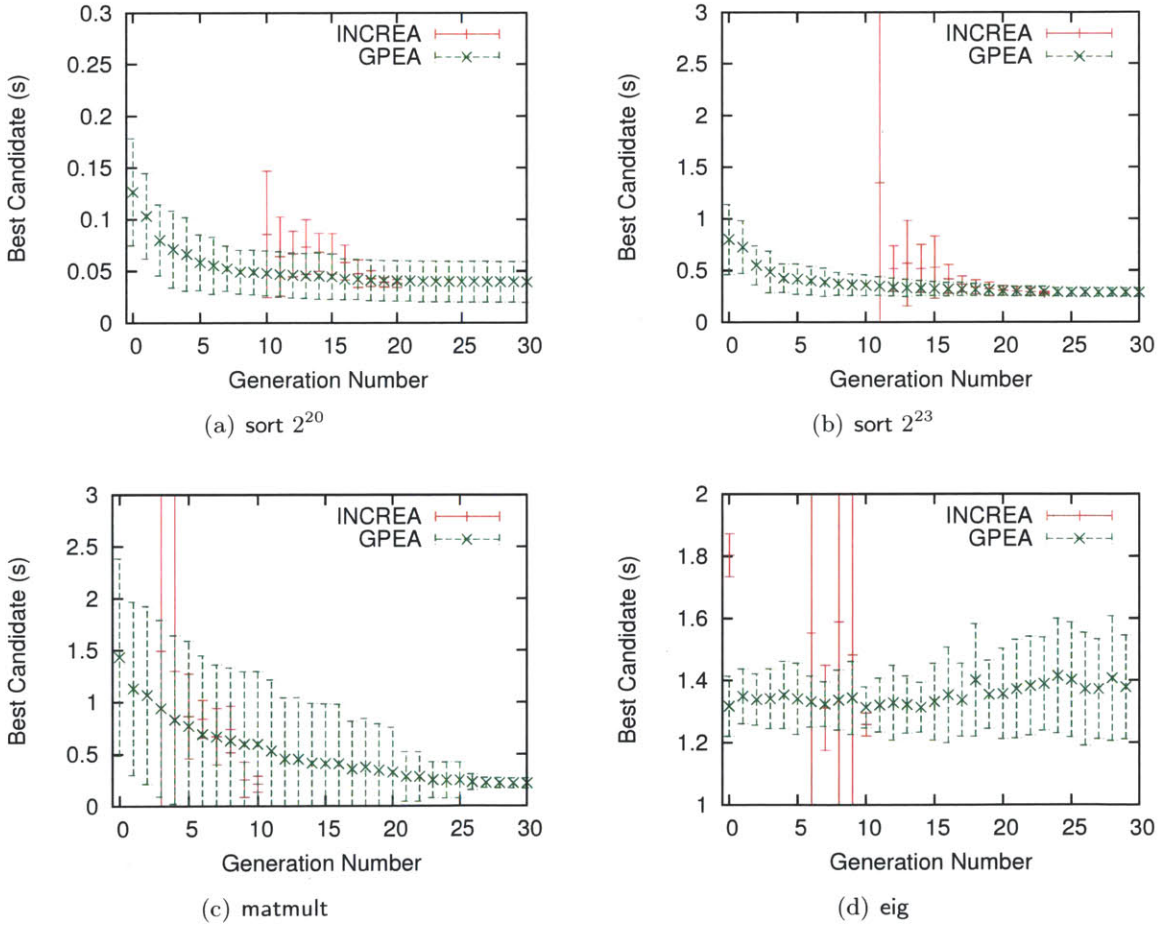
In practice we might choose parameters of either INCREA or GPEA to robustly ensure good autotuning or allow the programmer to vary them while tuning a particular problem and architecture. In the latter case, considering how quickly the tuner converges to the final solution is important. To more extensively compare the two tuners, we ran each tuner 30 times for each benchmark.

Table 6.1 compares the tuners mean performance with 30 runs based on time to convergence and the performance of the final solution. To account for noise, time to convergence is calculated as the first time that a candidate was found that was within 5% of the best fitness achieved. For all of the benchmarks except for eig, both tuners arrive at nearly the same solutions, while for eig INCREA finds a slightly better solution. For eig and matmult, INCREA converges an order of magnitude faster than GPEA. For sort, GPEAconverges faster on the small input size while INCREA converges faster on the larger input size. If one extrapolates convergences times to larger input sizes, it is clear that INCREA scales a lot better than GPEA for sort.

Figure 6.6 shows aggregate results from 30 runs for both INCREA and GPEA on each benchmark. INCREA generally has a large amount of variance in its early generations, because those generations are based on smaller input sizes that may have different optimal solutions than the largest input size. However, once INCREA reaches its final generation it exhibits lower variance than than GPEA. GPEA tends to converge slowly with gradually decreasing variance. Note that the first few generations for INCREA are not shown because, since it was training on extremely small input sizes, it finds candidates candidates that exceed the timeout set by our testing framework

		INCREA	GPEA	SS?
sort-2^{20}	Convergence	1464.7 ± 1992.0	599.2 ± 362.9	YES ($p = 0.03$)
	Performance	0.037 ± 0.004	0.034 ± 0.014	NO
sort-2^{23}	Convergence	2058.2 ± 2850.9	2480.5 ± 1194.5	NO
	Performance	0.275 ± 0.010	0.276 ± 0.041	NO
matmult	Convergence	278.5 ± 185.8	2394.2 ± 1931.0	YES ($p = 10^{-16}$)
	Performance	0.204 ± 0.001	0.203 ± 0.001	NO
eig	Convergence	92.1 ± 66.4	627.4 ± 530.2	YES ($p = 10^{-15}$)
	Performance	1.240 ± 0.025	1.250 ± 0.014	YES ($p = 0.05$)

Table 6.1: Comparison of INCREA and GPEA in terms of mean time to convergence in seconds and in terms of execution time of the final configuration. Standard deviation is shown after the $\pm$ symbol. The final column is statistical significance determined by a t-test. (Lower is better)



(a) sort 2^{20}

(b) sort 2^{23}

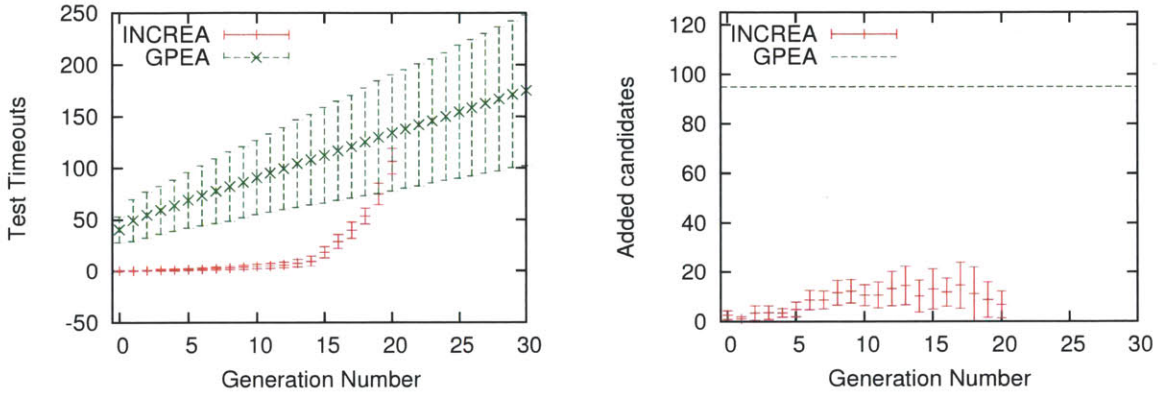
(c) matmult

(d) eig

Figure 6.6: Execution time for target input size with best individual of generation. Mean and standard deviation (shown in error bars) with 30 runs.

when run on the largest input size. These early generations account for a only a small amount of the total training time.

In 6.6(a) the INCREA's best candidate's execution time displays a "hump" that is caused because it finds optima for smaller input sizes that are not reused in the optimal solution for the target input size.



(a) Tests Halted due to Time Out

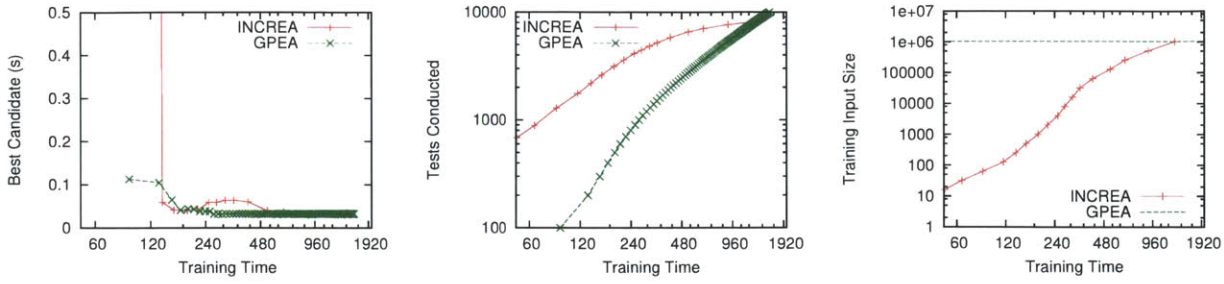
(b) Population Growth

Figure 6.7: Time out and population growth statistics of INCREA for 30 runs of sort on target input size 2^{20}. Error bars are mean plus and minus one standard deviation.

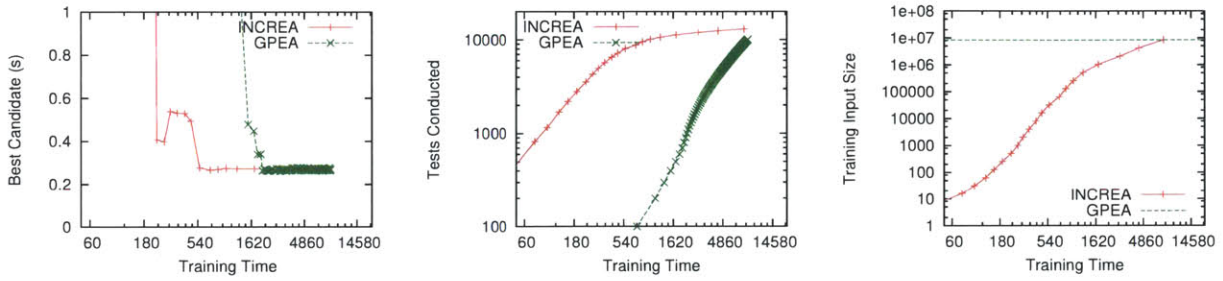
Using sort-2^{20}, in Figure 6.7(a) we examine how many tests are halted by each tuner, indicating very poor solutions. The timeout limit for both algorithms is set to be the same factor of the time of the current best solution. However, in GPEA this will always be a test with the target input size whereas with INCREA it is the current input size (which is at least half the time, half as large). Almost half of GPEA's initial population were stopped for timing out, while INCREA experiences most of its timeouts in the later generations where the difference between good and bad solutions grows with the larger input sizes. We also examine in Figure 6.7(b) how much the population grew each generation during the exploration phase. For INCREA the population expansion during exploration is larger in the middle generations as it converges to a final solution.

6.3.4 Representative runs

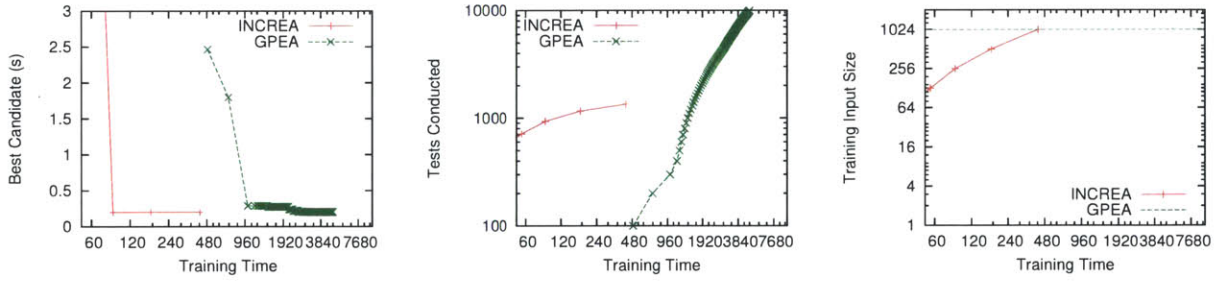
We now select a representative run for each benchmark to focus on run dynamics.



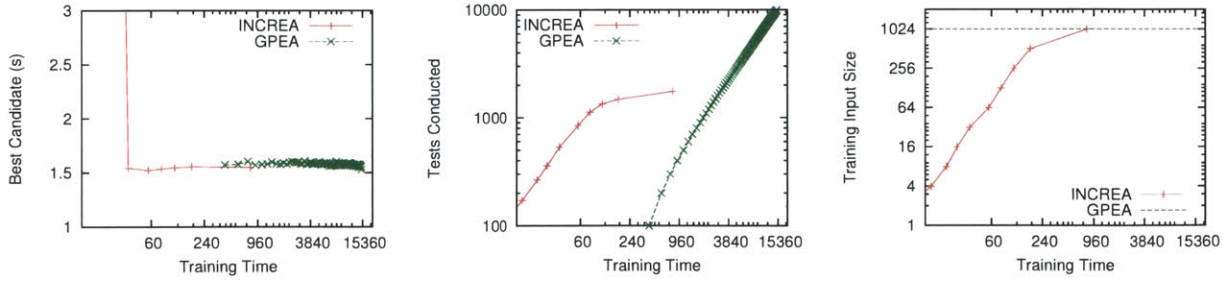
(a) sort 2^{20}



(b) sort 2^{23}



(c) matmult 1024×1024



(d) eig 1024×1024

Figure 6.8: Representative runs of INCREA and GPEA on each benchmark. The left graphs plot the execution time (on the target input size) of the best solution after each generation. The right graph plots the number of fitness evaluations conducted at the end of each generation. All graphs use seconds of training time as the x-axis.

sort: Sorting

Figures 6.8(a) and 6.8(b) show results from a representative run of each autotuner with two different target input sizes respectively. The benchmark consists of insertion-sort, quick-sort, radix sort, and 2/4/8/16/32-way merge-sorts. On this Xeon system, sort is relatively easy to tune because the optimal solution is relatively simple and the relative costs of the different algorithms are similar.

For the 2^{20} benchmark, both INCREA and GPEA consistently converge to a very similar solution which consists of small variations of quick-sort switching to insertion-sort at somewhere between 256 and 512. Despite arriving at a similar place, the two tuners get there in a very different way. Table 6.2, lists the best algorithm for each tuner at each generation in the run first shown in Figure 6.8(a). INCREA starts with small input sizes, where insertion-sort alone performs well, and for generations 0 to 7 is generating algorithms that primarily use insertion-sort for the sizes being tested. From generations 8 to 16, it creates variants of radix-sort and quicksort that are sequential for the input sizes being tested. In generation 17 it switches to a parallel quick sort and proceeds to optimize the cutoff constants on that for the remaining rounds. The first two of these major phases are locally optimal for the smaller input sizes they are trained on.

GPEA starts with the best of a set of random solutions, which correctly chooses insertion-sort for small input sizes. It then finds, in generation 3, that quick-sort, rather than the initially chosen radix-sort, performs better on large input sizes within the tested range. In generation 6, it refines its solution by paralellizing quick-sort. The remainder of the training time is spent looking for the exact values of the algorithmic cutoffs, which converge to their final values in generation 29.

We classified the possible mutation operations of INCREA and counted how frequently each was used in creating an offspring fitter than its parent. We identified specialized classes of operations that target specific elements of the genome. Table 6.3 lists statistics on each for the run first shown in Figure 6.8(a). The class most likely to generate an improved child scaled both algorithm and parallelism cutoffs. The class that changed just algorithms were less likely to cause improvement. Overall only 3.7% of mutations improved candidate fitness.

INCREA: sort		
Input size	Training Time (s)	Genome
2^0	6.9	Q 64 Q_p
2^1	14.6	Q 64 Q_p
2^2	26.6	I
2^3	37.6	I
2^4	50.3	I
2^5	64.1	I
2^6	86.5	I
2^7	115.7	I
2^8	138.6	I 270 R 1310 R_p
2^9	160.4	I 270 Q 1310 Q_p
2^{10}	190.1	I 270 Q 1310 Q_p
2^{11}	216.4	I 270 Q 3343 Q_p
2^{12}	250.0	I 189 R 13190 R_p
2^{13}	275.5	I 189 R 13190 R_p
2^{14}	307.6	I 189 R 17131 R_p
2^{15}	341.9	I 189 R 49718 R_p
2^{16}	409.3	I 189 R 124155 M^2
2^{17}	523.4	I 189 Q 5585 Q_p
2^{18}	642.9	I 189 Q 5585 Q_p
2^{19}	899.8	I 456 Q 5585 Q_p
2^{20}	1313.8	I 456 Q 5585 Q_p

GPEA: sort		
Gen	Training Time (s)	Genome
0	91.4	I 448 R
1	133.2	I 413 R
2	156.5	I 448 R
3	174.8	I 448 Q
4	192.0	I 448 Q
5	206.8	I 448 Q
6	222.9	I 448 Q 4096 Q_p
7	238.3	I 448 Q 4096 Q_p
8	253.0	I 448 Q 4096 Q_p
9	266.9	I 448 Q 4096 Q_p
10	281.1	I 371 Q 4096 Q_p
11	296.3	I 272 Q 4096 Q_p
12	310.8	I 272 Q 4096 Q_p
...		
27	530.2	I 272 Q 4096 Q_p
28	545.6	I 272 Q 4096 Q_p
29	559.5	I 370 Q 8192 Q_p
30	574.3	I 370 Q 8192 Q_p
...		

Table 6.2: Listing of the best genome of each generation for each autotuner for an example training run. The genomes are encoded as a list of algorithms (represented by letters), separated by the input sizes at which the resulting program will switch between them. The possible algorithms are: I = insertion-sort, Q = quick-sort, R = radix-sort, and M^x = x-way merge-sort. Algorithms may have a $_p$ subscript, which means they are run in parallel with a work stealing scheduler. For clarity, unreachable algorithms present in the genome are not shown.

Mutation Class	Count	Times Tried	Effect on fitness		
			Positive	Negative	None
Make an algorithm active	8	586	2.7%	83.8%	13.5%
Lognormally scale a cutoff	11	1535	4.4%	50.4%	45.1%
Randomy switch an algorithm	12	1343	2.5%	50.4%	25.7%
Lognormally change a parallism cutoff	2	974	5.2%	38.7%	56.1%

Table 6.3: Effective and ineffective mutations when INCREA solves sort (target input size 2^{20}.)

matmult: Dense Matrix Multiply

Figure 6.8(c) shows comparative results on matmult. The program choices are a naive matrix multiply and five different parallel recursive decompositions, including Strassen's Algorithm and a cache-oblivious decomposition. A tunable allows both autotuners to transpose combinations of inputs and outputs to the problem. To generate a valid solution, the autotuner must learn to put a base case in the lowest choice of the selector, otherwise it will create an infinite loop. Because many random mutations will create candidate algorithms that never terminate when tested, we impose a time limit on execution.

Both INCREA and GPEA converge to the same solution for matmult. This solution consists of transposing the second input and then doing a parallel cache-oblivious recursive decomposition down to 64×64 blocks which are processed sequentially.

While both tuners converge to same solution, INCREA arrives at it much more quickly. This is primarily due to the n^3 complexity of matrix multiply, which makes running small input size tests extremely cheap compared to larger input sizes and the large gap between fast and slow configurations. INCREA converges to the final solution in 88 seconds, using 935 trials, before GPEA has evaluated even 20% of its initial population of 100 trials. INCREA converges to a final solution in 9 generations when the input size has reached 256, while GPEA requires 45 generations at input size 1024. Overall, INCREA converges 32.8 times faster than GPEA for matrix multiply.

eig: Symmetric Eigenproblem

Figure 6.8(d) shows results for eig. Similar to matmult here INCREA performs much better because of the fast growth in cost of running tests. This benchmark is unique in that its timing results have higher variance due to locks and allocation in the underlying libraries used to implement certain mathematical functions. This high variance makes it difficult to autotune well, especially for GPEA which only runs a single test for each candidate algorithm. Both solutions found were of same structure, but INCREA was able to find better cutoffs values than the GPEA.

Chapter 7

Input Sensitivity

A fundamental problem that autotuning systems face is input sensitivity. For a large class of problems, the best optimization to use depends on the input data being processed. For example, sorting an almost-sorted list can be done most efficiently with a different algorithm than one optimized for sorting random data. In empirical autotuning systems, there is a danger that the auotuner will create an algorithm specifically optimized for the inputs it is provided during tuning. This may be suboptimal for inputs later encountered in production or be a compromise solution that is not best for any one input but performs well overall. For many problems, no single optimized program exists which can match the performance of a collection of optimized programs autotuned for different subsets of the input space.

This problem of input sensitivity is exacerbated by several features common to many classes of problems and types of autotuning systems. Many autotuning systems must handle large search spaces and variable accuracy algorithms with multiple objectives. They encounter inputs with non-superficial features that require domain specific knowledge to extract, Each of these challenges makes the problem of input sensitivity more difficult in a unique way.

The first challenge is the size of the optimization space. When it is small, an autotuner can exhaustively try all possible configurations on a large subset of inputs to characterize and resolve input sensitivity. However, exhaustive search does not scale to larger, complex search spaces. In the benchmarks we consider, the autotuner uses algorithmic choices embedded in the program to

construct arbitrary polyalgorithms which process a single input through a hybrid of many different individual techniques. This results in enormous search spaces, ranging from 10^{312} to 10^{1016} possible configurations of a program. These search spaces are far too large to search exhaustively and demand new techniques for dealing with input sensitivity.

The second challenge is that the performance of different algorithmic configurations may be sensitive to many input features that are domain-specific and require deep, possibly expensive, analysis to extract. Two inputs which superficially look similar may respond very differently to the same optimization. For example, our singular value decomposition benchmark is sensitive to the number of eigenvalues in the input matrix. This is not reflected in a generic feature such as input size. A complete solution to this problem must address both how to express and extract such domain specific features and how to do so in a cost effective manner. If the overhead of extracting a feature is too large, it may not be worth using. One must further weigh whether extracting another feature, somewhat correlated to existing ones, with the runtime cost of extracting it.

