PetaBricks: A Language and Compiler for 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
Master of Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 2009

Author..
Department of Electrical Engineering and Computer Science
September 1, 2009

Certified by..
Saman Amarasinghe
Professor
Thesis Supervisor

Accepted by..
Terry P. Orlando
Chair, Department Committee on Graduate Students

PetaBricks: A Language and Compiler for Algorithmic Choice

by
Jason Ansel

Submitted to the Department of Electrical Engineering and Computer Science on
August 25, 2009, in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

It is often impossible to obtain a one-size-fits-all solution for high performance algorithms when considering different choices for data distributions, parallelism, transformations, and blocking. The best solution to these choices is often tightly coupled to different architectures, problem sizes, data, and available system resources. In some cases, completely different algorithms may provide the best performance. Current compiler and programming language techniques are able to change some of these parameters, but today there is no simple way for the programmer to express or the compiler to choose different algorithms to handle different parts of the data. Existing solutions normally can handle only coarse-grained, library level selections or hand coded cutoffs between base cases and recursive cases.

We present PetaBricks, a new implicitly parallel language and compiler where having multiple implementations of multiple algorithms to solve a problem is the natural way of programming. We make algorithmic choice a first class construct of the language. 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.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

Acknowledgments

Many other people have made major contributions to the work presented in this thesis. I am grateful for the advice and guidance of my advisor Professor Saman Amarasinghe. I would also like to acknowledge the contributions of our collaborators in applied math, Cy Chan, Yee Lok Wong, and Alan Edelman. They were responsible for designing and implementing the majority of our benchmarks. Additionally, their feedback and ideas were greatly influential in shaping the PetaBricks language. I also would like to thank Marek Olszewski for implementing our work stealing dynamic scheduler. These people also contributed substantially to the writing of [4] and [7] on which the text of this thesis is based.

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 RQE committee, Armando Solar-Lezama and Martin Rinard whose feedback helped shape the talk I gave on this work. I finally would like to thank my family for their love and support.

This work is partially supported by NSF Award CCF-0832997 and an award from the Gigascale Systems Research Center.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

While 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 both algorithms, the glue code to connect them together, and figure out the best switch over points. Today's compilers are unable to change the nature of this composition because it is constructed with traditional control logic such as loops and switches. In this work, we propose new language constructs that allow the programmer to specify a menu of algorithmic choices and new compiler techniques to exploit these choices to generate high performance yet portable code.

Hand-coded algorithmic compositions are commonplace. A typical example of such a composition can be found in the C++ Standard Template Library (STL)[1] routine `std::sort`, which uses merge sort until the list is smaller than 15 elements and then switches to insertion sort. 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 problem has been addressed for certain specific algorithms by autotuning software, such as ATLAS [33] and FFTW [11, 12], which have training phases where optimal algorithms and

[1] From the version of the libstdc++ included with GCC 4.3.

cutoffs are automatically selected. Unfortunately, systems like this only work on the few algorithms provided by the library designer. In these systems, algorithmic choice is made by the application without the help of the compiler.

In this work, we present PetaBricks, a new implicitly parallel programming language for high performance computing. 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 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.

We have also developed a benchmark suite of PetaBricks programs. These benchmarks demonstrate the importance of making algorithmic choices available to the compiler. In all cases, hybrid algorithms, consisting of a non-trivial composition of user-provided algorithms, perform significantly better than any one algorithm alone.

In one of our benchmark programs, a multigrid solver for the Poisson equation, we demonstrate how to incorporate algorithms with variable convergence criteria in the autotuning process. This capability is vital when composing direct (exact) and iterative (approximate) methods in a recursive structure in such a way that guarantees a specified target accuracy for the output while ensuring near-optimal efficiency.

1.1 Motivating Example

As a motivation example, consider the problem of sorting. There are a huge number of ways to sort a list. 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 might get 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).

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. 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.

1.2 Outline

Section 2 describes the PetaBricks language. Section 3 describes the implementation of the compiler and autotuning system. Section 4 describes our benchmark suite. Section 5 presents experimental results. Section 6 covers related work. Finally, Sections 7 and 8 describe future work and conclusions.

1.3 Contributions

We make the following contributions:

- We present 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.

- While autotuners have exploited coarse-grained algorithmic choice at a programmatic level, to best of our knowledge this is the first compiler that incorporates fine-grained algorithmic choices in program optimization.

- We show how our compiler utilizes fine-grained algorithmic choice to get significant speedup over conventional algorithms.

- We show that PetaBricks programs adapt algorithmically to different architectures to create truly portable programs. We demonstrate that a PetaBricks program autotuned locally on an 8-way x86_64 performs 2.35x faster when compared to a configuration trained on a 8-way Sun Niagara 1 processor.

- We show that PetaBricks programs are scalable because they can adapt to expose increasing parallelism as the number of cores increases. We demonstrate that a configuration autotuned on 8 cores performs 2.14x faster than a configuration tuned on a single core, but executed on 8 cores.

- We present a compiler that can autotune programs with complex trade-offs such that we ensure the best performance for all required levels of accuracy.