The third challenge is interaction with variable accuracy algorithms. Many programs can produce outputs of varying quality, and the autotuner is required to produce configurations that will meet a target quality of service level. For example, our Poisson's equation solver benchmark must produce an output that matches the output of a direct solver to seven digits of precision with at least 90% confidence. Meeting such a requirement is especially difficult, because the difficulty level of different inputs can vary. For a class of inputs, a very fast polyalgorithm may suffice to achieve seven digits of accuracy, while for different inputs that same solver may not meet the accuracy target. A system that is not input aware is forced to use a more expensive algorithm that achieves the target across all inputs, and can not tailor different algorithms for different inputs.

This chapter presents a general means of automatically determining what algorithmic optimization to use when different ones suit different inputs. While input sensitivity seems to be intertwined with the complexity of large optimization spaces and input spaces, we show that it can be resolved via simple extensions to an existing autotuning system. We show that the complexity of input sensitivity can be managed, and that a small number of input optimized programs is often sufficient to get most of the benefits of input adaptation. The language keyword `input_feature`

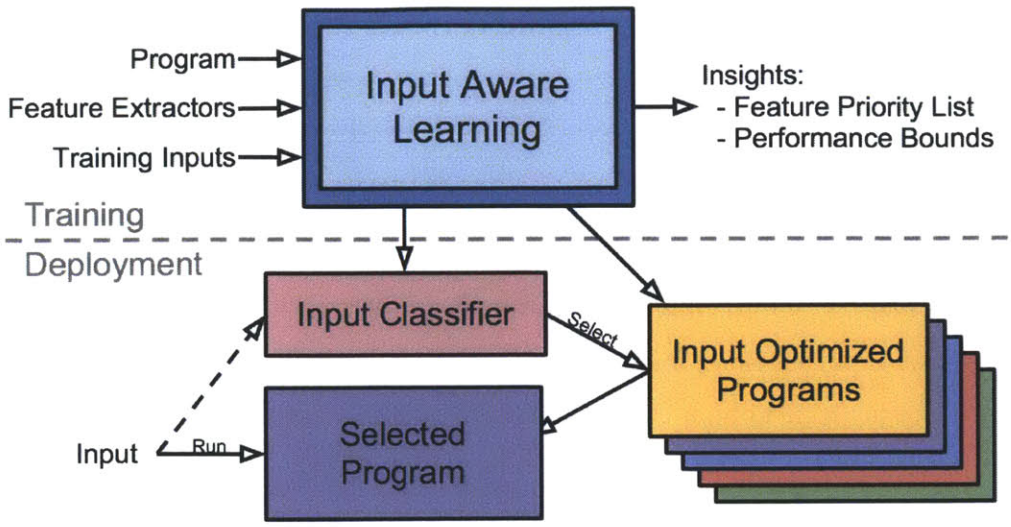


Figure 7.1: Usage of the system both at training time and deployment time. At deployment time the input classifier selects the input optimized program to handle each input. Input Aware Learning is described in Section 7.2.

(discussed in Section 2.4) is used to define the domain specific input features referenced in this chapter.

We propose a two level approach to solving the input sensitivity problem for complex algorithmic search spaces. At the first level we cluster the input space into a finite number of *input classes*. For each of these input classes, we use an evolutionary autotuner to create an optimized program configuration. We then train a large number of classifiers, each of which relies upon a varying number of input features, to assign each new input to one of the optimized program configurations created for one of the *input classes*. At the second level, prior to the execution of a program, given an input to process, we select among these candidate classifiers by means of competition so that one *production classifier* is chosen. This two level approach is able to achieve large speedups by making an optimization choice which is sensitive to input variation while managing the input space and search space complexity. We also propose a new language keyword that allows the programmer to specify arbitrary domain-specific input features with variable sampling levels.

7.1 Usage

Figure 7.1 describes the usage of our system for input sensitive algorithm design. At the first level, there is input aware learning which takes the user's program (containing algorithmic choices), the feature extractors specified by the `input_feature` language keyword (described in Chapter 2) and input exemplars as input. Input aware learning is described in Section 7.2. The output of the learning is an input classifier and a set of input optimized programs, each of which has been optimized for specific class of inputs.

When an arbitrary input is encountered in deployment, the classifier created by learning is used to select an input optimized program which is expected to perform the best on this input. The classifier will use some (possibly variable) subset of the feature extractors available to it to probe the input. Finally, the selected input optimized program will process the input and return the output to the user.

7.2 Input Aware Learning

In this section, for reference, we start by proposing a naive design of input aware learning. We use its design issues to convey the important considerations one needs to take when designing input aware learning for complex algorithmic autotuning. We then explain the two level approach we have developed.

7.2.1 A Simple Design and Its Issues

A straightforward way to construct an input classifier is via input-based clustering. First, construct feature vectors for every example input set with the `input_feature` extraction procedures encoded by the programmer. Then, cluster examples based on the feature vectors. Next, find a good algorithmic configuration for each cluster's centroid. For a new input, the classifier first invokes the feature extraction procedures to compute its feature vector, based on which, it finds out what input cluster the new input belongs, and then runs the configuration of that cluster on that new

input. This design has been used for addressing input sensitivity in program specialization and others [143]. However, applying it to algorithmic autotuning raises three issues.

First, it fails to acknowledge that two input sets that are similar may not have correspondingly similar configurations, and vice versa. As well, while there may be more than one configuration that suits an input set, but some will perform well on an input set similar to it, while others will not. In other words, there is no direct correspondence between similar input features, similar configurations and/or similar algorithm performance (measured in execution speed and accuracy). Instead the relationships among input properties, configurations and program behavior are non-linear and complex. We call this phenomena a *mapping disparity*. It implies that by assigning configurations based on the differences in input features, the simple design is likely to assign an inferior configuration for new input.

The second issue is that even if the configuration found by the simple classifier happens to provide the highest performance on that new input, its calculation accuracy may not meet the requirement. It is unclear how the simple design can handle accuracy-performance conflicts, a special complexity in algorithmic autotuning.

The third issue with the simple design is that it does not consider the overhead in feature extraction on the new input. Due to the complexity in algorithmic choice, some features may take a substantial time to extract. As the feature extraction occurs on the critical path of the program execution, the simple design may end up with a significant slowdown for the introduced extra work.

7.2.2 Design of the Two Level Learning

Motivated by the particular complexities of algorithmic autotuning, we develop a two level learning framework. The first level is shown in Figure 7.2. Like the simple design, in its first step it clusters and groups the input space into a finite number of input classes and then uses the autotuner to identify a good algorithmic configuration for each cluster's centroid. In addition however, to provide data on the mappings among inputs, configuration and performance, it executes every exemplar using the configuration of each cluster. These results will be used at the next level.

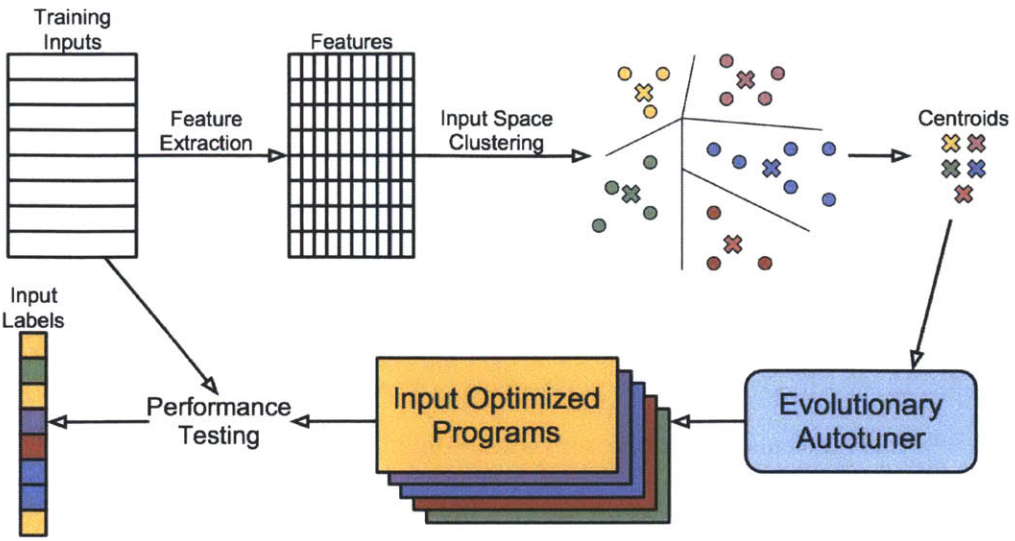


Figure 7.2: Selecting representative inputs to train input optimized programs.

The second level is shown in Figure 7.3. It addresses interpreting the mapping evidence previously collected on the inputs and their performance on a small set of "landmark" configurations taken from the centroids found earlier. It builds a large number of classifiers each different by which input features it references and/or different by the algorithm used to derive the classifier. It then computes an objective score, based on performance and feature extraction costs, for every classifier and selects the best one as the production classifier. Together, these two levels create an approach that is able to achieve large speedups by its sensitivity to input variation and configuration influence on program performance. We now provide detail on each level's design.

7.2.3 Level 1

The main objective of Level one is to identify a set of configurations for each class of inputs. We call these configurations "landmarks".

Specifically, there are four steps in this level of learning.

- **Step 1: Feature Extraction** We assemble a feature vector for each training input using the values computed by the *input_feature* procedures of the program. For each property, by using a tunable parameter such as `level` in the *input_feature* procedure in the program, we have collected values at z different costs which are what we call features.

- **Step 2: Input Clustering** We first normalize the input feature vectors to avoid biases imposed by the different value scales in different dimensions. We then group the inputs into a number of clusters (five in our experiments) by running a standard clustering algorithm (e.g., K-means) on the feature vectors. For each cluster, we compute its centroid. Note that the programmer is only required to provide a *input_feature* functions.

- **Step 3: Landmark Creation** We autotune the program using the PetaBricks evolutionary autotuner *multiple times*, once for each input cluster, using its centroid as the presumed inputs. While the default evolutionary search in autotuner generates random inputs at each step of search, we use a flag which indicates it should use the centroid instead. We call each configuration for each cluster's centroid as input data to the program, a *landmark*. The stochasticity of the autotuner means we may get different configurations albeit perhaps equal performing ones each time.

- **Step 4: Performance Measurement** We run each landmark configuration on *every* training input and record both the execution time and accuracy (if applicable) as its performance information.

We note that there is an alternative way to accomplish Steps 1 through 3 and identify landmarks. We could find the best configuration for each training input, group these configurations based on their similarity, and use the centroids of the groups as landmark configurations. This unfortunately is infeasible because it is too time consuming to find a suitable configuration for every input example. Modeling is not as effective as search, and search involves the composition and evaluation of hundreds of thousands configurations, taking hours or even days to finish. This also has the problem we mention previously: similar configurations do not have matching algorithm performance. For these reasons, we cluster in the space of inputs', determine an inputs centroid for each cluster and then autotune the centroid to get a landmark. This process will cost some extra time depending the number of landmarks we want to obtain, but it is a one time only cost to programmer.

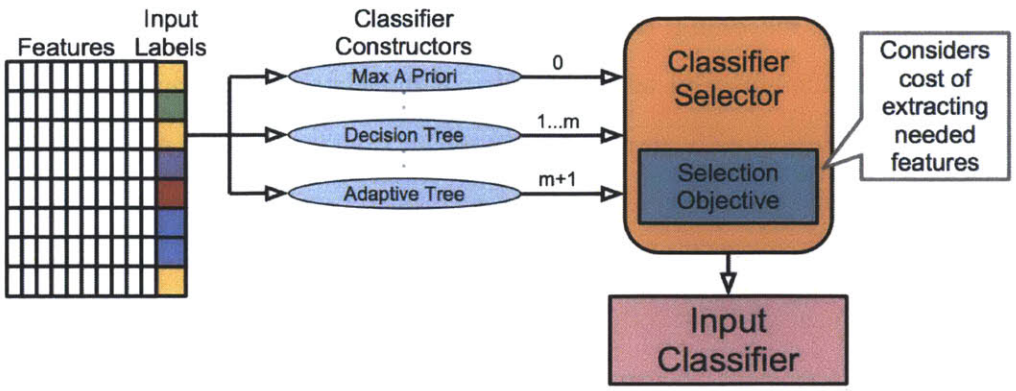


Figure 7.3: Constructing and selecting the input classifier.

7.2.4 Level 2

The main objective of Level 2 is to identify a production classifier. The challenge is to determine which input features are good predictors of a high performing classifier. If this was directly known, these features could be used to learn one classifier which would directly be used in production. Because it is not, the first sub-goal is to generate a candidate set of classifiers *each with a unique set of features* and all using the data that provides evidence of the relationship between inputs, configurations and algorithm performance. Incorporating this step and its wealth of evidence drastically improves results. The second sub-goal follows: choose among the candidates to identify the best one for production.

Data Conditioning Before Classifier Learning

We use machine learning to generate our classifiers. Per machine learning, we make each set of example inputs, their features, feature extraction costs, execution times and accuracy scores for each landmark configuration, a row of a dataset. We append to each row a label which is the best configuration for the input.

More formally, we create a datatable of 4-tuples where each 4-tuple is $< \mathbf{F}, \mathbf{T}, \mathbf{A}, \mathbf{E} >$, where $\mathbf{F}$ is a m-dimensional feature vector for this input, $\mathbf{T}$ and $\mathbf{A}$ are vectors of length $1 \times K_1$ where i^{th} entry represents the execution time and accuracy achieved for this input when i^{th} configuration is applied. $\mathbf{E}$ is a M-dimensional vector giving us the values for time taken for extraction of the

features. We first generate labels $L \in \{1 \ldots K_1\}$ for each input. Label l_i represents the best configuration for the i^{th} input. For problems where only minimizing the execution time is an objective (for example *sorting*) the label for i^{th} input is simply $\arg\max_j T_i^j$. For problems where both accuracy and execution time are objectives, we first choose a threshold for the accuracy and then select the subset of configurations that meet the accuracy threshold and among them pick the one that has the minimum execution time. For the case, in which none of the configs achieve desired accuracy threshold, we pick the configuration that gives the maximum accuracy.

We then divide our inputs into two sets, one set is used for training the classifier, the other for testing. We next pass the training set portion of the dataset to different classification methods which either reference different features or compute a classifier in a different way. Formally a method derives classifier C referencing a feature set $f_c \subset \mathbf{F}$ to predict the label, i.e., $C(F_i) \rightarrow L_i$.

Classifier Learning

We now describe the classifiers we derive and the methods we use to derive them.

Max-apriori classifier This classifier evaluates the empirical priors for each configuration label by looking at the training data. It chooses the configuration with the maximum prior (maximum number of inputs in the training data had this label) as the configuration for the entire test data. There is no training involved in this other than counting the exemplars of each label in the training data. As long as the inputs follow the same distribution in the test data as in the training data there is minimal mismatch between its performance on training and testing. It should be noted that this classifier does not have to extract any features of the input data.

Advantages: No training complexity, no cost incurred for extraction of features.

Disadvantages: Potentially highly inaccurate, vulnerable to error in estimates of prior.

Exhaustive Feature Subsets Classifiers Even though we have M features in the machine learning dataset, we only have $\frac{M}{z}$ input properties. The bottom level feature takes minimal time to extract and only looks at the partial input, while the top level takes significant amount of time as it looks at the entire input. However, features extracted for the same input could be highly

correlated. Hence, as a logical progression, we select a subset of features size of which ranges between $1 \ldots \frac{M}{z}$ where each entry is for a property. For each property we allow only one of its level to be part of the subset, and also allow it to be absent altogether. So for 4 properties with $z = 3$ levels we can generate 4^4 unique subsets. For each of these 256 subsets we then build a decision tree classifier [119] yielding a total of 256 classifiers.

The feature extraction time associated with each classifier is dependent on the subset of features used by the classifier, ie. for an input i, it is the summation $\sum_j E_i^j$. The decision tree algorithm references the label and features and tries minimize its label prediction error. It does not reference the feature extraction times, execution times or accuracy information. These will be referenced in accomplishing the second sub-goal of classifier selection.

Because we wanted to avoid any "learning to the data", we performed 10 fold cross validation.

Advantages: Feature selection could simply lead to higher accuracy and allow us to save feature extraction time.

Disadvantages: More training time, not scalable should the number of properties increase.

All features Classifier This classifier is one of the 256 Exhaustive Feature Subsets classifiers which we call out because it uses all the m features.

Advantages: Can lead to higher accuracy classification.

Disadvantages: More training time, higher feature extraction time, no feature selection.

Incremental Feature Examination classifier

Finally, we designed a classifier which works on an input in a sequential fashion. First, for every feature $f_m \in \mathbb{R}$, we divide it into multiple decision regions $\{d_1^m \ldots d_j^m\}$ where $j \geq K_1$. We then model the probability distributions under each class for each feature under each decision region $P_{m,j,k}(f_m = d_j^m | L_i = k)$. Given a pre-specified order of features it classifies in the following manner when deployed:

Step 1: Calculate the feature: Evaluate the m^{th} feature for the input and apply the thresholds to identify the decision region it belongs to.

Step 2: Calculate posterior probabilities: The posterior for a configuration (class label) k, given all the features $\{1 \ldots m\}$ acquired so far and let $d_1^1 \ldots d_j^i$ be the decision regions they belong to, is given by:

$$P(L_i = k | f_{1 \ldots m}) = \frac{\Pi_m P_{m,j,k}(f_m = d_j^m | L_i = k) P(L = k)}{\sum_k \Pi_m P_{m,j,k}(f_m = d_j^m | L_i = k) P(L = k)} \tag{7.1}$$

Step 3: Compare and decide: We pre-specify a threshold on the posterior Δ and we declare the configuration (class label) as k if its posterior $P(L_i = k | f_{1 \ldots m}) > \Delta$. If none of the posteriors are greater than this threshold, we return to step 1 to acquire more features.

In this method, we incrementally acquire features for a input point i based on judgement as to whether there is enough evidence (assessed via posteriors) for them to indicate one configuration. This results in a variable feature extraction time for different inputs thus providing potential further reductions in feature extraction time at the time the production classifier is applied to new inputs. For all the classifiers preceding this one, we extracted the same number of features for all the inputs.

This method can be applied after the previous method has found the best subset to further save on feature extraction time.

To train this classifier, we need to find the optimal decision regions for each feature and the threshold on posterior Δ such that the performance measurements mentioned above are minimized. This could be done via a simple continuous parameter search algorithm. Usually, more decision regions per feature help increase the performance and in some cases search over orders could help. This is the only classifier in our system where we introduce the domain specific cost function into the inner loop of training of a classifier.

Advantages: Reduced feature extraction time, scalable as the number of attributes increase.

Disadvantages: One possible drawback for this classifier is that it requires a storage of $i \times j \times k$ number of probabilities in a look up table. Training time.

Candidate Selection of Production Classifier

After we generate a set of classifiers based on different inputs or methods, we next need to select one as the production classifier. We start by applying every classifier on the test set and measuring the performance (execution time and accuracy, if required) of the algorithm when executing with its predicted configuration. We can compare this performance to that of the rest configuration. There are three objectives for the production classifier: 1) minimize execution time; 2) meet accuracy requirements ; and, 3) minimize the feature extraction time.

Let β_i be the minimum execution time for the input i by all the representative polyalgorithms. Let $\Psi(i, L_i)$ be the execution time of i when its class label is L_i, given by classifier C and $g_i = \sum_j T_j, j \in f_c$ be the feature extraction time associated with this classification.

Given a classifier we measure its efficacy for our problem as follows:

For time only:The cost incurred (represented by r_i) for classifying a data point to configuration c_i will be $r_i = \Psi(i, c_i) + g_i$. The cost function (represented by R) for all the data will be the average of all their costs, that is, $R = \sum_i(r_i)/N$, where N is the total number of data lists. We refer to R as performance cost in the following description.

For time and accuracy: Let H be the accuracy threshold, that is, only when the accuracy of the computation (e.g., binpacking) result at a data list exceeds H, the result is useful. The value of H can be prefixed by programmer.

Suppose the fraction of data lists whose computation results are inaccurate (ie. accuracy is less than H) is s when classifier C is applied to our data set. We set a target on the s. If a classifier does not meet this target, it is considered invalid (or incurring a huge cost). If a classifier meets this target then the cost of this classifier is calculated as defined above.

7.2.5 Discussion of the Two Level Learning

This two level learning has several important properties.

First, it uses a two level design to systematically address *mapping disparity*. Its first phase takes advantage of the properties of inputs to identify landmark configurations. It then furnishes

evidence of how example inputs and different landmarks affect program performance (execution time and accuracy). Its second phase uses this evidence to (indirectly) learn a production classifier. By classifying based upon best landmark configuration it avoids misusing similarity of inputs. The means by which it evaluates each candidate classifier (trained to identify the best landmark) to determine the production classifier takes into account the performance of the configurations both in terms of execution time and accuracy.

Second, this two level learning reconciles the stress between accuracy and performance by introducing a programmer-centric scheme and a coherent treatment to the dual objectives at both levels of learning. The scheme allows programmers to provide two thresholds. One is an *accuracy threshold*, which determines whether the computation result is considered as accurate; the other is a *satisfaction threshold*, which determines whether the statistical accuracy guarantee (e.g., the calculation is accurate in 95% time) offered by a configuration meets the programmer's needs. The scheme is consistently followed by both levels of learning.

Third, it seamlessly integrates consideration of the feature extraction overhead into the construction of input classifiers. Expensive feature extractions may provide more accurate feature values but cost more time than cheap ones do. The key question is to select the feature extractions that can strike a good tradeoff between the usefulness of the features and the overhead. Our two level learning framework contains two approaches to finding the sweet point. One uses exhaustive feature selection, the other uses adaptive feature selection. Both strive to maximize the performance while maintaining the accuracy target.

Fourth, our learning framework maintains an open design, allowing easy integration of other types of classifiers. Any other classification algorithm could be integrated into our system without loss of generality. Plus it takes advantage of the PetaBricks autotuner to intelligently search through the vast configuration space.