Chapter 2

PetaBricks Language

In designing the language we had the following major goals:

- Expose algorithmic choices to the compiler

- Allow choices to specify different granularities and corner cases

- Expose all valid execution orders, to allow parallel execution

- Automate consistency checks between different choices

- Provide flexible data structures, including n-dimensional arrays, trees, and sparse representations

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*, *from*, and *through* arguments, which represent inputs, outputs, and 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 runtime.

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.

PetaBricks does not contain an outer sequential control flow. The user specifies 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 2.1 shows an example PetaBricks transform, that performs a matrix multiplication. The transform header is on lines 1 to 3. The first rule (line 6 to 9) 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. Rules two, three, and four (line 12 to 39) represent three ways of recursively decomposing matrix multiply into smaller matrix multiplies. The compiler must pick when to apply these recursive decompositions. The last two rules are actually not needed because they are automatically inferred by the compiler as it explores blocking strategies for iteration. The autotuner discovers that the last two rules provide no advantage over the compiler's intrinsic strategies and correctly chooses not to use them.

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 parameters to the autotuner. PetaBricks analyzes where these tunable values are used, and autotunes them at an appropriate time in the learning process.

PetaBricks contains the following additional language features that will not be discussed here in detail:

- `%{ ... }%` escapes used to embed raw C++ in the output file. This is primarily used for calling external libraries. External libraries must be thread safe.

- 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. This construct is syntactic sugar for adding an extra dimension to the matrix, which may then be collapsed by analysis.

- Rule priorities and `where` clauses are used to handle corner cases gracefully.

- Template transforms, similar to templates in C++, where each template instance is autotuned separately.

```
transform MatrixMultiply
from A[c,h], B[w,c]
to AB[w,h]
{
  // Base case, compute a single element
  to(AB.cell(x,y) out)
  from(A.row(y) a, B.column(x) b) {
    out = dot(a,b);
  }

  // Recursively decompose in c
  to(AB ab)
  from(A.region(0,   0, c/2, h  ) a1,
       A.region(c/2, 0, c,   h  ) a2,
       B.region(0,   0, w,   c/2) b1,
       B.region(0, c/2, w,   c  ) b2) {
    ab = MatrixAdd(MatrixMultiply(a1, b1),
                   MatrixMultiply(a2, b2));
  }

  // Recursively decompose in w
  to(AB.region(0,   0, w/2, h  ) ab1,
     AB.region(w/2, 0, w,   h  ) ab2)
  from( A a,
        B.region(0,   0, w/2, c  ) b1,
        B.region(w/2, 0, w,   c  ) b2) {
    ab1 = MatrixMultiply(a, b1);
    ab2 = MatrixMultiply(a, b2);
  }

  // Recursively decompose in h
  to(AB.region(0, 0,   w, h/2) ab1,
     AB.region(0, h/2, w, h  ) ab2)
  from(A.region(0,   0, c,   h/2) a1,
       A.region(0, h/2, c,   h  ) a2,
       B b) {
    ab1=MatrixMultiply(a1, b);
    ab2=MatrixMultiply(a2, b);
  }
}
```

Figure 2.1: PetaBricks source code for MatrixMultiply

Chapter 3

Implementation

The PetaBricks implementation consists of three 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.

3.1 PetaBricks Compiler

To help illustrate the compilation process we will use the example transform `RollingSum`, shown in Figure 3.2. 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

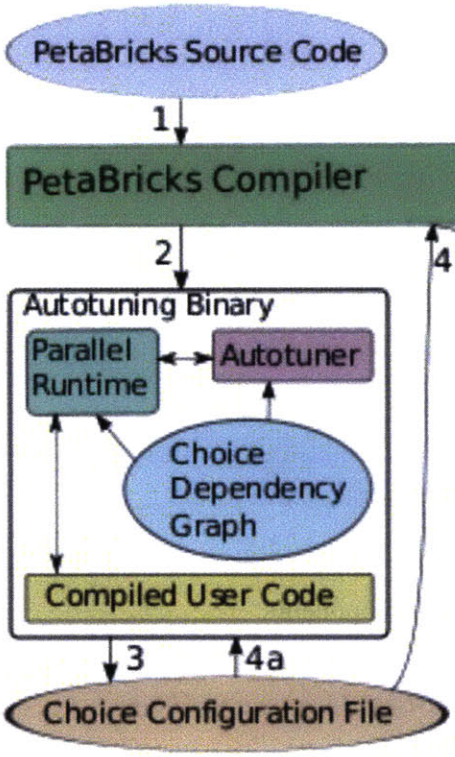


Figure 3.1: Interactions between the compiler and output binaries. First, in Steps 1 and 2, the compiler reads the source code and generates an autotuning binary. Next, in Step 3, autotuning is run to generate a choice configuration file. Finally, either the autotuning binary is used with the configuration 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).