7.3 Evaluation

To measure the efficacy of our system we tested it on 6 of the benchmarks described in Chapters 4 and 5. Of these benchmarks 1 requires fixed accuracy and 5 require variable accuracy. Each of

these benchmarks was modified to add feature extractors for their inputs and a richer set of input generators to exercise these features. Each feature extractor was set to 3 different sampling levels providing more accurate measurements at increasing costs. Tests were conducted on a 32-core (8×4-sockets) Xeon X7550 system running GNU/Linux (Debian 6.0.6).

We use two primary baselines to provide both a lower bound of performance without input adaptation and an upper bound of the limits of input adaption. Neither baseline includes (or requires) any feature extraction costs.

- *Static oracle* uses a single configuration for all inputs. This configuration is selected by trying each input optimized program configuration and picking the one with the best performance. The static oracle is the performance that would be obtained by not using our system and instead using an autotuner without input adaptation. In practice the static oracle may be better than some offline autotuners, because such autotuners may train on non-representative sets of inputs.

- *Dynamic oracle* uses *the best* configuration for each input. It is the lower bound of the best possible performance that can be obtained by our input classifier. It is equivalent to a classifier that always picks the best optimized program and requires no features to do so. We allow the dynamic oracle to miss the accuracy target on up to 10% of the inputs, to match the selection criteria of the input classifier.

7.3.1 Input Features and Inputs

We use 6 of the benchmarks described in Chapters 4 and 5 to evaluate our results. We modified each of these benchmarks to contain the input features described below.

Sort Sort, described in Section 4.1.2, is the only non-variable accuracy benchmark shown. Input variability comes from different algorithms having fast and slow inputs, for example QuickSort has pathological input cases and InsertionSort is good on mostly-sorted lists. For input features we use standard deviation, duplication, sortedness, and the performance of a test sort on a subsequence of the list.

Sort1 results are sorting real-world inputs taken from the Central Contractor Registration (CCR) FOIA Extract, which lists all government contractors available under FOIA from data.gov. *Sort2* results are sorting synthetic inputs generated from a collection of input generators meant to span the space of features.

Clustering Clustering is described in Section 4.4.2 and uses input the features: radius, centers, density, and range.

Clustering1 results are clustering real-world inputs taken from the Poker Hand Data Set from UCI machine learning repository. *Clustering2* results are clustering synthetic inputs generated from a collection of input generators meant to span the space of features.

Bin Packing Bin packing is described in Section 4.4.1 and contains 4 input feature extractors: average, standard deviation, value range, and sortedness.

Singular Value Decomposition The SVD benchmark is described in Section 4.4.3 and for input features we used range, the standard deviation of the input, and a count of zeros in the input.

Poisson 2D Poisson equation is a multigrid benchmark described in Chapter 5. For input features we used the residual measure of the input, the standard deviation of the input, and a count of zeros in the input.

Helmholtz 3D Holmholtz equation is a multigrid benchmark described in Chapter 5. For input features we used the residual measure of the input, the standard deviation of the input, and a count of zeros in the input.

7.3.2 Experimental Results

Figure 7.4 shows the overall performance of our system on an isolated testing data set. Overall results range 1.046x speedup for helmholtz3d, to 3.054x speedup for sorting. Both of these results are close to the dynamic oracle performance of 1.111x and 6.622x for these same two benchmarks.

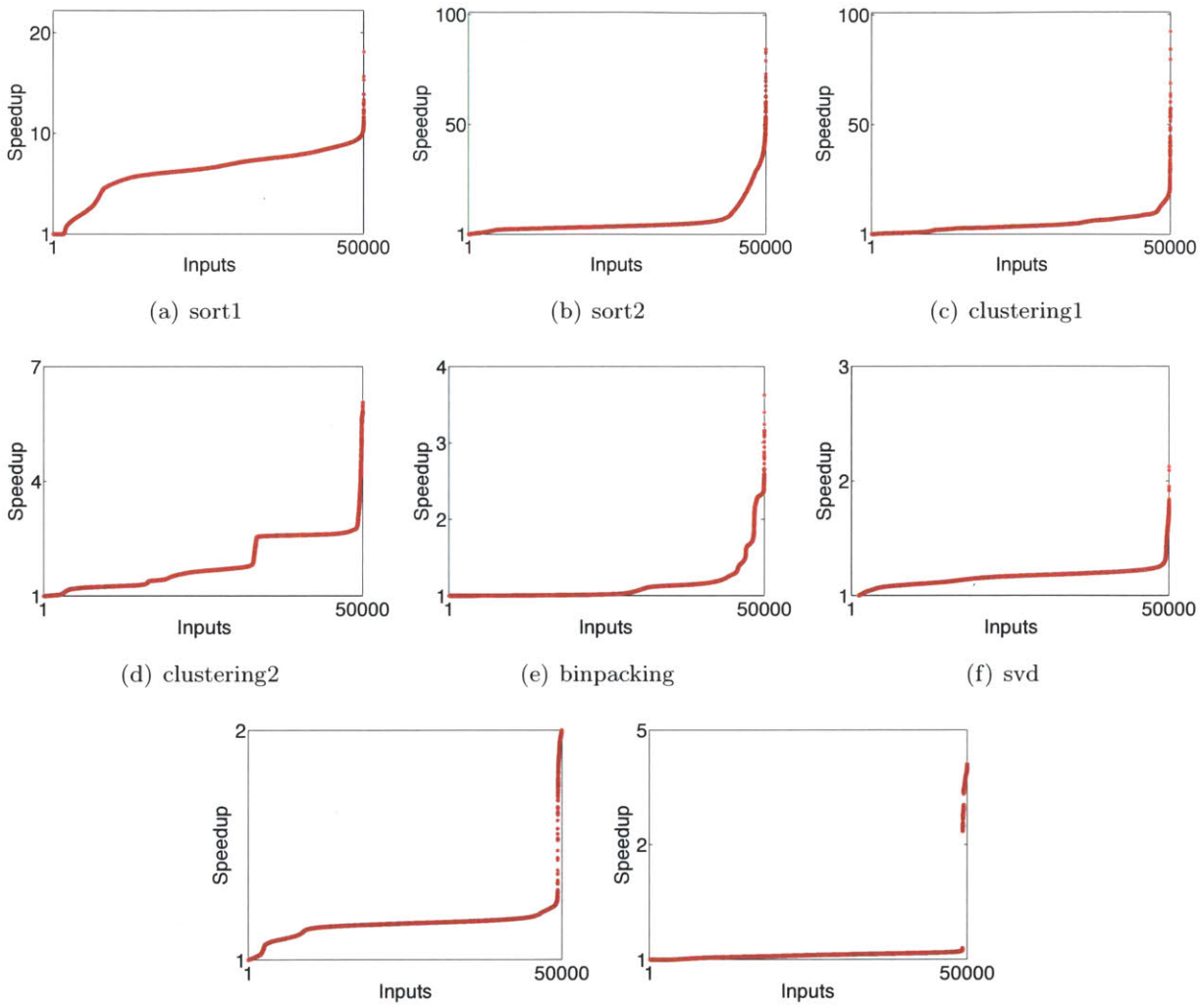
Benchmark Name	Dynamic Oracle	Classifier (w/o feature extraction)	Classifier (w/ feature extraction)
sort1	5.104×	2.946×	2.905×
sort2	6.622×	3.054×	3.016×
clustering1	3.696×	2.378×	2.370×
clustering2	1.674×	1.446×	1.180×
binpacking	1.094×	1.093×	1.081×
svd	1.164×	1.108×	1.105×
poisson2d	1.121×	1.086×	1.086×
helmholtz3d	1.111×	1.046×	1.044×

Figure 7.4: Mean speedup over the static oracle of the generated input classifier (with and without feature extraction costs included) and the dynamic oracle. Static oracle uses the best *single* configuration for all inputs, and is the best performance achievable without using input adaption. Dynamic oracle uses the best configuration for each input, and is the upper bound speedup one would get with a "perfect" input classifier.

Generally, the less expensive features (in terms of feature extraction time) were sufficient to meet the best performance. Additionally, we note that most of the features we extracted had orders of magnitude smaller extraction time when compared to the execution time. This obviated the need for the adaptive classifier in the current scenario.

In the sort benchmark, we also tried both real world inputs (sort1) and inputs from our own generator (sort2). For real world input, the best classifier used the sorted list and sortedness features at its intermediate sampling level and the duplication and deviation at the cheapest level. 2.946x speedup was achieved compared to a dynamic oracle speedup of 5.104x. For inputs from our own generator, the best classifier used the sorted list and sortedness features at its intermediate sampling level, achieving our largest speedup of 3.054x compared to a dynamic oracle of 6.622x.

In the clustering benchmark, we tried real world inputs and those from our own generator. For real world input, the best classifier used the density feature at its cheapest level, and algorithms selected by the classifier causes a 2.378x shorter execution time (compared to the dynamic oracle speedup of 3.696x), For our own generator, the best classifier used the centers feature at its cheapest sampling level, achieving a 1.446x speedup compared to a dynamic oracle of 1.674x. However,

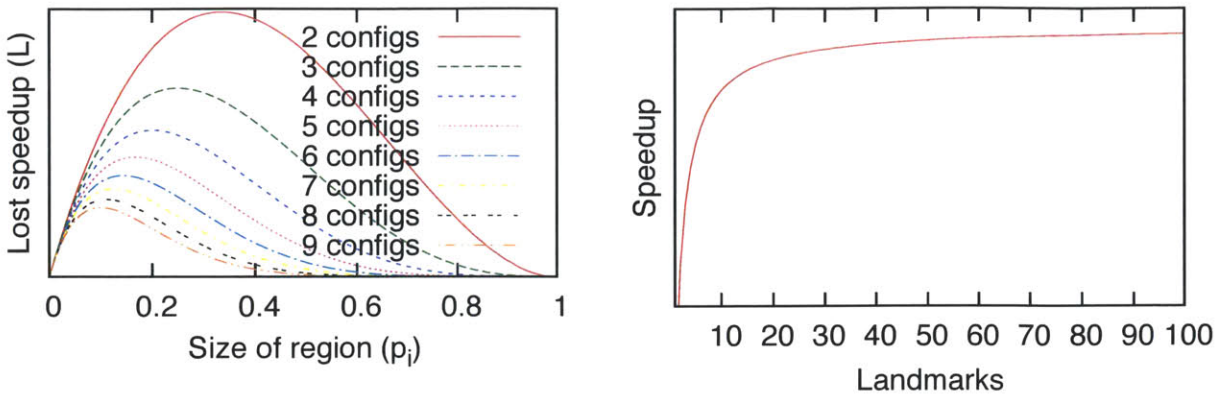


(a) sort1 (b) sort2 (c) clustering1

(d) clustering2 (e) binpacking (f) svd

(g) poisson2d (h) helmholtz3d

Figure 7.5: Distribution of speedups over static oracle for each individual input. For each problem, some individual inputs get much larger speedups than the mean.



(a) Predicted loss in speedup contributed by input space regions of different sizes.

(b) Predicted speedup with a worst-case region size with different numbers of sampled landmark configurations.

Figure 7.6: Model predicted speedup compared to sampling all input points as the number of landmarks are increated. Y-axis units are omitted because the problem-specific scaling term in the model is unknown.

centers feature is the most expensive feature relative to execution time, which lowers the effective speedup to just 1.180x.

In the binpacking benchmark, the best classifier used the deviation and sortedness features at the intermediate level. The classifier is selecting algorithms that cause a 1.093x faster execution time (close to the dynamic oracle speedup of 1.094x).

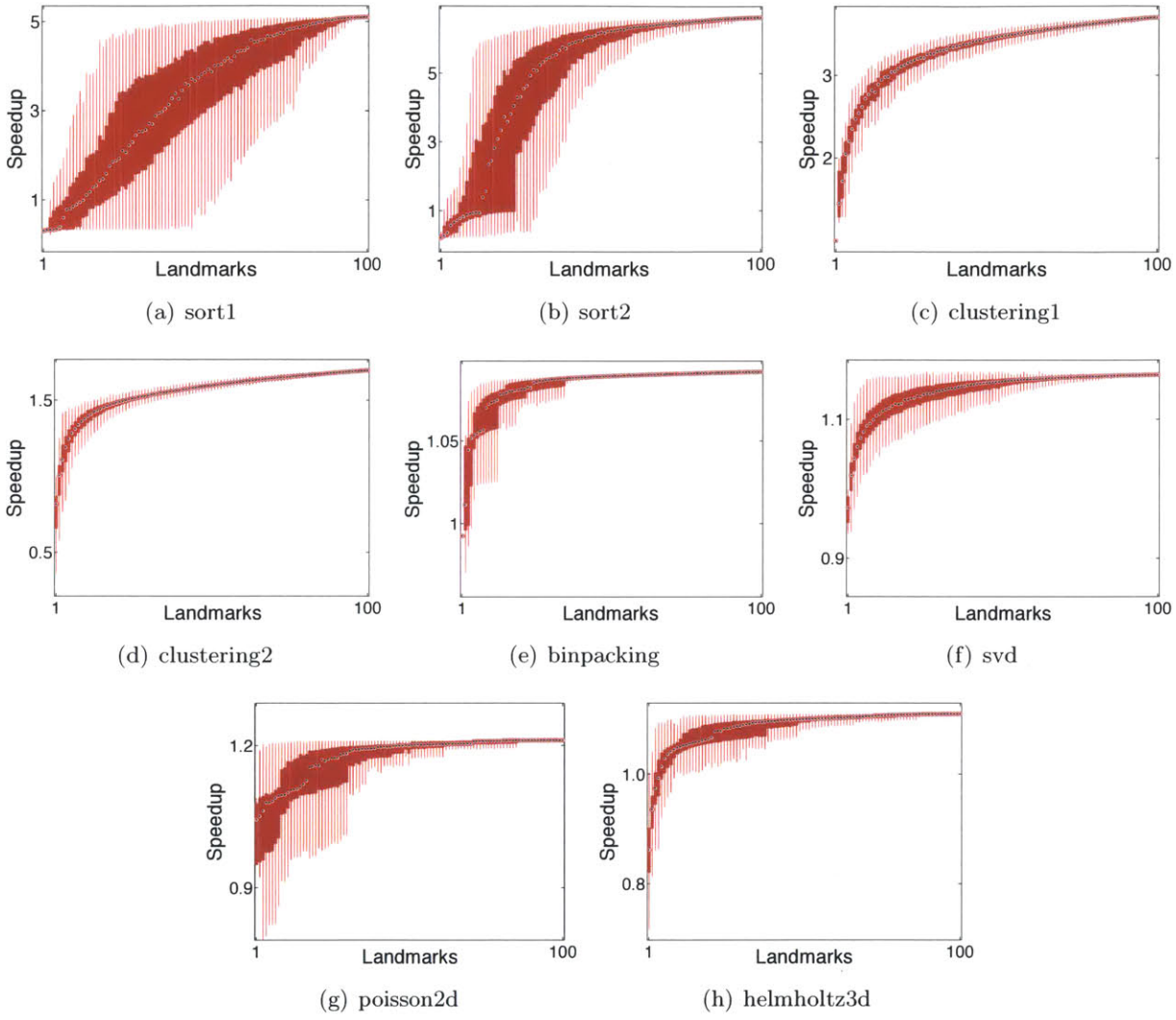
In the svd benchmark, the best classifier used only zeros input feature at the intermediate level and achieved 1.108x speedup compared to a dynamic oracle of 1.164x.

In the poisson2d benchmark, the best classifier employed the input features zeros at the intermediate level, achieving a 1.086x speedup compared to a dynamic oracle of 1.121x.

In the helmholtz3d benchmark, the best classifier used the residue, zeros and deviation input features at the intermediate level and the range feature at the cheapest level. This benchmark showed a 1.046x speedup, compared to a dynamic oracle speedup of 1.111x.

7.3.3 Input Generation

For *sort1* and *clustering1* we used real world input data taken from production systems. The performance for this real world data can be compared to *sort2*, *clustering2*, and other benchmarks



(a) sort1 (b) sort2 (c) clustering1

(d) clustering2 (e) binpacking (f) svd

(g) poisson2d (h) helmholtz3d

Figure 7.7: Measured speedup over static oracle as the number of landmark configurations changes, using 1000 random subsets of the 100 landmarks used in other results. Error bars show median, first quartiles, third quartiles, min, and max.

where we used synthetic input generators designed to span the feature space. For sort, real world inputs provides similar mean speedup to synthetic inputs. For clustering, real world inputs saw much larger speedups than synthetic inputs. Interestingly, the classifier for synthetic inputs in clustering needs to use a much more expensive set of input features because the classes of inputs are harder to distinguish.

Figure 7.5 shows the distribution of speedups for individual inputs to each program, sorted such that the largest speedup is on the right. What is interesting here is the speedups are not uniform. For each benchmark there exist small sets of inputs with very large speedups, in some cases up to $90x$. This shows that way inputs are chosen can have a large effect on mean speedup observed. If one had a real world input distribution that favored these types of inputs the overall speedup of this technique would be much larger. In other words, the relative benefits of using of input adaptation techniques can vary drastically depending on your input data distribution.

7.3.4 Model of Diminishing Returns with More Landmark Configurations

In addition to the evaluation of our classifier performance it is important to evaluate if our methodology of clustering and using 100 landmark configurations is sufficient. To help gain insight into this question we created a theoretical model where we consider the input search space of a program where some finite number of optimal program configurations dominate different subsets of the input space. For each of these dominate configurations, we define the values p_i and s_i, where p_i is fraction of the inputs in the search space where this configuration dominates and s_i is the speedup on these configurations obtained by training a configuration for any of the inputs where this configuration dominates. The model assumes that no speedup is obtained if one of these points is not sampled. We also assume that all inputs have equal cost before the speedups are applied, to avoid the need for weighting terms.

If we assume the k landmark configurations are sampled uniform randomly (which is likely a worse technique than our actual clustering) the total expected loss in speedup, L, compared to a

perfect method that sampled all points would be:

$$L = \sum_i (1 - p_i)^k p_i s_i$$

Where $(1-p_i)^k$ represents the chance of "missing" the region of the search space where configuration i is optimal and $p_i s_i$ represents the cost of missing that region of the search space in terms of speedup.

Figure 7.6(a) shows the value of this function for a single region as p_i changes. One can see that on the extremes $p_i = 0$ and $p_i = 1$ there is no loss in speedup, because either the region is so small a speedup in that region does not matter or the region is so large that random sampling is likely to find it. For each number of configs, there exists a worst-case region size where the expected loss in speedup is maximized. We can find this worst-case region size by solving for p_i in $\frac{dL}{dp_i} = 0$ which results in a worst-case $p_i = \frac{1}{k+1}$. Using this worst-case region size, Figure 7.6(b) shows the diminishing returns predicted by our model as more landmark configurations are sampled. Figure 7.7 validates this theoretical model by running each benchmark with varying numbers of landmark configurations. This experiment takes random subsets of the 100 landmarks used in other results and measures that speedup over the static oracle. Real benchmarks show a similar trend of diminishing returns with more landmarks that is predicted by our model. We believe that this is strong evidence that using a fixed number of landmark configurations suffices in practice, however correct number of landmarks needed may vary between benchmarks.

Chapter 8

Online Autotuning

We have shown so far that 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.

In this chapter, 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.

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.

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.

8.1 Competition Execution Model

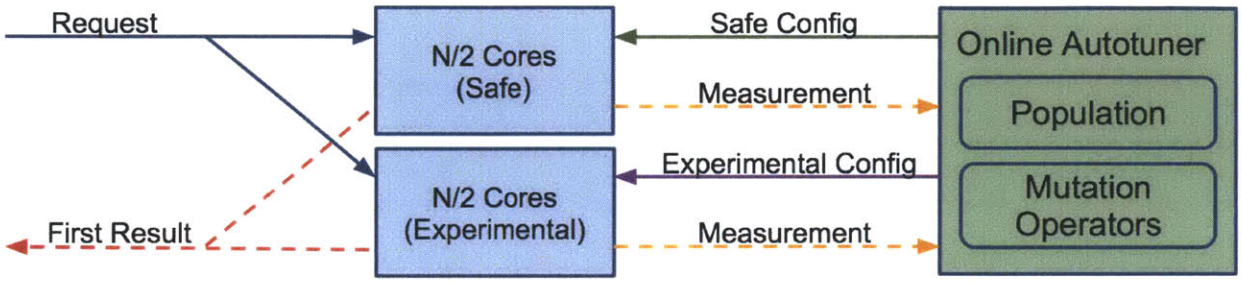


Figure 8.1: High level flow of the runtime system. The data on dotted lines may not be transmitted for the slower configuration, which can be terminated before completion.

Figure 8.1 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.

8.1.1 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.

8.1.2 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.

8.2 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 8.1. 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 8.2.4.

8.2.1 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 8.2.4 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 8.2.3) 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.

8.2.2 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.

8.2.3 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".

8.2.4 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) [46, 49, 139] 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 [130].

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 [32] 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 [20]:

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

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 [130] for more details.

8.2.5 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 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.

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

8.3 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 8.1. 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.

8.3.1 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.

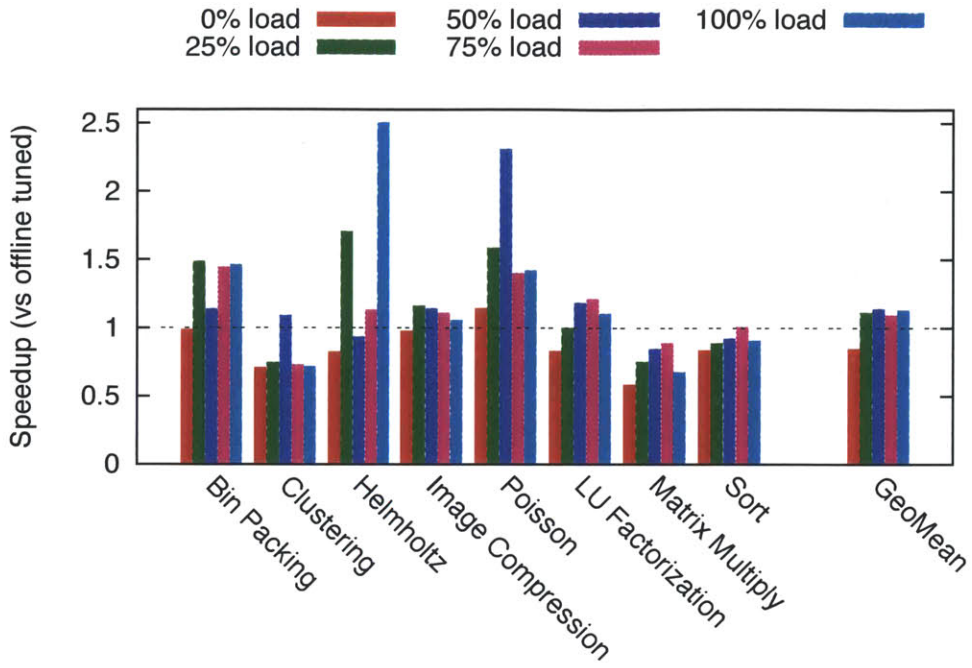
- For the variable accuracy benchmarks, additional speedup can be obtained since the online tuner receives runtime feedback on how well it is meeting its accuracy targets. If it observes that it is over delivering on its quality of service target it can opportunistically change algorithms, enabling it 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.

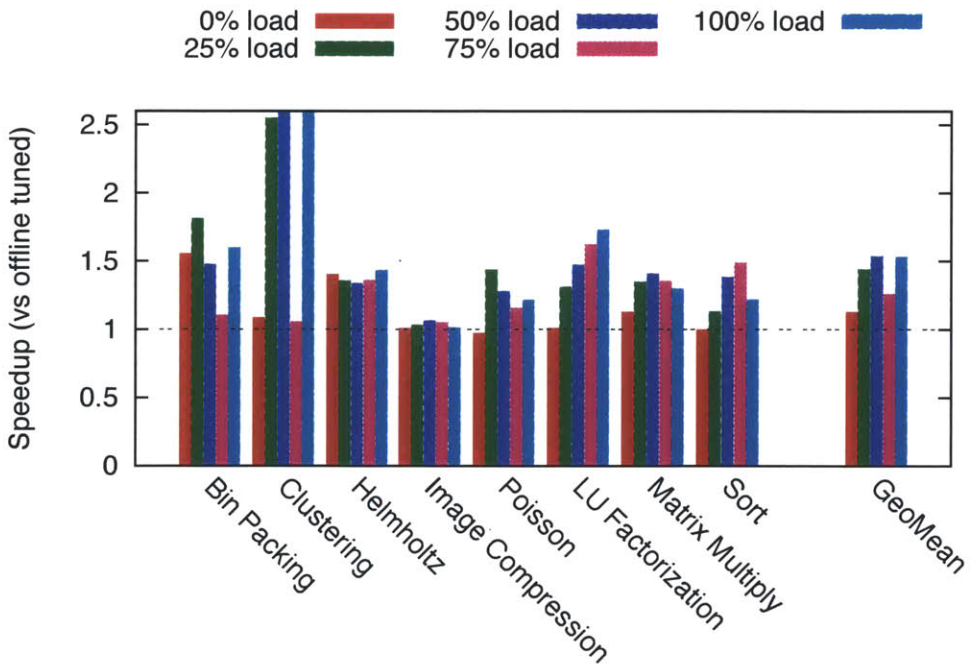
- As the number of processing cores continues to grow exponentially, the amount of per core memory bandwidth is decreasing dramatically since per-chip memory bandwidth is growing only at a linear rate [26]. This fact, coupled with Amdahl's law, makes it particularly difficult to write applications with 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.

8.3.2 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



(a) Xeon8



(b) AMD48

Figure 8.2: 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.

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 8.2(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 [12]). 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 8.2(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. 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.

8.3.3 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

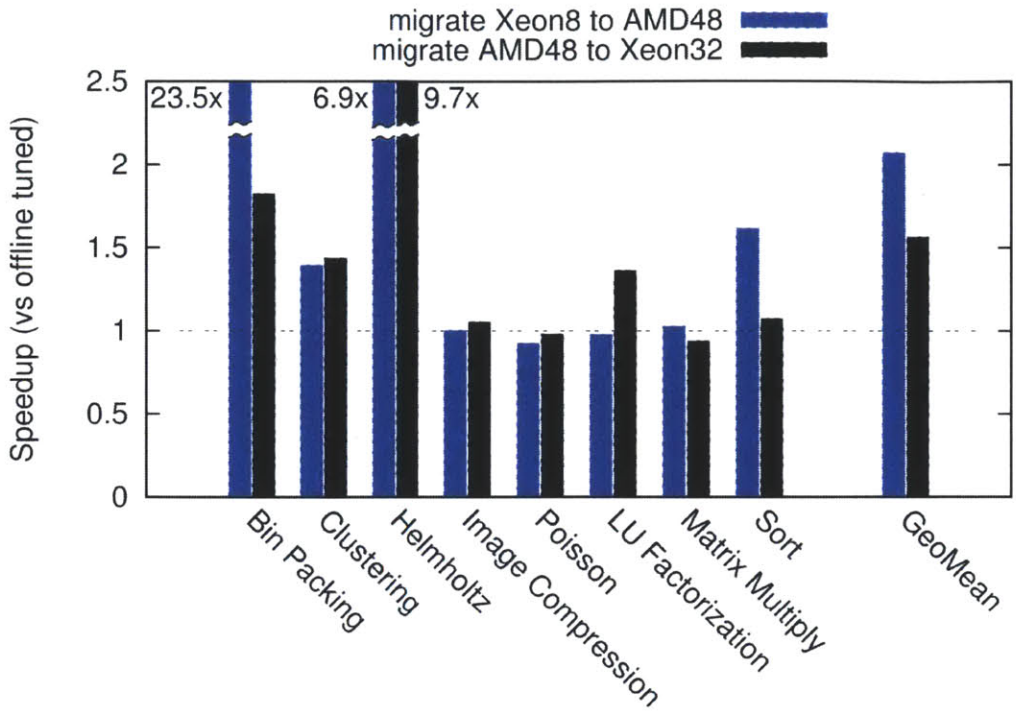


Figure 8.3: Speedups (or slowdowns) of each benchmark after a migration between microarchitectures. "Normalized throughput" is the throughput over the first 10 minutes of execution of SiblingRivalry (including time to learn), divided by the throughput of the first 10 minutes of an offline tuned configuration using the entire system.

core on the system. Figure 8.3 shows the speedups for each benchmark after such a migration. SiblingRivalry shows a geometric mean speedup of 1.8x in this migration experiment.

Starting Configuration Figure 8.4 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 8.5 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

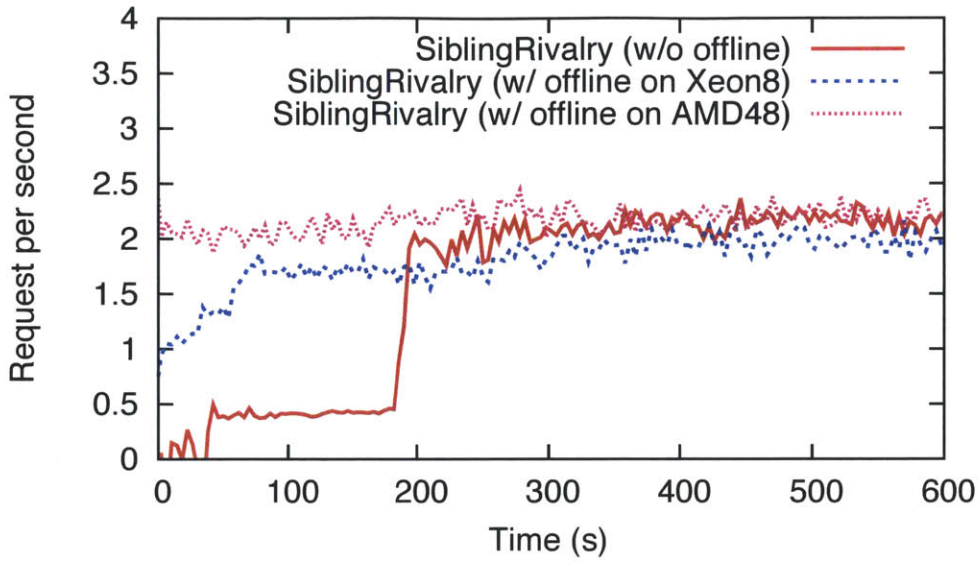


Figure 8.4: The effect of using an offline tuned configuration as a starting point for SiblingRivaly on the Sort benchmark. We compare starting from a random configuration ("w/o offline") to configurations found through offline training on the same and a different architecture.

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.

8.4 Hyperparameter Tuning

SiblingRivalry uses as an underlying algorithm bandit-based adaptive operator selection and the Upper Confidence Bound (UCB) algorithm. This technique introduces two hyperparameters: W - the length of the history window, and C - the balance point between exploration and exploitation. UCB is only optimal if these hyperparamters are set by an oracle or through some other search technique. In practice, a user of this technique must either use a fixed, non-optimal assignment of these hyperparameters, or perform a search over hyperparameters whenever the search space changes. Unfortunately, in practice, finding good values of these hyperparameters may be more expensive that the actual search itself. While [61] addresses the robustness of hyperparameters in empirical academic study we present a practically motivated, real world study

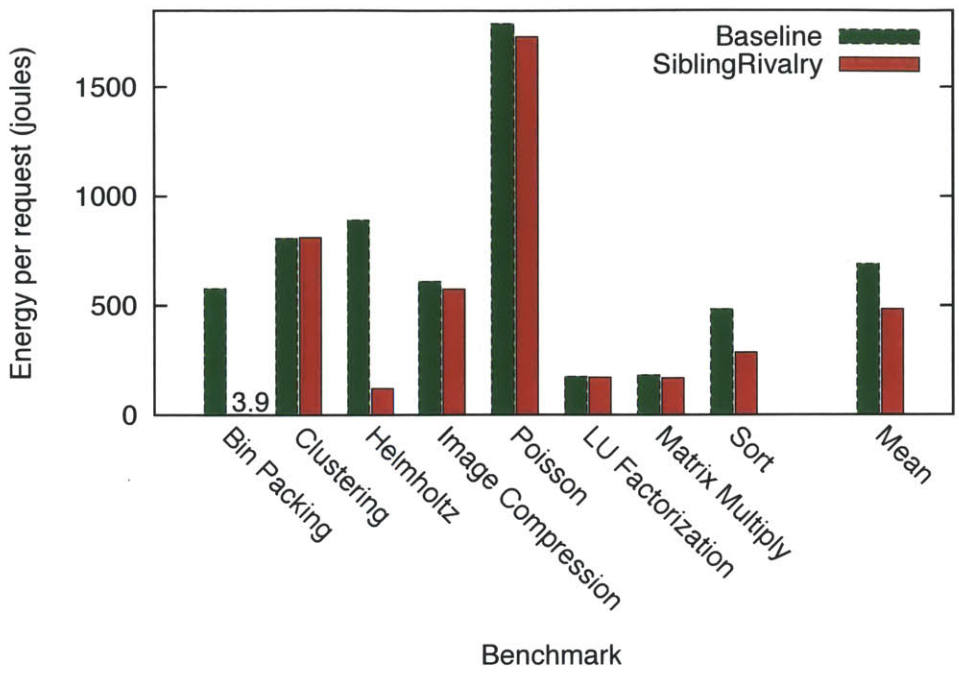


Figure 8.5: Average energy use per request for each benchmark after migrate Xeon8 to AMD48.

on setting hyperparameters. We define evaluation metrics that can be used in score functions that appropriately gauge the autotuner's performance in either a static or dynamic environment and use them to ask:

- How much does the optimal assignment of hyperparameters vary when tuning different programs in two classes of environments - static or dynamic?
- Does there exist a single "robust" assignment of hyperparameters for a context that performs close to optimal across all benchmarks?

8.4.1 Tuning the Tuner

The hyperparameters C (exploration/exploitation trade-off) and W (window size) can have a significant impact on the efficacy of SiblingRivalry. For example, if C is set too high, it might dominate the exploitation term and all operators will be applied approximately uniformly, regardless of their past performance. If, on the other hand, C is set too low, it will be dominated by the

exploitation term $\hat{q}_{i,t}$ and new, possibly better operators will rarely be applied in favor of operators which made only marginal improvements in the past.

The problem is further complicated by the fact that the optimal balance between exploration and exploitation is highly problem-dependent [61]. For example, programs with a lot of algorithmic choices are likely to benefit from a high exploration rate. This is because algorithmic changes create discontinuities in the program's fitness, and operator weights calculated for a given set of algorithms will not be accurate when those algorithms suddenly change. When such changes occur, exploration should become the dominant behavior. For other programs, e.g. those where only a few mutators improve performance, sacrificing exploration in favor of exploitation might be optimal. This is especially true for programs with few algorithmic choices - once the optimal algorithmic choices have been made, the autotuner should focus on adjusting cutoffs and tunables using an exploitative strategy with a comparatively low C.

The optimal value of C is also closely tied to the optimal value of W, which controls the size of the history window. The autotuner looks at operator applications in the past W races, and uses the outcome of those applications to assign a quality score to each operator. This is based on the assumption that an operator's past performance is a predictor of its future performance, which may not always be true. For example, changes in algorithms can create discontinuities in the fitness landscape, making past operator performance largely irrelevant. However, if W is large, this past performance will still be taken into account for quite some time. In such situations, a small W might be preferred.

Furthermore, optimal values of C and W are not independent. Due to the way $\hat{q}_{i,t}$ is computed, the value of the exploitation term grows with W. Thus by changing W, which superficially controls only the size of the history window, one might accidentally alter the exploration/exploitation balance. For this reason, C and W should be tuned together.

Finally, the task of selecting hyperparameters is complicated by the fact that different hyperparameter values might be optimal at different stages of the autotuning process. As described earlier, a larger C might be favorable following algorithm changes, with a smaller C when optimal algorithmic choices have already been made. Currently, however, SiblingRivalry does not allow

dynamically adjusting hyperparameters throughout the run, which have to be statically set before the autotuning begins.

8.4.2 Evaluation metrics

Because there is no single metric that will suffice to evaluate performance under different hyperparameter values, we use three separate metrics to evaluate SiblingRivalry on a given benchmark program with different hyperparameters:

1. **Mean throughput**: the number of requests processed per second, averaged over the entire duration of the run. Equal to the average number of races per second.

2. **Best candidate throughput**: inverse of the runtime of the fastest candidate found during the duration of the run. For variable accuracy benchmarks, only candidates that met the accuracy target are considered.

3. **Time to convergence**: number of races until a candidate has been found that has a throughput within 5% of the best candidate for the given run. For variable accuracy benchmarks, only candidates that met the accuracy target are considered.

To enable a fair comparison between SiblingRivalry's performance under different hyperparameter values, we define a single objective metric for each scenario that combines one or more of the metrics outlined above. We call this metric the *score function* f_b for each benchmark b, and its output the *score*.

We consider two classes of execution contexts: static and dynamic. In the static context, the program's execution environment is mostly unchanging. In this setting, the user cares mostly about the quality of the best candidate. Convergence time is of little concern, as the autotuner only has to learn once and then adapt very infrequently. For the sake of comparison, we assume in this scenario the user assigns a weight of 80% to the best candidate's throughput, and only 20% to the convergence time. Hence the score function for the static context:

$$f_b(C, W) = 0.8 \times \text{best\_throughput}_b(C, W) + 0.2 \times \text{convergence\_time}_b^{-1}(C, W)$$

In the dynamic context, the user cares both about average throughput and the convergence time. The convergence time is a major consideration since execution conditions change often in a dynamic system and necessitate frequent adaptation. Ideally, the autotuner would converge very quickly to a very fast configuration. However, the user is willing sacrifice some of the speed for improved convergence time. We can capture this notion using the following score function:

$$f_b(C, W) = 0.5 \times \text{mean\_throughput}_b(C, W) + 0.5 \times \text{convergence\_time}_b^{-1}(C, W)$$

We normalize throughput and convergence time with respect to their best measured values for the benchmark, so that the computed scores assume values in the range $[0, 1]$, from worst to best. Note that those are theoretical bounds: in practice it is often impossible to simultaneously maximize both throughput and convergence time.

	static context				dynamic context			
	Xeon8		AMD48		Xeon8		AMD48	
	C	W	C	W	C	W	C	W
Sort	50.00	5	5.00	5	5.00	5	5.00	5
Bin Packing	0.01	5	0.10	5	5.00	500	5.00	500
Poisson	50.00	500	50.00	500	0.01	500	5.00	5
Image Compression	0.10	100	50.00	50	0.01	100	50.00	50

(a) Best performing values of the hyperparameters C and W over an empirical sample.

	static context		dynamic context	
	Xeon8	AMD48	Xeon8	AMD48
Sort	0.8921	0.8453	0.9039	0.9173
Bin Packing	0.8368	0.8470	0.9002	0.9137
Poisson	0.8002	0.8039	0.8792	0.6285
Image Compression	0.9538	0.9897	0.9403	0.9778

(b) Scores of the best performing hyperparameters.

Figure 8.6: Best performing hyperparameters and associated score function values under static and dynamic autotuning scenarios.

8.4.3 Results

We evaluated the hyperparameter sensitivity of SiblingRivalry by running the autotuner on a set of four benchmarks: sort, Bin Packing, Image Compression and Poisson. We used twenty different combinations of C and W for each benchmark: $(C, W) = [0.01, 0.1, 0.5, 5, 50] \times [5, 50, 100, 500]$.

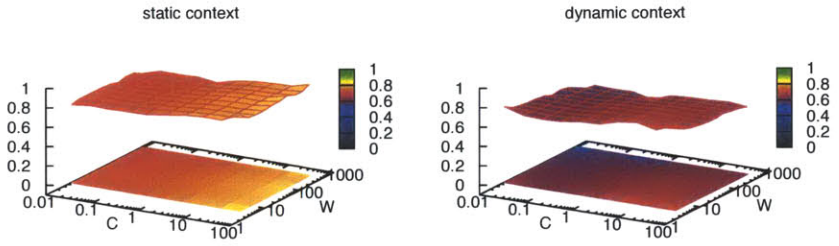
For each run, we measured the metrics described in Section 8.4.2 and used them to compute score function values. We performed all tests on the `Xeon8` and `AMD48` systems (see Table 8.2). The reported numbers for `Xeon8` have been averaged over 30 runs, and the numbers for `AMD48` over 20 runs. The benchmarks are described in more detail in Chapter 4.

Acronym	Processor Type	Operating System	Processors
Xeon8	Intel Xeon X5460 3.16GHz	Debian 5.0	2 ($\times$4 cores)
AMD48	AMD Opteron 6168 1.9GHz	Debian 5.0	4 ($\times$12 cores)

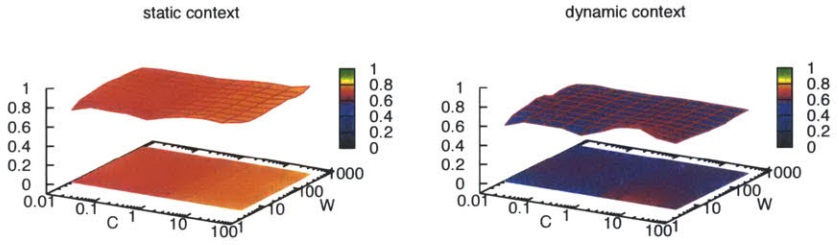
Table 8.2: Specifications of the test systems used.

Figures 8.7 and 8.8 show select scores as a function of C and W on the `Xeon8` amd `AMD48` systems for benchmarks in both static and dynamic scenarios. All benchmarks except Image Compression show moderate to high sensitivity to hyperparameter values, with Bin Packing performance ranging from as low as 0.1028 at $(C, W) = (0.01, 5)$ to as high as 0.9002 at $(C, W) = (5, 500)$ in the dynamic scenario on the `Xeon8`. On average, the dynamic context was harder to autotune with a mean score of 0.6181 as opposed to static system's 0.6919 (Figure 8.9). This result confirms the intuition that maintaining a high average throughput while minimizing convergence time is generally more difficult than finding a very high-throughput candidate after a longer autotuning process.

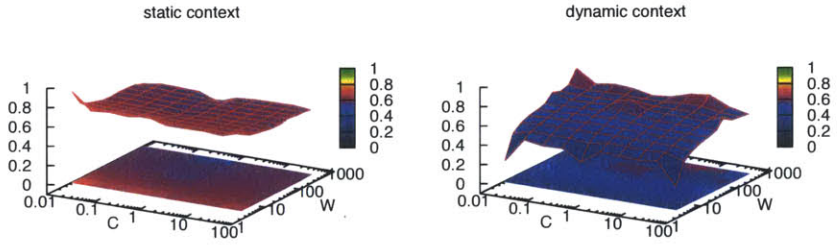
The optimal hyperparameter values for each benchmark ranged considerably and depended on both the scenario and the architecture (Table 8.6). sort tended to perform best with a moderate C and a low W, underlining the importance of exploration in the autotuning process of this benchmark. Bin Packing in the static context favored a balance between exploration and exploitation of a small number of recently tried operators. In the dynamic context Bin Packing performed best with much longer history windows (optimal $W = 500$) and with only a moderate exploration term $C = 5$. This is expected as Bin Packing in the dynamic context is comparatively difficult to autotune and hence benefits from a long history of operator performance. Poisson was



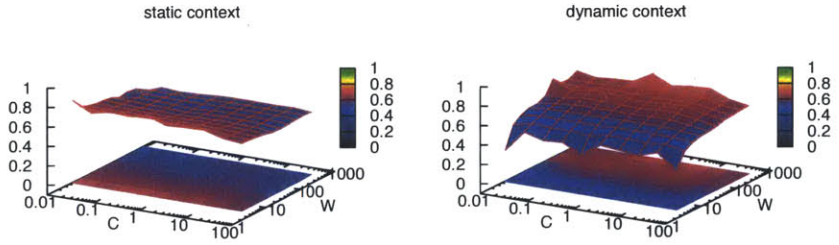
(a) sort on `Xeon8`



(b) sort on `AMD48`



(c) Bin Packing on `Xeon8`



(d) Bin Packing on `AMD48`

Figure 8.7: Scores for Sort and Bin Packing as a function of C and W. The colored rectangle is a plane projection of the 3D surface and is shown for clarity.

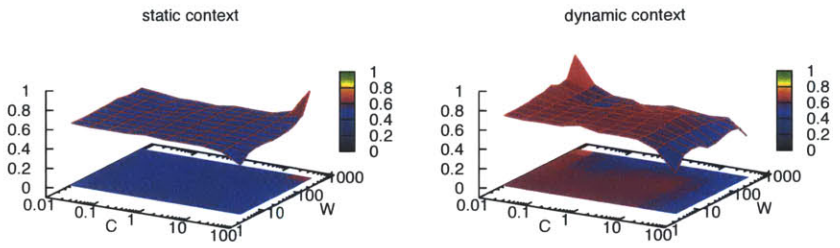
another "difficult" benchmark, and as a result performed better with long histories ($W = 500$ for almost all architecures and contexts). In the static scenario it performed best with a high $C = 50$, confirming the authors' intuition that exploration is favorable if we are given more time to converge. In the dynamic context exploration was favored less (optimal $C = 0.01$ for the `Xeon8` and $C = 5$ for the `AMD48`). In the case of Image Compression, many hyperparameters performed close to optimum suggesting that it is an easy benchmark to tune. Medium W were preferred across architectures and scenarios, with $W = 100$ and $W = 50$ for the static and dynamic contexts, respectively. Image Compression on `AMD48` favored a higher $C = 50$ for both scenarios, as opposed to the low $C = 0.1$ and $C = 0.01$ for the static and dynamic contexts on the `Xeon8`. This result suggests exploitation of a limited number of well-performing operators on the `Xeon8`, as opposed to a more explorative behavior on the `AMD48`. We suspect this is due to a much higher parallelism of the `AMD48` architecture, where as parallelism increases different operators become effective.

8.4.4 Hyperparameter Robustness

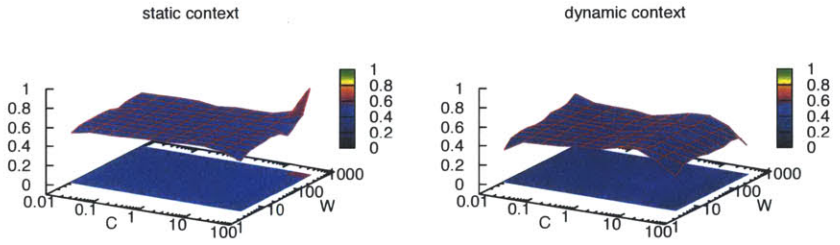
Our results demonstrate that autotuning performance can vary significantly depending on the selection of hyperparameter values. However, in a real-world setting the user cannot afford to run expensive experiments to determine which values work best for their particular program and architecture. For this reason, we performed an empirical investigation whether there exists a single assignment of C and W that works well across programs and architectures.

We used the score functions from Section 8.4.2 to find hyperparameters that maximized the mean score on all the benchmarks. We found that the hyperparameters $(C, W) = (5, 5)$ for the static context and $(C, W) = (5, 100)$ for the dynamic context maximized this score. The results are shown in Table 8.3. For the sake of illustration, we normalized each score with respect to the optimum for the given benchmark and scenario (Table 8.6(b)).

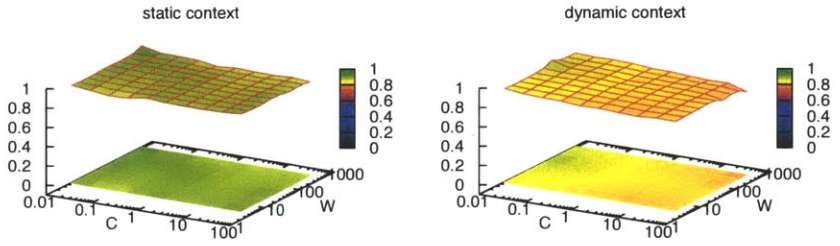
Despite fixing hyperparameter values across benchmarks, we measured a mean normalized score of 88.32% for the static and 82.45% for the dynamic context, which means that we only sacrificed less than 20% of the performance by not tuning hyperparameters on a per-benchmark and per-architecture basis. This result shows that the hyperparameters we found are likely to generalize to



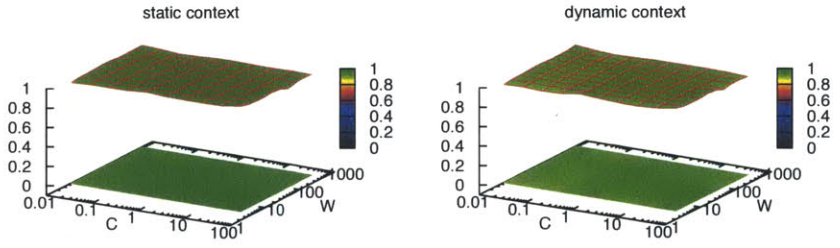
(a) Poisson on `Xeon8`



(b) Poisson on `AMD48`



(c) Image Compression on `Xeon8`



(d) Image Compression on `AMD48`

Figure 8.8: Measured scores for Poisson and Image Compression.

	static context		dynamic context	
	Xeon8	**AMD48**	**Xeon8**	**AMD48**
Sort	95.71%	100%	74.16%	61.12%
Bin Packing	85.61%	94.72%	67.42%	88.74%
Poisson	70.64%	71.09%	90.77%	96.07%
Image Compression	92.44%	96.35%	89.92%	91.42%

Table 8.3: Benchmark scores for the globally optimal values of hyperparameters normalized with respect to the best score for the given benchmark and scenario. The optimal hyperparameters were $C = 5$, $W = 5$ for the static context, and $C = 5$, $W = 100$ for the dynamic context. Mean normalized scores were 88.32% and 82.45% for the static and dynamic contexts, respectively.

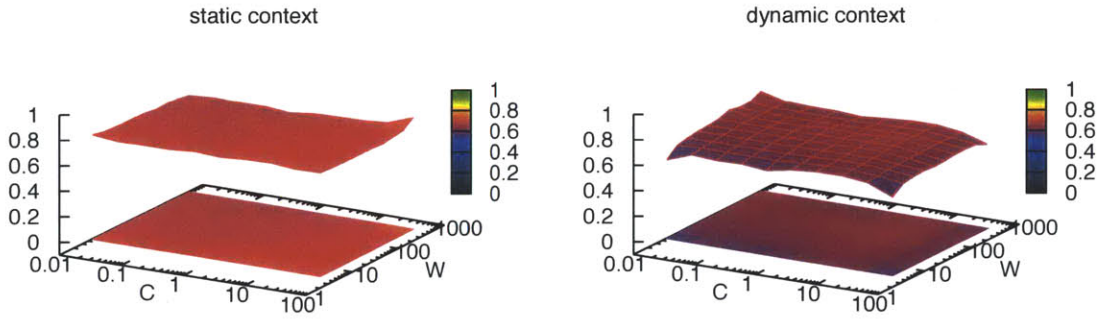


Figure 8.9: Scores for the static and dynamic scenarios averaged over the sort, Bin Packing, Poisson and Image Compression benchmarks and the Xeon8 and AMD48 architectures. The mean scores across all benchmarks, architectures and hyperparameter values were 0.6919 for the static and 0.6181 for the dynamic contexts.

other benchmarks, thus providing sensible defaults and removing the need to optimize them on a per-program basis. They also align with our results for individual benchmarks (Figure 8.6), where we found that exploration (moderate to high C, low W) is beneficial if we can afford the extra convergence time (static context), whereas exploitation (low to moderate C, high W) is preferred if average throughput and low convergence time are of interest (dynamic context).

Chapter 9

OpenTuner

Program autotuning can achieve better or more portable performance in a number of domains. However, autotuners themselves are rarely portable between projects, for a number of reasons: using a domain-informed search space representation is critical to achieving good results; search spaces can be intractably large and require advanced machine learning techniques; and the landscape of search spaces can vary greatly between different problems, sometimes requiring domain specific search techniques to explore efficiently.

In this chapter, we present OpenTuner, a new framework for building domain-specific program autotuners. A core concept in OpenTuner is the use of *ensembles* of search techniques. Many search techniques (both built in and user-defined) are run at the same time, each testing candidate configurations. Techniques which perform well by finding better configurations are allocated larger budgets of tests to run, while techniques which perform poorly are allocated fewer tests or disabled entirely. Techniques are able to share results using a common results database to constructively help each other in finding an optimal solution. To allocate tests between techniques we use an optimal solution to the *multi-armed bandit* problem using area under the curve credit assignment. Ensembles of techniques solve the large and complex search space problem by providing both a robust solutions to many types of large search spaces and a way to seamlessly incorporate domain specific search techniques.

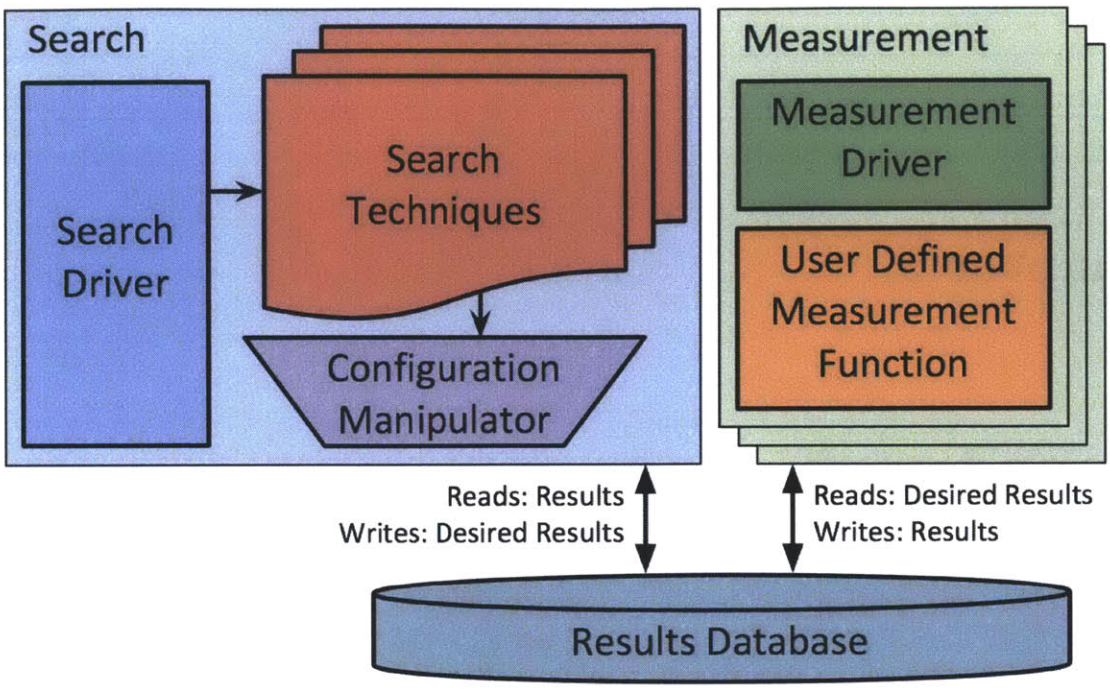


Figure 9.1: Overview of the major components in the OpenTuner framework.

9.1 The OpenTuner Framework

Our terminology reflects that the autotuning problem is cast as a search problem. The search space is made up of *configurations*, which are concrete assignments of a set of *parameters*. Parameters can be *primitive* such as an integer or *complex* such as a permutation of a list. When the performance, output accuracy, or other metrics of a configuration are measured (typically by running it in a domain-specific way), we call this measurement a *result*. *Search techniques* are methods for exploring the search space and make requests for measurement called *desired results*. Search techniques can change configurations using a user-defined *configuration manipulator*, which also includes *parameters* corresponding directly the parameters in the configuration. Some parameters include *manipulators*, which are opaque functions that make stochastic changes to a specific parameter in a configuration.

Figure 9.1 provides an overview of the major components in OpenTuner. The *search* process includes techniques, which use the user defined configuration manipulator in order to read and write configurations. The *measurement* processes evaluate candidate configurations using a user defined

measurement function. These two components communicate exclusively through a *results database* used to record all results collected during the tuning process, as well as the providing ability to perform multiple measurements in parallel.

9.1.1 OpenTuner Usage

To implement an autotuner with OpenTuner, first, the user must define the search space by creating a *configuration manipulator*. This configuration manipulator includes a set of parameter objects which OpenTuner will search over. Second, the user must define a *run* function which evaluates the fitness of a given configuration in the search space to produce a result. These must be implemented in a small Python program in order to interface with the OpenTuner API.

Figure 9.2 shows an example of using OpenTuner to search over the space of compiler flags to GCC in order to minimize execution time of the resulting program. In Section 9.2, we present results on an expanded version of this example which obtains up to 2.8x speedup over `-O3`.

This example tunes three types of flags to GCC. First it choses between the four optimization levels `-O0`, `-O1`, `-O2`, `-O3`. Second, for 176 flags listed on line 8, it decides between turning the flag on (with `-fFLAG`), off (with `-fno-FLAG`), or omitting the flag in order to let default value to take precedence. Including the default value as a choice is not necessary for completeness, but speeds up convergence and results in shorter command lines. Finally, it assigns a bounded integer value to the 145 parameters on line 15 with the `--param NAME=VALUE` command line option.

The method `manipulator` (line 23), is called once at startup and creates a `ConfigurationManipulator` object which defines the search space of GCC flags. All accesses to configurations by search techniques are done through the configuration manipulator. For optimization level, an `IntegerParameter` between 0 and 3 is created. For each flag, a `EnumParameter` is created which can take the values `on`, `off`, and `default`. Finally, for the remaining bounded GCC parameters, an `IntegerParameter` is created with the appropriate range.

The method `run` (line 40), implements the measurement function for configurations. First, the configuration is realized as specific command line to `g++`. Next, this `g++` command line is run to produce an executable, `tmp.bin`, which is then run using `call_program`. `Call_program` is

```python
import opentuner
from opentuner import ConfigurationManipulator
from opentuner import EnumParameter
from opentuner import IntegerParameter
from opentuner import MeasurementInterface
from opentuner import Result

GCC_FLAGS = [
  'align-functions', 'align-jumps', 'align-labels',
  'branch-count-reg', 'branch-probabilities',
  # ... (176 total)
]

# (name, min, max)
GCC_PARAMS = [
  ('early-inlining-insns', 0, 1000),
  ('gcse-cost-distance-ratio', 0, 100),
  # ... (145 total)
]

class GccFlagsTuner(MeasurementInterface):

  def manipulator(self):
    """
    Define the search space by creating a
    ConfigurationManipulator
    """
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(
      IntegerParameter('opt_level', 0, 3))
    for flag in GCC_FLAGS:
      manipulator.add_parameter(
        EnumParameter(flag,
                      ['on', 'off', 'default']))
    for param, min, max in GCC_PARAMS:
      manipulator.add_parameter(
        IntegerParameter(param, min, max))
    return manipulator

  def run(self, desired_result, input, limit):
    """
    Compile and run a given configuration then
    return performance
    """
    cfg = desired_result.configuration.data
    gcc_cmd = 'g++ raytracer.cpp -o ./tmp.bin'
    gcc_cmd += ' -O{0}'.format(cfg['opt_level'])
    for flag in GCC_FLAGS:
      if cfg[flag] == 'on':
        gcc_cmd += ' -f{0}'.format(flag)
      elif cfg[flag] == 'off':
        gcc_cmd += ' -fno-{0}'.format(flag)
    for param, min, max in GCC_PARAMS:
      gcc_cmd += ' --param {0}={1}'.format(
        param, cfg[param])

    compile_result = self.call_program(gcc_cmd)
    assert compile_result['returncode'] == 0
    run_result = self.call_program('./tmp.bin')
    assert run_result['returncode'] == 0
    return Result(time=run_result['time'])

if __name__ == '__main__':
  argparser = opentuner.default_argparser()
  GccFlagsTuner.main(argparser.parse_args())
```

Figure 9.2: GCC/G++ flags autotuner using OpenTuner.

a convince function which runs and measures the execution time of the given program. Finally, a `Result` is constructed and returned, which is a database record type containing many other optional fields such as `time`, `accuracy`, and `energy`. By default OpenTuner minimizes the `time` field, however this can be customized.

9.1.2 Search Techniques

To provide robust search, OpenTuner includes techniques that can handle many types of search spaces and runs a collection of search techniques at the same time. Techniques which perform well are allocated more tests, while techniques which perform poorly are allocated fewer tests. Techniques share results through the results database, so that improvements made by one technique can benefit other techniques. OpenTuner techniques are meant to be extended. Users can define custom techniques which implement domain-specific heuristics and add them to *ensembles* of pre-defined techniques.

Ensembles of techniques are created by instantiating a *meta technique*, which is a technique made up of a collection of other techniques. The OpenTuner search driver interacts with a single *root* technique, which is typically a meta technique. When the meta technique gets allocated tests, it incrementally decides how to divide theses tests among its sub-techniques. OpenTuner contains an extensible class hierarchy of techniques and meta techniques, which can be combined together and used in autotuners.

AUC Bandit Meta Technique

In addition to a number of simple meta techniques, such as round robin, OpenTuner's core meta technique used in results is the *multi-armed bandit with sliding window, area under the curve credit assignment* (AUC Bandit) meta technique. A similar technique was used in [110] in the different context of online operator selection. It is based on an optimal solution to the multi-armed bandit problem [62]. The multi-armed bandit problem is the problem of picking levers to pull on a slot machine with many arms each with an unknown payout probability. It encapsulates a fundamental

trade-off between *exploitation* (using the best known technique) and *exploration* (estimating the performance of all techniques).

A detailed explanation of the bandit algorithm and AUC credit assignment is provided in Section 8.2.4. The same algorithm is used here (though in a different context).

Other Techniques

OpenTuner includes implementations of the techniques: differential evolution; many variants of Nelder Mead and Torczon hill climbers; a number of evolutionary mutation techniques; pattern search; particle swarm optimization; and random search. These techniques span a range of strategies and are each biased to perform best in different types of search spaces. They also each contain many settings which can be configured to change their behavior. Each technique has been modified so that with some probability it will use information found by other techniques if other techniques have discovered a better configuration.

The default meta technique, used in results and meant to be robust, uses an AUC Bandit meta technique to combine greedy mutation, differential evolution, and two hill climber instances.

9.1.3 Configuration Manipulator

The configuration manipulator provides a layer of abstraction between the search techniques and the raw configuration structure. It is primarily responsible for managing a list of parameter objects, each of which can be used by search techniques to read and write parts of the underlying configuration.

The default implementation of the configuration manipulator uses a fixed list of parameters and stores the configuration as a dictionary from parameter name to parameter-dependant data type. The configuration manipulator can be extended by the user either to change the underlying data structure used for configurations or to support a dynamic list of parameters that is dependant on the configuration instance.

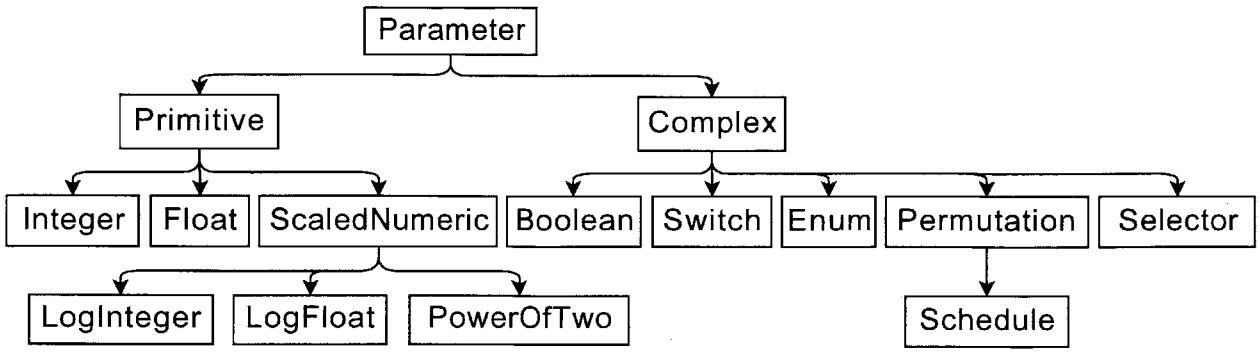


Figure 9.3: Hierarchy of built in parameter types. User defined types can be added at any point below Primitive or `Complex` in the tree.

Parameter Types

Figure 9.3 shows the class hierarchy of built-in parameter types in OpenTuner. Each parameter type is responsible for interfacing between the raw representation of a parameter in the configuration and standardized view of that parameter presented to the search techniques. Parameter types can be extended both to change the underlying representation, and to change the abstraction provided to search techniques to cause a parameter to be search in different ways.

From the viewpoint of search techniques there are two main types of parameters, each of which provides a different abstraction to the search techniques:

Primitive parameters present a view to search techniques of a numeric value with an upper and lower bound. These upper and lower bounds can be dependant on the configuration instance.

The built in parameter types `Float` and `LogFloat` (and similarly `Integer` and `LogInteger`) both have identical representations in the configuration, but present a different view of the underlying value to the search techniques. `Float` is presented directly to to search techniques, while `LogFloat` presents a log scaled view of the underlying parameter to search techniques. To a search technique, halving and doubling a log scaled parameter are changes of equal magnitude. Log scaled variants of parameters are often better for parameters such as block sizes where fixed changes in values have diminishing effects the larger the parameter becomes. `PowerOfTwo` is a commonly used special case, similar to `LogInteger`, where the legal values of the parameter are restricted to powers of two.

Complex parameters present a more opaque view to search techniques. Complex parameters have a variable set of manipulation operators (manipulators) which make stochastic changes to the underlying parameter. These manipulators are arbitrary functions defined on the parameter which can make high level type dependant changes. Complex parameters are meant to be easily extended to add domain specific structures to the search space.

The built in parameter types `Boolean`, `Switch`, and `Enum` could theoretically also be represented as primitive parameters, since they each can be translated directly to a small integer representation. However, in the context of search techniques they make more sense as complex parameters. The reason for this is that for primitive parameters search techniques will attempt to follow gradients. These parameter types are unordered collections of values for which no gradients exist. Thus, the complex parameter abstraction is a more efficient representation to search over.

The `Permutation` parameter type assigns an order to a given list of values and has manipulators which make various types of random changes to the permutation. A `Schedule` parameter is a `Permutation` with a set of dependencies that limit the legal order. Schedules are implemented as a permutation that gets topologically sorted after each change. Finally, a `Selector` parameter is a special type of tree which is used to define a mapping from an integer input to an enumerated value type.

In addition to these primary primitive and complex abstractions for parameter types, there are a number of derived ways that search techniques will interact with parameters in order to more precisely convey intent. These are additional methods on parameter which contain default implementations for both primitive and complex parameter types. These methods can optionally be overridden for specific parameters types to improve search techniques. Parameter types will work without these methods being overridden, however implementing them can improve results.

As an example, a common operation in many search techniques is to add the difference between configuration A and B to configuration C. This is used both in differential evolution and many hill climbers. Complex parameters have a default implementation of this indent which compares the value of the parameter in the 3 configurations: if $A = B$, then there is no difference and the result is C; similarly, if $B = C$, then A is returned; otherwise a change should be made so random

manipulators are called. This works in general, however for individual parameter types there are often better interpretations. For example with permutations, one could calculate the positional movement of each item in the list an calculate a new permutation by applying these movements again.

9.1.4 Objectives

OpenTuner supports multiple user defined objectives. Result records have fields for `time`, `accuracy`, `energy`, `size`, `confidence`, and user defined data. The default objective is to minimize time. Many other objectives are supported, such as: maximize accuracy; threshold accuracy while minimizing time; and maximize accuracy then minimize size. The user can easily define their own objective by defining comparison operators and display methods on a subclass of Objective.

9.1.5 Search Driver and Measurement

OpenTuner is divided into two submodules, search and measurement. The search driver and measurement driver in each of these modules orchestrate most of the framework of the search process. These two modules communicate only through the results database. The measurement module is minimal by design and is primarily a wrapper around the user defined measurement function which creates results from configurations.

This division between search and measurement is motivated by a number of different factors:

- To allow parallelism between multiple search measurement processes, possibly across different machines. Parallelism is most important in the measurement processes since in most autotuning applications measurement costs dominate. To allow for parallelism the search driver will make multiple requests for desired results without waiting for each request to be fulfilled. If a specific technique is blocking waiting for results, other techniques in the ensemble will used to fill out requests to prevent idle time.

- The separation of the measurement modules is desirable to support online learning and sideline learning. In these setups, autotuning is not done before deployment of an application, but is done online as an application is running or during idle time. Since the measurement module

is minimal by design, it can be replaced by an domain specific online learning module which periodically examines the database to decide which which configuration to use and records performance back to the database.

- Finally, in many embedded or mobile settings which require constrained measurement environments it is desirable to have a minimal measurement module which can easily be re-implemented in other languages without needing to modify the majority of the OpenTuner framework.

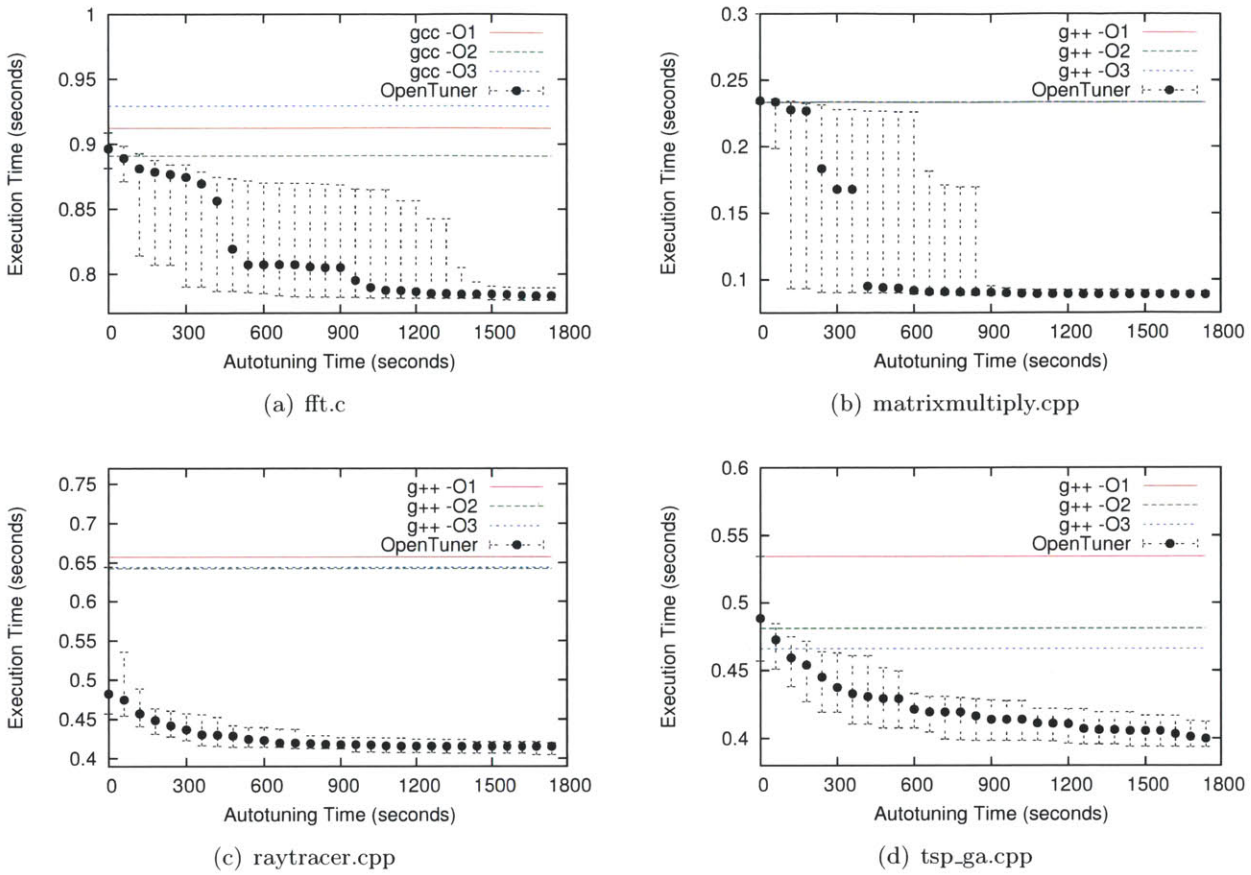
9.1.6 Results Database

The results database is a fully featured SQL database. All major database types are supported, and SQLite is used if the user has not configured a database type so that no setup is required. It allows different techniques to query and share results in a variety of ways and is useful for introspection about the performance of techniques across large numbers of runs of the autotuner.

9.2 Experimental Results

Project	Benchmark	Possible Configurations
GCC/G++ Flags	*all*	10^{806}
Halide	Blur	10^{52}
Halide	Wavelet	10^{44}
HPL	*n/a*	$10^{9.9}$
PetaBricks	Poisson	10^{3657}
PetaBricks	Sort	10^{90}
PetaBricks	Strassen	10^{188}
PetaBricks	TriSolve	10^{1559}
Stencil	*all*	$10^{6.5}$
Unitary	*n/a*	10^{21}

Figure 9.4: Search space sizes in number of possible configurations, as represented in OpenTuner.



(a) fft.c

(b) matrixmultiply.cpp

(c) raytracer.cpp

(d) tsp_ga.cpp

Figure 9.5: **GCC/G++ Flags**: Execution time (lower is better) as a function of autotuning time. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles. Note that in (b) the O1/O2/O3 and in (c) the O2/O3 lines are on top of each other and may be difficult to see.

We validated OpenTuner by using it to implemented autotuners for six distinct projects. This section describes these six projects, the autotuners we implemented, and presents results comparing to prior practices in each project.

Figure 9.4 lists, for each benchmark, the number of distinct configurations that can be generated by OpenTuner. This measure is not perfect because some configurations may be semantically equivalent and the search space depends on the representation chosen in OpenTuner. It does, however, provide a sense of the relative size of each search space, which is useful as a first approximation of tuning difficulty.

9.2.1 GCC/G++ Flags

The GCC/G++ flags autotuner is described in detail in Section 9.1.1. There are a number features that were omitted from the earlier example code for simplicity, which are included in the full version of the autotuner.

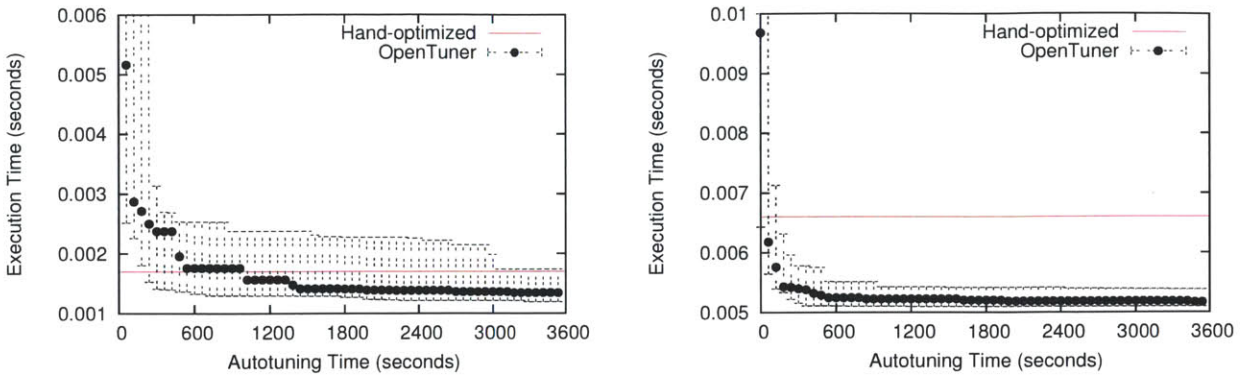
First, we added error checking to gracefully handle the compiler or the output program hanging, crashing, running out of memory, or otherwise going wrong. Our tests uncovered a number of bugs in GCC which triggered internal compiler errors and we implemented code to detect, diagnose, and avoid error-causing sets of flags. We are submitting bug reports for these crashes to the GCC developers.

Second, instead of using a fixed list of flags and parameters (which the example does for simplicity), our full autotuner automatically extracts the supported flags from `g++ --help=optimizers`. Parameters and legal ranges are extracted automatically from `params.def` in the GCC source code.

Additionally, there were a number of smaller features such as: time limits to abort slow tests which will not be optimal; use of `LogInteger` parameter types for some values; a `save_final_config` method to output the final flags; and many command line options to autotuner behavior.

We ran experiments using `gcc 4.7.3-1ubuntu1`, on an 8 total core, 2-socket Xeon E5320. We allowed flags such a `-ffast-math` which can change rounding / NaN behavior of floating point numbers and have small impacts on program results. We still observe speedups with these flags removed.

For target programs to optimize we used: A fast Fourier transform in C, fft.c, taken from the SPLASH2 [158] benchmark suite; A C++ template matrix multiply, `matrixmultiply.cpp`, written by Xiang Fan [60] (version 3); A C++ ray tracer, `raytracer.cpp`, taken from the scratchpixel website [114]; and a genetic algorithm to solve the traveling salesman program in C++, `tsp_ga.cpp`, by Kristoffer Nordkvist [104], which we modified to run deterministically. These programs were chosen to span a range from highly optimized codes, like `fft.c` which contains cache aware tiling



(a) Blur

(b) Wavelet

Figure 9.6: **Halide**: Execution time (lower is better) as a function of autotuning time. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles.

and threading, to less optimized codes, like `matrixmultiply.cpp` which contains only a transpose of one of the inputs.

Figure 9.5 shows the performance for autotuning GCC/G++ flags on four different sample programs. Final speedups ranged from 1.15× for FFT to 2.82× for matrix multiply. Examining the frequencies of different flags in the final configurations, we can see some patterns and some differences between the benchmarks. In all programs `-funsafemath-optimizations` (and related flags) and `-O3` flags were very common. There were a number of flags that were only common only in specific benchmarks:

- `matrixmultiply.cpp`: `-fvariable-expansion-in-unroller` and `-ftree-vectorize`
- `raytracer.cpp`: `-fno-reg-struct-return`
- `fft.c`: `--param=allow-packed-store-data-races=1`, `-frerun-cse-after-loop`, and `-funroll-all-loops`
- `tsp_ga.cpp`: `--param=use-canonical-types=1` and `-fno-schedule-insns2`.

However these most common flags alone do not account for all of the speedup. Full command lines found contained typically 200 to 300 options and are difficult to understand by hand.

9.2.2 Halide

Halide [120,121] is a domain-specific language and compiler for image processing and computational photography, specifically targeted towards image processing *pipelines* that contain several stages. Halide separates the scheduling of the image processing stages from the expression of the kernels themselves, allowing expert programmers to dictate complex schedules that result in high performance.

The Halide project originally integrated an autotuner, which was removed from the project because it became too complex to maintain and was rarely used in practice. We hope that our new OpenTuner-based autotuner for Halide, presented here, will be easier to maintain, both because it benefits from some of the lessons learned from the original autotuner and because it provides a clear separation between the search techniques and the definition of the search space. Unfortunately, the original Halide autotuner cannot be used as a baseline to compare against, because the Halide code base has changed too much since its removal.

The autotuning problem in Halide is to synthesize execution schedules that control how Halide generates code. As an example, the hand-tuned schedule (against which we compare our autotuner) for the blur example is:

```
1 blur_y.split(y, y, yi, 8)
2       .parallel(y)
3       .vectorize(x, 8);
4 blur_x.store_at(blur_y, y)
5       .compute_at(blur_y, yi)
6       .vectorize(x, 8);
```

`blur_y(x, y)` and `blur_y(x, y)` are Halide functions in the program. The scheduling operators which the autotuner can use to synthesize schedules are:

- `split` introduces a new variable and loop nest by adding a layer of blocking. We limit the number of splits to at most 4 per dimension of each function, which is sufficient in practice. We represent each of these splits as a `PowerOfTwoParameter`, where setting the size of the split to 1 corresponds to not using the split operator.

- `parallel`, `vectorize`, and `unroll` cause the loop nest associated with a given variable in the function to be executed in parallel, SSE vectorized, or unrolled. OpenTuner represents these operators as an `EnumParameter` for each variable/function pair including temporary variables possibly introduced by splits to decide on an operator, including no operator as a choice.

- `reorder` / `reorder_storage` take a list of variables and reorganizes the loop nest order or storage order for those variables. We represent this is as a `PermutationParameter`, which includes all possible variables introduced by splits.

- `compute_at` / `store_at` cause the execution or storage for a given function to be embedded inside of the loop nest of a different function. We represent this as an `EnumParameter` with all legal function/variable pairs and special tokens for global and inline as options.

The most difficult parameter to search is `compute_at` because most choices combinations for this parameter will create invalid schedules are are rejected by the compiler. We created a custom domain specific technique which attempted to create more legal schedules by biasing the search of the parameter.

Figure 9.6 presents results for blur and the inverse Daubechies wavelet transform written in Halide. For both of these examples OpenTuner is able to create schedules that beat the hand optimized schedules shipping with the Halide source code. Results were collected on an 8-core Core i7 920 processor using a development build of Halide.

9.2.3 High Performance Linpack

The High Performance Linpack benchmark [57] is used to evaluate floating point performance of machines ranging from small multiprocessors to large-scale clusters, and is the evaluation criterion for the Top 500 [145] supercomputer benchmark. The benchmark measures the speed of solving a large random dense linear system of equations using distributed memory. Achieving optimal performance requires tuning about fifteen parameters, including matrix block sizes and algorithmic parameters. To assist in tuning, HPL includes a built in autotuner that uses exhaustive search over user-provided discrete values of the parameters.

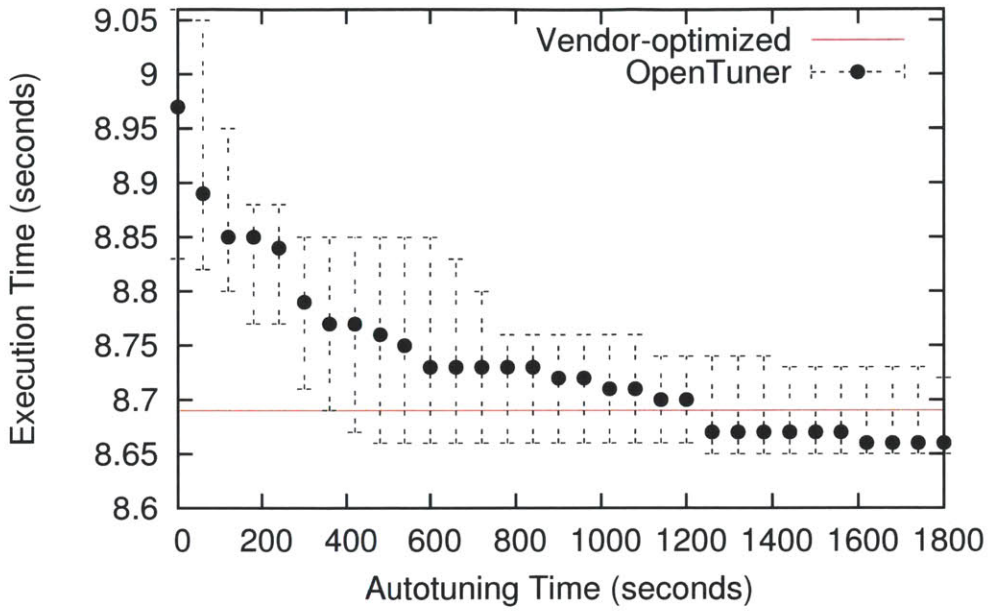
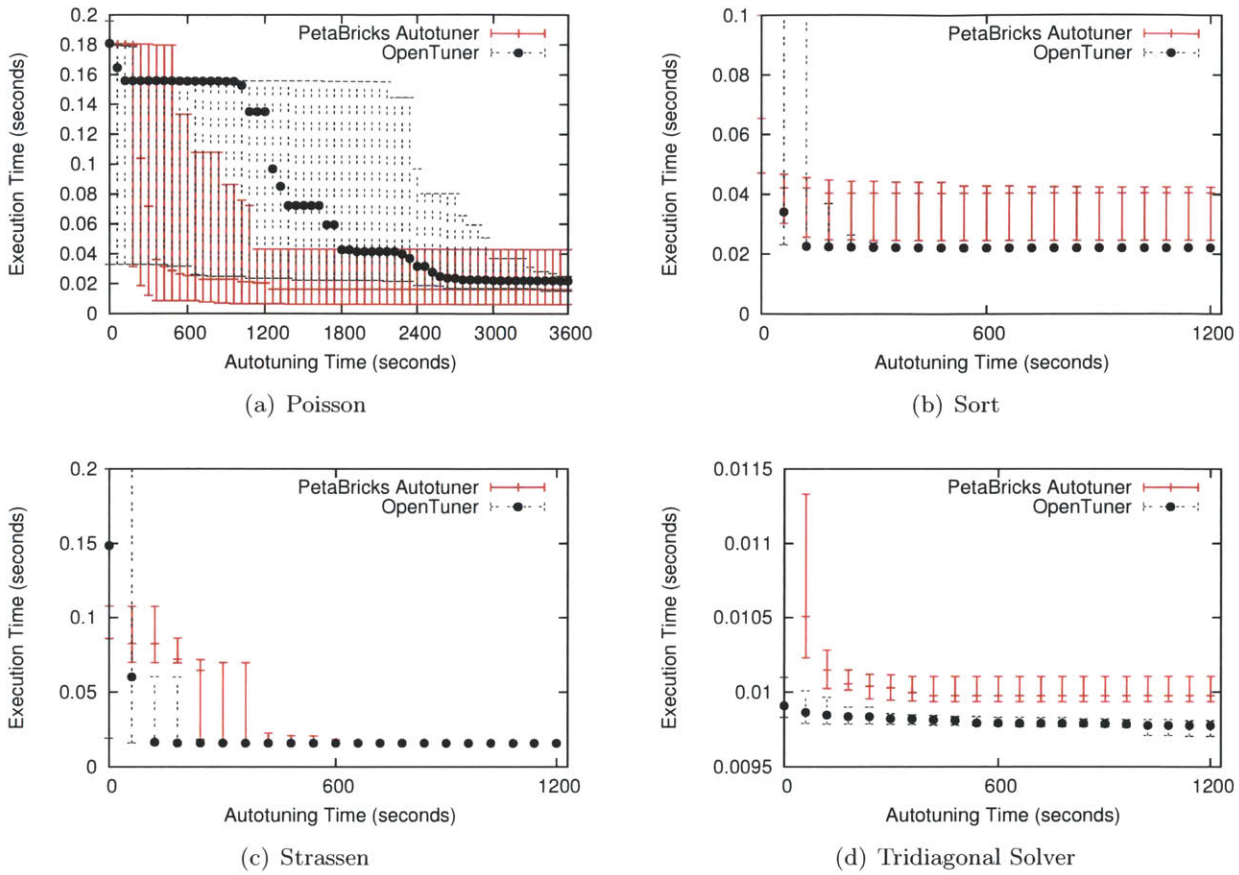


Figure 9.7: **High Performance Linpack**: Execution time (lower is better) as a function of autotuning time. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles.

We run HPL on a 2.93 GHz Intel Sandy Bridge quad-core machine running Linux kernel 3.2.0, compiled with GCC 4.5 and using the Intel Math Kernel Library (MKL) 11.0 for optimized math operations. For comparison purposes, we evaluate performance relative to Intel's optimized HPL implementation [1]. We encode the input tuning parameters for HPL as naïvely as possible, without using any machine-specific knowledge. For most parameters, we utilize `EnumParameter` or `SwitchParameter`, as they generally represent discrete choices in the algorithm used. The major parameter that controls performance is the blocksize of the matrix; this we represent as an `IntegerParameter` to give as much freedom as possible for the autotuner for searching. Another major parameter controls the distribution of the matrix onto the processors; we represent this by enumerating all 2D decompositions possible for the number of processors on the machine.

Figure 9.7 shows the results of 30 tuning runs using OpenTuner, compared with the vendor-provided performance. The median performance across runs, after 1200 seconds of autotuning, exceeds the performance of Intel's optimized parameters. Overall, OpenTuner obtains a best

[1] Available at `http://software.intel.com/en-us/articles/intel-math-kernel-library-linpack-download`.



(a) Poisson

(b) Sort

(c) Strassen

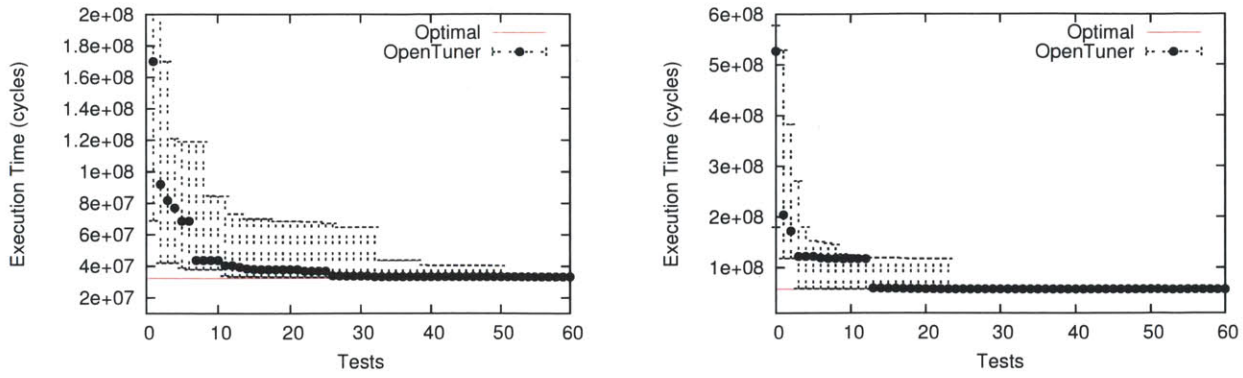
(d) Tridiagonal Solver

Figure 9.8: **PetaBricks**: Execution time (lower is better) as a function of autotuning time. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles.

performance of 86.5% of theoretical peak performance on this machine, while exploring a miniscule amount of the overall search space. Furthermore, the blocksize chosen is not a power of two, and is generally a value unlikely to be guessed for use in hand-tuning.

9.2.4 PetaBricks

Figure 9.8 compares OpenTuner to the PetaBricks autotuner on 4 PetaBricks benchmarks, described in Chapter 4. The PetaBricks autotuner uses a different strategy, described in Chapter 6, that starts with tests on very small problem inputs and incrementally works up to full sized inputs. In all cases, the autotuners arrive at similar solutions, and for Strassen, the exact same solution. For



(a) Laplacian

(b) Divergence

Figure 9.9: **Stencil**: Execution time (lower is better) as a function of tests. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles.

Sort and Tridiagonal Solver, OpenTuner beats the native PetaBricks autotuner, while for Poisson the PetaBricks autotuner arrives at a better solution, but has much higher variance.

The Poisson equation solver (Figure 9.8(a)) presents the most difficult search space. The search space for Poisson in PetaBricks is described in detail in Chapter 5. It is a variable accuracy benchmark where the goal of the autotuner is to find a solution that provides 8-digits of accuracy while minimizing time. All points in Figure 9.8(a) satisfy the accuracy target, so we do not display accuracy. OpenTuner uses the *ThresholdAccuracyMinimizeTime* objective described in Section 9.1.4. The Poisson search space selects between direct solvers, iterative solvers, and multigrid solvers where the shape of the multigrid V-cycle/W-cycle is defined by the autotuner. The optimal solution is a poly-algorthm composed of multigrid W-cycles. However, it is difficult to obtain 8-digits of accuracy with randomly generated multigrid cycle shapes, but is easy with a direct solver (which solves the problem exactly). This creates a large "plateau" which is difficult for the autotuners to improve upon, and is shown near 0.16. The native PetaBricks autotuner is less affected by this plateau because it constructs algorithms incrementally bottom up; however the use of these smaller input sizes causes larger variance as mistakes early on get amplified.

9.2.5 Stencil

In [86], the authors describe a generalized system for autotuning memory-bound stencil computations on modern multicore machines and GPUs. By composing domain-specific transformations, the authors explore a large space of implementations for each kernel; the original autotuning methodology involves exhaustive search over thousands of implementations for each kernel.

We obtained the raw execution data, courtesy of the authors, and use OpenTuner instead of exhaustive search on the data from a Nehalem-class 2.66 GHz Intel Xeon machine, running Linux 2.6. We compare against the optimal performance obtained by the original autotuning system through exhaustive search. The search space for this problem involves searching for parameters for the parallel decomposition, cache and thread blocking, and loop unrolling for each kernel; to limit the impact of control flow and cache misalignment, these parameters depend on one another (for example, the loop unrolling will be a small integer divisor of the thread blocking). We encode these parameters as `PowerOfTwoParameter`s but ensure that invalid combinations are discarded.

Figure 9.9 shows the results of using OpenTuner for the Laplacian and divergence kernel benchmarks, showing the median performance obtained over 30 trials as a function of the number of tests. OpenTuner is able to obtain peak performance on Laplacian after less than 35 tests of candidate implementations and 25 implementations for divergence; thus, using OpenTuner, less than 2% of the search space needs to be explored to reach optimal performance. These results show that even for problems where exhaustive search is tractable (though it may take days), OpenTuner can drastically improve convergence to the optimal performance with little programmer effort.

9.2.6 Unitary Matrices

As a somewhat different example, we use OpenTuner in an example from physics, namely the quantum control problem of synthesizing unitary matrices in $SU(2)$ in optimal time, using a finite control set composed of rotations around two non-parallel axes. (Such rotations generate the complete space $SU(2)$.)

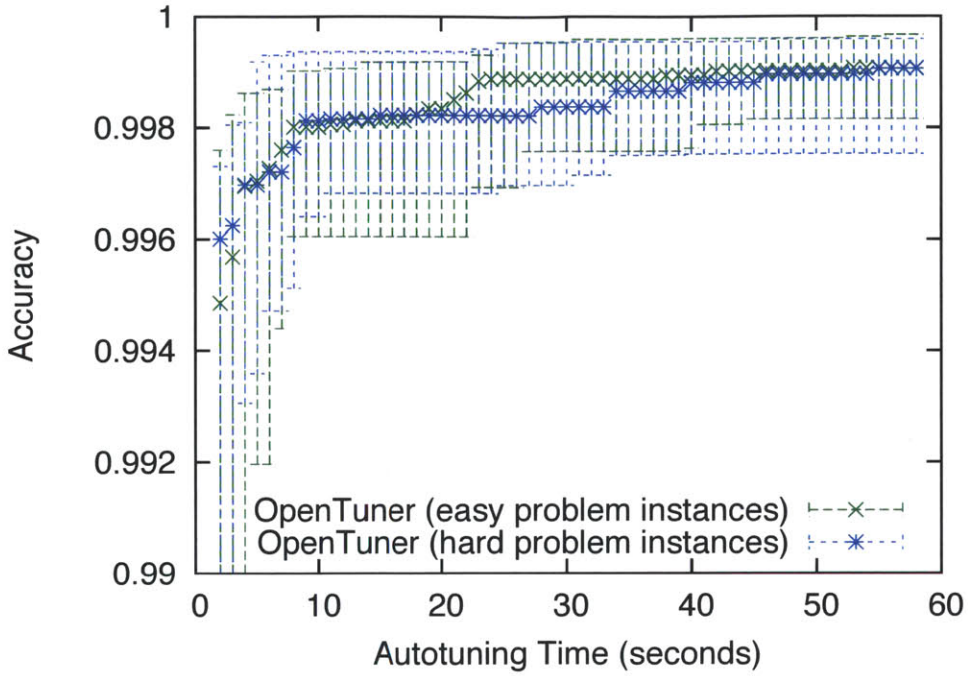


Figure 9.10: **Unitary**: Accuracy (higher is better) as a function of autotuning time. Aggregated performance of 30 runs of OpenTuner, error bars indicate median, 20th, and 80th percentiles.

Unlike other examples, which use OpenTuner as a traditional autotuner to optimize a program, the Unitary example uses OpenTuner to perform a search over the problem space as a subroutine at runtime in a program. The problem has a fixed set of operators (or controls), represented as matrices, and the goal is to find a sequence of operators that, when multiplied together, produce a given target matrix. The objective function is an accuracy value defined as a function of the distance of the product of the sequence to the goal (also called the *trace fidelity*).

Figure 9.10 shows the performance of the Unitary example on both easy and hard instances of the problem. For both types of problem instance OpenTuner is able to meet the accuracy target within the first few seconds. This is example shows that OpenTuner can be used for more types of search problems than just program autotuning.

9.2.7 Results Summary

While implementing these six autotuners in OpenTuner, the biggest lesson we learned reinforced a core message of this thesis of the need for domain-specific representations and domain-specific

search techniques in autotuning. As an example, the initial version of the PetaBricks autotuner we implemented just used a point in high dimensional space as the configuration representation. This generic mapping of the search space did not work at all. It produced final configurations an order of magnitude slower than the results presented from our autotuner that uses selector parameter types. Similarly, Halide's search space strongly motivates domain specific techniques that make large coordinated jumps, for example, swapping scheduling operators on x and y across all functions in the program. We were able to add domain-specific representations and techniques to OpenTuner at a fraction of the time and effort of building a fully custom system for that project. OpenTuner was able to seamlessly integrate the techniques with its ensemble approach.

Chapter 10

Related Work

PetaBricks primarily differentiates from prior work in that it is the first programming language to incorporate algorithmic choice, variable accuracy, and autotuning. Most of this chapter will focus on related projects that use autotuning. A major different between PetaBricks and the majority of other projects is the size of the search spaces. PetaBricks has massive and complex search spaces, containing up to 10^{3000} configuration and difficult non-linear dependencies created by algorithmic selectors. The majority of related projects are dealing with far simpler search spaces. Often these search spaces are small enough to search exhaustively. In other cases the search spaces are simple enough that hill climbers (such as Nelder-Mead [103]) suffice. We believe hat these search spaces are small both because there was not a good way to represent algorithmic choice and, in some cases, because programmers perform manual pruning of their search space in order to fit the search space to a limited search technique. PetaBricks and OpenTuner can expand the scope of autotuning by creating and then being able to search far more complex search spaces than have previously been used.

10.1 Autotuning

A number of offline empirical autotuning frameworks have been developed for building efficient, portable libraries in specific domains; selected projects and techniques used are summarized in Figure 10.1. PHiPAC [30] is an autotuning system for dense matrix multiply, generating portable

Package	Domain	Search Method
Active Harmony [137]	Runtime System	Nelder-Mead
ATLAS [155]	Dense Linear Algebra	Exhaustive
FFTW [66]	Fast Fourier Transform	Exhaustive/Dynamic Prog.
Insieme [84]	Compiler	Differential Evolution
Milepost GCC [71]	Compiler	IID Model + Central DB
OSKI [151]	Sparse Linear Algebra	Exhaustive+Heuristic
PATUS [42]	Stencil Computations	Nelder-Mead or Evolutionary
SEEC / Heartbeats [77, 100]	Runtime System	Control Theory
Sepya [87]	Stencil Computations	Random-Restart Gradient Ascent
SPIRAL [118]	DSP Algorithms	Pareto Active Learning

Figure 10.1: Summary of selected related projects using autotuning

C code and search scripts to tune for specific systems. ATLAS [155, 156] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LAPACK. FFTW [65,66] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include: SPARSITY [80] for sparse matrix computations, SPIRAL [63, 118, 147] for digital signal processing, UHFFT [4] for FFT on multicore systems, PATUS [42] and Sepya [87] for stencil computations, OSKI [151] for sparse matrix kernels, and autotuning frameworks for optimizing sequential [95, 96] and parallel [108] sorting algorithms, PHiPAC [30] is an autotuning system for dense matrix multiply, generating portable C code and searching scripts to tune for specific systems, UHFFT [4] for FFT on multicore systems, and autotuning frameworks for optimizing sequential [96] and parallel [108] sorting algorithms. 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 [30] is an autotuning system for dense matrix multiply. ATLAS [155] utilizes empirical autotuning to produce a cache-contained matrix multiply. FFTW [66] uses empirical autotuning to combine solvers for FFTs. A system by Kessler *et al.* [7, 89] automatically composes algorithms using emperical techniques. Other autotuning systems include SPARSITY [80] for sparse matrix computations, SPIRAL [63,118,147] for digital signal processing, UHFFT [4] for FFT on multicore systems, and OSKI [151] for sparse matrix kernels. ActiveHarmony [45, 144] provides a general framework for tuning configurable

libraries and exploring different compiler optimizations. Diniz and Rinard [54] present a system to automatically switch between a fixed number of compiler optimization settings at runtime for different blocks of code using alternating sampling an production phases. In addition to these systems, various performance models and tuning techniques [35, 91, 157, 164] have been proposed to evaluate and guide automatic performance tuning. There exists a large variety of work related to PetaBrick's approach of autotuning computer programs. PHiPAC [30] is an autotuning system for dense matrix multiply, generating portable C code and search scripts to tune for specific systems. ATLAS [155] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LAPACK. FFTW [66] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [80] for sparse matrix computations, SPIRAL [118] for digital signal processing, UHFFT [4] for FFT on multicore systems, and OSKI [151] for sparse matrix kernels.

In addition to these systems, various performance models and tuning techniques [35, 91, 150, 157, 162, 164] have been proposed to evaluate and guide automatic performance tuning.

The area of iterative compilation contains many projects that use different machine learning techniques to optimize lower level compiler optimizations [1, 5, 112]. These projects change both the order that compiler passes are applied and the types of passes that are applied. Moss and Page [101] introduce techniques for exploring many different fine grained instruction orderings. Donadio et al. [56] introduced a language for expressing families of loop transformations with different parameters.

MILEPOST GCC [2, 39, 40, 58, 68–71, 102, 142] and cTuning.org is a particularly notable project in the area of iterative compilation and collective optimization. They maintain a high quality machine learning enabled version GCC that uses a centralized knowledge database. They are unquestionably a research leader in the area optimizing traditional compiler optimizations. Their later focus is on building collective knowledge bases to share tuning information between different machine types and from many sources.

There are a number of systems that provide high-level abstractions to ease the burden of programming adaptive applications. STAPL [141] is an C++ template library that support

adaptive algorithms and autotuning. Paluska et al. propose a programming framework [111] that allows programmers to specify goals of application behavior and techniques to satisfy those goals. The application hierarchically decomposes different situations and adapts to them dynamically. Andersson et al. [7] and Kessler et al. [89] provide a framework for composing parallel algorithmic components.

Additionally, there has been a large amount of work [22,55,148,149] 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 lead to an improvement of performance when applied. Conversely, autotuning searches the space of many available program variations without a priori knowledge of which configurations will perform better.

In the dynamic autotuning space, there have been a number of systems developed [24,27,38,77, 78,88,137] 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 [78] 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 [76] to provide feedback to the control system. A similar technique is employed in [77], where a simpler heuristic-based controller dynamically adjusts the degree of loop perforation performed on a target application to trade QoS for performance. The principle theme of these studies is to react to dynamic changes in the system behavior rather than proactively adapt algorithm configurations based on the characteristics of program inputs.

Atune-IL [129] allows programmers to annotate their parallel programs with different parameters. Their system then uses autotuning techniques to set these parameters and improve performance.

FLAME [74] is a domain-specific tuning system, providing a formal approach to the design of linear algebra methods. The system produces C and Fortran implementations from high-level specifications via code generation.

Yi and Whaley proposed a framework [161] to automate the production of optimized general-purpose library kernels. An embedded scripting language, POET, is used to describe custom optimizations for an algorithm. Specification files written in POET are fed into a transformation engine, which then generates and tunes different implementations. The POET system requires programmers to describe specific algorithmic optimizations, rather than allowing the compiler to explore choices automatically.

SPL [159] is a domain-specific language and compiler system targeted to digital signal processing. The compiler takes signal processing transforms represented by SPL formulas and explores different transformations and optimizations to produce efficient C and Fortran code. However, the SPL system was designed only for tuning sequential machines.

A number of studies have considered program inputs in library constructions [29, 64, 81, 117, 140, 154]. They concentrate on some specific library functions (e.g., FFT, sorting) while the algorithmic choices in these studies are limited. Tian and others have proposed an input-centric framework [143] for dynamic program optimizations and showed the benefits in enhancing Just-In-Time compilation. Jung and others have considered inputs when selecting the appropriate data structures to use [85]. Several recent studies have explored the influence of program inputs on GPU program optimizations [97, 127].

10.2 Variable Accuracy

Techniques such as Loop Perforation [100], Code Perforation [77], and Task Skipping [124, 125] automatically transform existing computations and/or programs to achieve higher performance. 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). In contrast, our system provides a new language and compiler that enables programmers to safely write 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 [78] 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.

Our work also bears similarities to the Green system [24], 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.

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 for computational tractability [165]. 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 [79], 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.

The way in which PetaBricks defines correctness for variable accuracy in terms of an accuracy metric and performs a search for correct programs is similar to Program Synthesis. Program Synthesis [132–135] defines the correctness of a program in terms of assertions, such as equivalence between a "sketch" and a specification. The search is performed using a SAT solver and guarantees correctness for all inputs. This is a more powerful guarantee than the statistical guarantees of PetaBricks, however, the technique does not scale as well to large programs.

10.3 Multigrid

Some multigrid solvers using algorithmic choice have been presented in the past. SuperSolvers [28] is not an autotuner but rather a system for designing composite algorithms that leverage multiple algorithmic choices to solve sparse linear systems reliably. Our approach differs by the use of tuning algorithmic choice at different levels of the multigrid hierarchy and the use of tuned subproblems during recursion. Unfortunately, no direct performance comparison was possible for this chapter due to the lack of availability of source code.

Cache-aware implementations of multigrid have also been developed. In [131], [126], and [90] optimizations improve cache utilization by reducing capacity and conflict misses during linear relaxation and inter-grid transfers. An autotuner was presented in [47] to automatically search the space of cache and memory optimizations for the relaxation step over a variety of hardware architectures. The optimizations presented in these related works are for the most part orthogonal to the approach taken in this thesis. There is no reason lower-level optimizations cannot be combined with algorithmic tuning at the level of cycle shape.

10.4 Autotuning Techniques

Layered learning, [136], used for robot soccer, is broadly related to our work. Like INCREA, layered learning is used when a mapping directly from inputs to outputs is not tractable and when a task can be decomposed to be solved bottom up. In layered learning however, composition occurs through learning, in the general sense of abstracting and solving the local concept-learning

task. (See [75] where genetic programming is used for learning.) INCREA combines optimizations, in contrast. Both approaches use domain specific knowledge to determine appropriate learning granularity: input size doubling in INCREA) and subtask definition in layered learning. Layered learning occurs separately on each hand designed level with a hand designed interface between each level. In contrast, INCREA incorporates the entire composition into one algorithm which automatically lengthens the genome only as needed.

Using an adaptive sampling strategy for fitness estimation dates back to [3]. A combination of approaches from [36, 138] inform INCREA's strategy. In [36] a t-test is evaluated and found to be effective when an appropriate population size is not known. In [138] individuals are further evaluated only if there is some chance that the outcome of the tournaments they participate in can change. The GPEA may derive some of its robustness to noise from its use of a relatively large population. See [18, 33] for discussions of this robustness.

There is one evolutionary algorithm, named Differential Evolution (DE) [115], 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 [51, 168] 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 [51]. 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.

10.5 Online Autotuning

In the context of methods in evolutionary algorithms that provide parameter adjustment or configuration, the taxonomy of Eiben [59] distinguishes between offline "parameter tuning" and online "parameter control". Operator selection is similar to parameter control because it is online.

However, it differs from parameter control because the means of choosing among a set of operators contrasts to refining a scalar parameter value.

Adaptive methods, in contrast to self-adaptive methods, explicitly use isolated feedback about past performance of an operator to guide how a parameter is updated. An adaptive operator strategy has two components: operator credit assignment and an operator selection rule. The *credit assignment* component assigns a weight to an operator based on its past performance. An operator's performance is generally measured in terms related to the objective quality of the candidate solutions it has generated. The *operator selection rule* is a procedure for choosing one operator among the eligible set based upon the weight of each. There are three popular adaptive methods: probability matching, adaptive pursuit and multi-armed bandit. Fialho has authored (in collaboration with assorted others) a large body of work on adaptive operation selection, see, for example, [61, 62]. The strategy we implement is multi-armed bandit with AUC credit assignment. This strategy is comparison-based and hence invariant to the scale of the fitness function which can vary significantly between PetaBricks programs. The invariance is important to the feasibility of hyperparameter selection on a general, rather than a per-program, basis.

There is one evolutionary algorithm, differential evolution [116], that takes a comparison-based approach to search like our autotuner. However, differential evolution compares a parent to its offspring, while our algorithm is not always competing parent and offspring. The current best solution is one contestant in the competition and its competitor is not necessarily its offspring. Differential evolution also uses a method different from applying program-dependent mutation operators to generate its offspring.

10.6 Autotuning Heterogeneous Architectures

The use of autotuning techniques is even more commonplace when optimizing GPGPU programs. Autotuning is typically applied in a program or domain-specific fashion. Such systems use autotuners to construct poly-algorithms for solving large tridiagonal systems [48], for tuning GPU sorting algorithms [72], autotuning 3D FFT with a focus on padding and bandwidth optimizations [105], autotuning sparse matrix-vector multiply by building performance models [41],

and to tune dense linear algebra with a mix of model-driven and empirical techniques [146]. These techniques are often specific to a problem or class of problems.

Besides problem-specific techniques, there is a high-level directive-based GPU programming that uses HMPP workbench to automatically generate CUDA/OpenCL code, and auto-tunes on the optimization space on the generated GPU kernels [73]. However, this technique and the previous ones only point towards autotuning as a necessity in order to get the best performance on modern GPUs; they do not address how to utilize all available resources together to achieve the best performance on a heterogeneous system.

Several methods to efficiently distribute workload between different devices have been studied. StarPU applies work-stealing framework to balance work among subsystems [21]. However, StarPU requires the programmer to write separate CPU and GPU code, and relies entirely on dynamic work-stealing guided by automatic hints to distribute tasks. CnC-HC automatically generates CPU, GPU, and FPGA code and uses a work-stealing scheduler to distribute work among different devices guided by manual hints only [128]. Qilin, automatically generates code and uses *adaptive mapping* for performance tuning [98]. During the training run, Qilin executes the program on different input sizes on CPUs and GPUs separately, and uses the result to determine workload partitions between the CPUs and the GPUs. However, the mapping of these systems may not be ideal. Our system automates the entire process, both translating kernels to different targets, and empirically determining where they should run in our hybrid work-stealing/work-pushing runtime. For real applications, the ideal mapping is globally non-linearly inter-dependent with all other choices, and our global optimization captures this during autotuning. Our results demonstrate the importance of global learning.

CGCM [82] uses a technique for automatic management of GPU/CPU memory communication. This technique is similar to our analysis for determining when lazy or eager copy-outs are used. Their technique uses a conservative reachability analysis to determine where to insert calls into a runtime that dynamically tracks and maps data to different memories. They focus on a number of optimizations to this technique to reduce runtime overhead. While CGCM manages data movement automatically, it requires some programmer help when managing parallelism.

A number of other programming languages attempt to make programming for GPGPU devices easier. CUDA-lite [166], automates some of the parallelization and memory management tasks in writing CUDA. JCUDA [160] alleviates many of the difficulties in using the GPU from Java code. There have also been efforts to map subsets of C to the GPU [25, 94]. These techniques put affine data access restrictions on the program. There have been other efforts to map OpenMP to the GPU [92]. While these efforts make running code on the GPU easier, they will produce the same code for each heterogeneous system and do not allow algorithmic choice, or empirically infer the best mapping for each machine.

Researchers have also studied the relative throughput of the CPU and the GPU and discovered contradictory findings with some studies showing 100x performance differences and others just 2.5x [93]. Our work sheds more light on this debate, showing that both claims can be correct. The best device to use is highly dependant both on the architecture and algorithmic choices and cannot be determined by simple relative performance numbers. We also show cases where the best throughput is obtained by using both the CPU and the GPU in parallel.

Chapter 11

Conclusions

The overriding goal of this thesis was to automate the process of optimizing computer programs to create programs that can adapt to work optimally in different environments and to conform to different requirements. We presented the PetaBricks programming language which focuses on ways for expressing program implementation search spaces at the language level and OpenTuner which provides sophisticated techniques for searching these spaces in a way that can easily be adopted by other projects.

PetaBricks, introduced in Chapter 2 and discussed throughout this thesis, is the first language that allows programmers to naturally express algorithmic choice explicitly so as to empower the compiler to perform deeper optimization. We have created a compiler and an autotuner that is not only able to compose a complex program using fine-grained algorithmic choices but also find the right choice for many other parameters. In Chapters 4 and 5 we showed the efficacy of this system by developing a non-trivial suite of benchmark applications. Many of these benchmarks also exposes the accuracy of different choices to the compiler. Our results show that the autotuned hybrid programs are always better than any of the individual algorithms.

PetaBricks introduces a new programming model where trade-offs 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. Using these benchmarks, we have provided evidence of the importance of exposing accuracy and algorithmic choices to the compiler when autotuning variable accuracy programs.

As a notable demonstration of this programming model, in Chapter 5 we introduced a novel dynamic programming approach to autotuning multigrid algorithms. Our approach tunes with an awareness of accuracy that allows fair comparison between various direct, iterative, and recursive algorithmic types such that optimal solutions are built from the bottom up. We demonstrated that the resulting tuned cycles achieve excellent performance compared to algorithmically static implementations of multigrid.

The PetaBricks autotuner, which we discussed in Chapter 6, is an evolutionary algorithm that is efficiently designed for problems which are suited to incremental shortcuts and require them because of they have large search spaces and expensive solution evaluation. It also efficiently handles problems which have noisy candidate solution quality. In the so called "real world", problems of this sort abound. This improves over a general purpose evolutionary algorithm that ignores the incremental structure that exists in these problems and, while it may identify a solution, it wastes computation, takes too long and produces error prone results. The PetaBricks autotuner solves smaller to larger problem instances as generations progress and it expands and shrinks its genome and population each generation. For further efficiency, it cuts off work that doesn't appear to promise a fruitful result. It addresses noisy solution quality efficiently by focusing on resolving it for small solutions which have a much lower cost of evaluation.

Chapter 7 shows a two level solution to the problem of input sensitivity in autotuning that, first, clusters to find input sets that are similar in the multi-dimensional property space and uses an evolutionary autotuner to build an optimized program each of these clusters, and then builds an adaptive overhead aware classifier which assigns each input to a specific input optimized program. This provides a general means of automatically determining what algorithmic optimization to use when different optimization strategies suit different inputs. Though this work, we are able to extract principles for understanding the performance and accuracy space of a program across a variety of inputs, and achieve speedups of up to 3x. While at first input sensitivity seems to be excessively

complicated issue where one must deal with large optimization spaces and complex input spaces, we show that input sensitivity can be handled with simple extensions to an existing autotuning system. We also showed that there are fundamental diminishing returns as more and more input adaptation is added to a system and that a little bit of input adaption can go a long way.

Our online autotuner, SiblingRivalry (Chapter 8), 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. 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. 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.

Finally, Chapter 9 presents OpenTuner, a new framework for building domain-specific multi-objective program autotuners. OpenTuner supports fully customizable configuration representations and an extensible technique representation to allow for domain-specific techniques. OpenTuner introduces the concept of ensembles of search techniques in autotuning, which allow many search techniques to work together to find an optimal solution and provides a more robust search than a single technique alone. OpenTuner is free and open source [153] and as the community adds more techniques and representations to this flexible framework, there will be less of a need to create a new representation or techniques for each project and we hope that the system will work out-of-the-box for most creators of autotuners. OpenTuner pushes the state of the art forward in program autotuning in a way that can easily be adopted by other projects. We hope that OpenTuner will be an enabling technology that will allow the expanded use of program autotuning both to more domains and by expanding the role of program autotuning in existing domains.

We have presented a variety of results in this thesis. In Chapter 4, we saw that PetaBricks can provide significant speedups over using a single algorithm or a hard coded heuristic; that by trading accuracy for performance one can meet changing quality of service targets without wasted

resources; and that vastly different configurations are needed on different heterogenous machines and processor/coprogressor types. Chapter 5 showed a novel dynamic programming technique for creating multigrid V-cycle shapes tailored to a specific problem and execution environment. Chapter 6 showed how our botton-up evolutionary algorithm outperforms more traditional evolution autotuners. Chapter 7 showed that many programs are input sensitive, and demonstrated speedups by automatically adapting algorithms to best fit program inputs. Chapter 8 demonstrated our online autotuner and showed speedups when having programs adapt to load on a system. Finally, Chapter 9 introduced OpenTuner and showed how sophisticated autotuning techniques can be used to provide speedups for other projects.

In this thesis we showed the importance of autotuning and that there is no one size fits all solution to program optimization. One needs to use different techniques and optimizations to get performance in different situations. We identified many important problems in autotuning and provided viable solutions to each. We believe autotuning is ripe for general adoption and may soon be ubiquitous in software development and deployment toolchains. We envision a future where programs are more dynamic and automatically adapt to fit their environment, inputs, available resources, and changing requirements.

Perhaps send me that paragraph alone before

Bibliography

[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 *Symposium on Code Generation and Optimization*, 2006.

[2] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, John Thomson, Mark Toussaint, and Christopher K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO'06, pages 295 305, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Akiko N. Aizawa and Benjamin W. Wah. Scheduling of genetic algorithms in a noisy environment. *Evolutionary Computation*, 2(2):97–122, 1994.

[4] 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.

[5] 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 *Conference on Languages, Compilers, and Tools for Embedded Systems*, New York, NY, USA, 2004.

[6] Ed Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, and

Danny Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[7] Jesper Andersson, Morgan Ericsson, Christoph Kessler, and Welf Lowe. Profile-guided composition. *Lecture Notes in Computer Science*, 4954:157–164, March 2008.

[8] Jason Ansel. Petabricks: A language and compiler for algorithmic choice. S.m. thesis, Massachusetts Institute of Technology, Cambridge, MA, Sep 2009.

[9] Jason Ansel, Yee Lok Won ans Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. Technical Report MIT/CSAIL Technical Report MIT-CSAIL-TR-2010-032, Massachusetts Institute of Technology, Cambridge, MA, Jul 2010.

[10] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[11] Jason Ansel and Cy Chan. Petabricks: Building adaptable and more efficient programs for the multi-core era. *Crossroads, The ACM Magazine for Students (XRDS)*, 17(1):32–37, Sep 2010.

[12] 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.

[13] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Manuscript submitted for review*, July 2013.

[14] Jason Ansel, Petr Marchenko, Ulfar Erlingsson, Elijah Taylor, Brad Chen, Derek Schuff, David Sehr, Cliff Biffle, , and Bennet Yee. Language-independent sandboxing of just-in-

time compilation and self-modifying code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Jose, CA, Jun 2011.

[15] Jason Ansel, Maciej Pacula, Saman Amarasinghe, and Una-May O'Reilly. An efficient evolutionary algorithm for solving bottom up problems. In *Annual Conference on Genetic and Evolutionary Computation*, Dublin, Ireland, July 2011.

[16] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: Online autotuning through local competitions. In *International Conference on Compilers Architecture and Synthesis for Embedded Systems*, Tampere, Finland, Oct 2012.

[17] 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 *International Symposium on Code Generation and Optimization*, Chamonix, France, Apr 2011.

[18] Dirk V. Arnold and Hans-Georg Beyer. On the benefits of populations for noisy optimization. *Evolutionary Computation*, 11(2):111–127, 2003.

[19] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *ACM-SIAM Symposium on Discrete Algorithms*, January 2007.

[20] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, 2003.

[21] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2), 2011.

[22] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *PLDI*, pages 149–159, 1996.

[23] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, New York NY, 1996.

[24] Woongki Baek and Trishul Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2010.

[25] Muthu Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-CUDA code generation for affine programs. In Rajiv Gupta, editor, *Compiler Construction*, volume 6011. Springer Berlin / Heidelberg, 2010.

[26] 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.

[27] V. Bhat, M. Parashar, . Hua Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed. Enabling self-managing applications using model-based online control strategies. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pages 15–24, Washington, DC, USA, 2006.

[28] Sanjukta Bhowmick, Padma Raghavan, and Keita Teranishi. A combinatorial scheme for developing efficient composite solvers. In *Proceedings of the International Conference on Computational Science-Part II*, pages 325–334, London, UK, 2002.

[29] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the ACM International Conference on Supercomputing*, pages 340–347, 1997.

[30] 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.

[31] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 3–16, New York, NY, USA, 1991. ACM.

[32] Andrew P. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997.

[33] Jurgen Branke. Creating robust solutions by means of evolutionary algorithms. In Agoston Eiben, Thomas Baeck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature, PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 119–. Springer Berlin / Heidelberg, 1998.

[34] Jurgen Branke, Christian Schmidt, and Hartmut Schmec. Efficient fitness estimation in noisy environments. In *Proceedings of Genetic and Evolutionary Computation*, pages 243–250, 2001.

[35] 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.

[36] Erick Cantu-Paz. Adaptive sampling for noisy problems. In *Genetic and Evolutionary Computation, GECCO 2004*, volume 3102 of *Lecture Notes in Computer Science*, pages 947–958. Springer Berlin / Heidelberg, 2004.

[37] Cy Chan, Jason Ansel, Yee Lok Wong, Saman Amarasinghe, and Alan Edelman. Autotuning multigrid with petabricks. In *ACM/IEEE Conference on Supercomputing*, Portland, OR, Nov 2009.

[38] Fangzhe Chang and Vijay Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing*, 4:49–62, March 2001.

[39] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):21:1–21:30, October 2012.

[40] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI'10, pages 448–459, New York, NY, USA, 2010. ACM.

[41] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2010.

[42] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS*, pages 676–687. IEEE, 2011.

[43] Gene Cooperman, Jason Ansel, and Xiaoqin Ma. Adaptive checkpointing for master-worker style parallelism. In *IEEE Computer Society International Conference on Cluster Computing*, Boston, MA, Sep 2005.

[44] Gene Cooperman, Jason Ansel, and Xiaoqin Ma. Transparent adaptive library-based checkpointing for master-worker style parallelism. In *IEEE International Symposium on Cluster Computing and the Grid*, Singapore, May 2006.

[45] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[46] Luis DaCosta, Alvaro Fialho, Marc Schoenauer, and Michèle Sebag. Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, GECCO '08, pages 913–920, New York, NY, USA, 2008.

[47] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization

and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[48] Andrew Davidson, Yao Zhang, and John D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *Parallel and Distributed Processing Symposium*. IEEE, May 2011.

[49] Lawrence Davis. Adapting operator probabilities in genetic algorithms. In J. David Schaffer, editor, *ICGA*, pages 61–69, 1989.

[50] 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.

[51] 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, *Parallel Problem Solving from Nature – PPSN VI*, pages 849–858, Berlin, 2000. Springer.

[52] James W. Demmel. *Applied Numerical Linear Algebra*. August 1997.

[53] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *Manuscript submitted for review*, July 2013.

[54] Pedro Diniz and Martin Rinard. Eliminating synchronization overhead in automatically parallelized programs using dynamic feedback. *ACM Trans. Comput. Syst.*, 17:89–132, May 1999.

[55] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: an effective technique for adaptive computing. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 71–84, New York, NY, USA, 1997.

[56] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, Mara Jesus Garzaran, David Padua, Keshav Pingali, Bull Sa, and Inria Futurs. A language for the compact representation of multiple program versions. In *In Languages and Compilers for Parallel Computers (LCPC'05)*, page 15, 2005.

[57] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[58] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F. P. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 78–88, New York, NY, USA, 2009. ACM.

[59] A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 3(2):124 –141, July 1999.

[60] Xiang Fan. Optimize your code: Matrix multiplication. `https://tinyurl.com/kuvzbp9`, 2009.

[61] Álvaro Fialho. *Adaptive Operator Selection for Optimization.* PhD thesis, Université Paris-Sud XI, Orsay, France, December 2010.

[62] Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michele Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence – Special Issue on Learning and Intelligent Optimization*, 2010.

[63] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC)*, volume 5658 of *Lecture Notes in Computer Science*, pages 385–410. Springer, 2009.

[64] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[65] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384, 1998.

[66] 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".

[67] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, Jun 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[68] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher Williams, and Michael F. P. OBoyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39:296–327, 2011. 10.1007/s10766-010-0161-2.

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

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

[71] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mirceà 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 *GCC Developers' Summit*, Jul 2008.

[72] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. In *ACM SIGMOD*, 2006.

[73] Scott Grauer-Gray, Lifan Xu, Robert Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing Conference*. IEEE, May 2012.

[74] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[75] Steven M. Gustafson and William H. Hsu. Layered learning in genetic programming for a co-operative robot soccer problem. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 291–301, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[76] 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 *Proceeding of the 7th international conference on Autonomic computing*, ICAC '10, pages 79–88, New York, NY, USA, 2010.

[77] 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.

[78] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Power-aware computing with dynamic knobs. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011*, 2011.

[79] 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.

[80] 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, 2001.

[81] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[82] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In *Programming language design and implementation*, New York, NY, USA, 2011.

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

[84] Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 10:1–10:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[85] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. Brainy: effective selection of data structures. In *Proceedings of the 32nd ACM SIGPLAN conference*

on Programming language design and implementation, PLDI '11, pages 86–97, New York, NY, USA, 2011. ACM.

[86] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[87] Shoaib Ashraf Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages.* PhD thesis, EECS Department, University of California, Berkeley, Jan 2013.

[88] Gabor Karsai, Akos Ledeczi, Janos Sztipanovits, Gabor Peceli, Gyula Simon, and Tamas Kovacshazy. An approach to self-adaptive software based on supervisory control. In *In 2nd International Workshop in Self-adaptive software, (IWSAS-01), Robert Laddaga, Howard Shrobe, and Paul.* 2001.

[89] Christoph W. Kessler and Welf Lowe. A framework for performance-aware composition of explicitly parallel components. In *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 227–234, 2007.

[90] Markus Kowarschik and Christian Wei. Dimepack – a cache-optimized multigrid library. In *The International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 425–430. CSREA, CSREA Press, 2001.

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

[92] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44, February 2009.

[93] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund,

Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *international symposium on Computer architecture*, New York, NY, USA, 2010.

[94] Allen Leung, Nicolas Vasilache, Benoit Meister, Muthu Baskaran, David Wohlford, Cedric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Workshop on General-Purpose Computation on Graphics Processing Units*, New York, NY, USA, 2010.

[95] Xiaoming Li, Maria Jesus Garzaran, and David Padua. A dynamically tuned sorting library. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–122, March 2004.

[96] Xiaoming Li, Mara Jess Garzarn, and David Padua. Optimizing sorting with genetic algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 99–110, 2005.

[97] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu programs optimization. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, pages 1–10, 2009.

[98] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *International Symposium on Microarchitecture*, New York, NY, USA, 2009.

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

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

[101] Andrew Moss and Dan Page. Program interpolation. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 31–40, New York, NY, USA, 2009.

[102] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. Practical aggregation of semantical program properties for machine learning based optimization. In *Proceedings of the 2010 international conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES'10, pages 197–206, New York, NY, USA, 2010. ACM.

[103] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[104] Kristoffer Nordkvist. Solving TSP with a genetic algorithm in C++. `https://tinyurl.com/lq3uqlh`, 2012.

[105] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-d FFT library for CUDA GPUs. In *High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, 2009.

[106] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *The International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar 2009.

[107] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Scaling deterministic multithreading. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, Newport Beach, CA, Mar 2011.

[108] Marek Olszewski and Michael Voss. Install-time system for automatic generation of optimized parallel sorting algorithms. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 17–23, 2004.

[109] Marek Olszewski, Qin Zhao, David Koh, Jason Ansel, and Saman Amarasinghe. Aikido: Accelerating shared data dynamic analyses. In *The International Conference on Architectural Support for Programming Languages and Operating Systems*, London, UK, March 2012.

[110] Maciej Pacula, Jason Ansel, Saman Amarasinghe, and Una-May O'Reilly. Hyperparameter tuning in bandit-based adaptive operator selection. In *European Conference on the Applications of Evolutionary Computation*, Malaga, Spain, Apr 2012.

[111] Justin Mazzola Paluska, Hubert Pham, Umar Saif, Grace Chau, Chris Terman, and Steve Ward. Structured decomposition of adaptive applications. In *Proceedings of the Annual IEEE International Conference on Pervasive Computing and Communications*, pages 1–10, Washington, DC, USA, 2008.

[112] Eunjung Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *Symposium on Code Generation and Optimization*, April 2011.

[113] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. In *The International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, TX, March 2013.

[114] Scratch Pixel. 3D Basic Lessons: Writing a simple raytracer. `https://tinyurl.com/lp8ncnw`, 2012.

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

[116] Kenneth Price, Rainer M. Storn, and Jouni A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[117] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[118] 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):21–45, 2004.

[119] J.R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[120] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[121] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[122] Richard H. Rand. *Computer algebra in applied mathematics: an introduction to MACSYMA*. Number 94 in Research notes in mathematics. London, UK, 1984.

[123] Michael Rieker, Jason Ansel, and Gene Cooperman. Transparent user-level checkpointing for the native posix thread library for linux. In *The International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, Jun 2006.

[124] 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.

[125] Martin Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 369–386, 2007.

[126] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3d scientific computations. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 32, Washington, DC, USA, 2000. IEEE Computer Society.

[127] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. Adaptive input-aware compilation for graphics engines. In *Proceedings of ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation*, 2012.

[128] Alina Sbîrlea, Yi Zou, Zoran Budimlíc, Jason Cong, and Vivek Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. In *International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, New York, NY, USA, 2012.

[129] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. Atune-IL: An instrumentation language for auto-tuning parallel applications. In *Euro-Par Conference*, August 2009.

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

[131] Sriram Sellappa and Siddhartha Chatterjee. Cache-efficient multigrid algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):115–133, 2004.

[132] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 167–178, New York, NY, USA, 2007. ACM.

[133] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 136–148, New York, NY, USA, 2008. ACM.

[134] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 281–294, New York, NY, USA, 2005. ACM.

[135] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.

[136] Peter Stone and Manuela Veloso. Layered learning. In Ramon Lopez de Montaras and Enric Plaza, editors, *Machine Learning: ECML 2000*, volume 1810 of *Lecture Notes in Computer Science*, pages 369–381. Springer Berlin / Heidelberg, 2000.

[137] 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.

[138] Astro Teller and David Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[139] 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*, pages 77–90. 2007.

[140] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–288, 2005.

[141] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–288, New York, NY, USA, 2005.

[142] John Thomson, Michael O'Boyle, Grigori Fursin, and Björn Franke. Reducing training time in a one-shot machine learning-based compiler. In *Proceedings of the 22nd international*

conference on Languages and Compilers for Parallel Computing, LCPC'09, pages 399–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[143] K. Tian, Y. Jiang, E. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[144] Ananta Tiwari, Chun Chen, Cha Jacqueline, Mary Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[145] Top500. Top 500 supercomputer sites. http://www.top500.org/, 2010.

[146] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing*, November 2008.

[147] Yevgen Voronenko, Frédéric de Mesmay, and Markus Püschel. Computer generation of general size linear transform libraries. In *International Symposium on Code Generation and Optimization (CGO)*, pages 102–113, 2009.

[148] Michael Voss and Rudolf Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *Proceedings of the International Conference on Parallel Processing*, pages 163–170, 2000.

[149] Michael Voss and Rudolf Eigenmann. High-level adaptive program optimization with adapt. *ACM SIGPLAN Notices*, 36(7):93–102, 2001.

[150] Richard Vuduc, James W. Demmel, and Jeff A. Bilmes. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, 2004.

[151] 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.

[152] P. Waldemar and T. Ramstad. Hybrid KLT-SVD image compression. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, page 2713, Washington, DC, USA, 1997.

[153] OpenTuner Website. `https://opentuner.org/`, 2013.

[154] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[155] Richard Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *ACM/IEEE Conference on Supercomputing*, pages 1–27, Washington, DC, USA, 1998.

[156] Richard Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[157] 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.

[158] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *proceedings of 22nd Annual International Symposium on Computer Architecture News*, pages 24–36, June 1995.

[159] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. SPL: a language and compiler for DSP algorithms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–308, New York, NY, USA, 2001.

[160] Yonghong Yan, Max Grossman, and Vivek Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704. Springer Berlin / Heidelberg, 2009.

[161] Qing Yi and Richard Clint Whaley. Automated transformation for performance-critical kernels. In *Proceedings of the ACM SIGPLAN Symposium on Library-Centric Software Design*, Oct. 2007.

[162] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–76, New York, NY, USA, 2003.

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

[164] 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.

[165] Lotfi A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Commun. ACM*, 37(3):77 84, 1994.

[166] Sain zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen mei W. Hwu. CUDA-Lite: Reducing GPU programming complexity. In *Workshops on Languages and Compilers for Parallel Computing*. Springer, 2008.

[167] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Symposium on Principles and Practice of Parallel Programming*, January 2010.

[168] 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*. CIMNE, Barcelona, Spain, 2002.