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. The intermediate representation is built up as the phases proceed. It starts as an abstract syntax tree and ends as a 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

```
transform RollingSum
from A[n]
to B[n]
{
  //rule0: sum all elements to the left
  to(B.cell(i) b) from(A.region(0, i) in) {
    b=sum(in);
  }

  //rule1: use the previously computed value
  to(B.cell(i) b) from(A.cell(i) a,
                       B.cell(i-1) leftSum) {
    b=a+leftSum;
  }
}
```

Figure 3.2: PetaBricks source code for RollingSum. A simple example used to demonstrate the compilation process. The output element B_x is the sum of the input elements $A_0..A_x$.

each rule a symbolic *center*, and rewriting all dependencies to be relative to this center. (This is done using the Maxima symbolic algebra library [25].) 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. These 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 matrix type. The choice grid divides each matrix into rectilinear regions where a uniform set of rules are applicable. It does this using an inference system to sort the applicable regions and divide them into smaller, simplified regions. 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. 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.

Non-rectilinear regions can also be created using where clauses on rules. In applicable regions and choice grids the bounding box for these regions is computed and used. An analysis pass of the choice grid handles these regions. For each rectilinear region in the choice grid, if some rules are restricted by where clauses, these restricted rules are replaced by meta-rules that are unrestricted. These meta-rules are constructed by finding sets of rules that cover the entire region, and packaging them up into a single meta-rule. Multiple meta-rules are added to encode any choice.

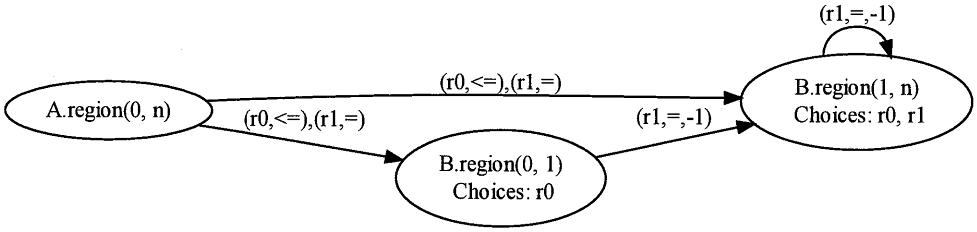


Figure 3.3: *Choice dependency graph* for RollingSum (in Figure 3.2). Arrows point the opposite direction of dependency (the direction data flows). Edges are annotated with rules and directions, offsets of 0 are not shown.

Choice dependency graph analysis. A data dependency graph is constructed using the simplified regions from the choice grid. The data dependency graph consists of edges between these symbolic regions. 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 3.3 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. 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. Cycles are eliminated by merging strongly connected components, into meta-nodes. The scheduler then finds an axis and direction for iterating this larger node where the cycle is gone, it then recursively schedules the components making up this larger node using the remaining edges.

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. These processes are explained in further detail in Sections 3.3 and 3.4.

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 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 Section 5. 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.

In addition to tuning algorithm selection, PetaBricks uses an n-ary search tuning algorithm to optimize additional parameters such as parallel-sequential cutoff points for individual algorithms, iteration orders, block sizes (for data data parallel rules), data layout, as well as user specified tunable parameters.

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 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 [13], 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` 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 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.

3.6 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.

Chapter 4

Benchmarks

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.

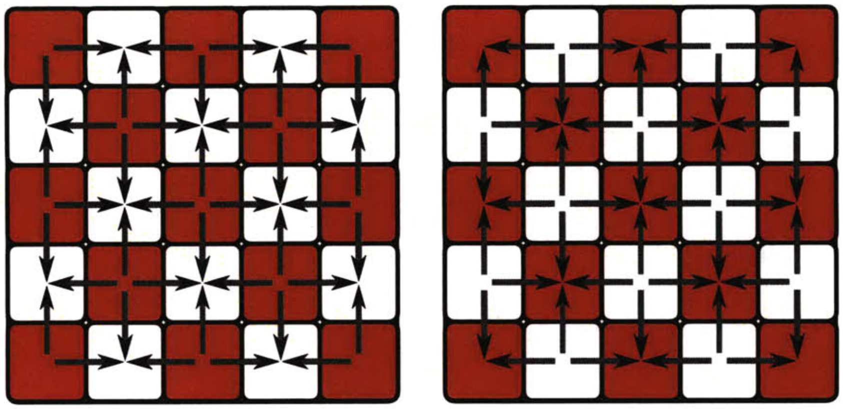
4.1 Poisson's Equation

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

$$\nabla^2 \phi = f \quad \text{and} \quad Tx = b, \tag{4.1}$$

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

To solve Poisson's equation on a 2D grid, we explore the use of four methods: one direct (band Cholesky factorization through LAPACK's DPBSV routine) and three iterative (Jacobi Iteration, Red-Black Successive Over Relaxation (SOR), and Multigrid). From top to bottom, each of the iterative methods has a larger overhead, but yields a better asymptotic serial complexity [8]. The table below lists the complexity of each algorithm, n is the number of cells in the grid.



(a) Dependencies for the red cells

(b) Dependencies for the black cells

Figure 4.1: Checkerboard dependency pattern for Red-Black SOR. Black cells are shown in white for clarity.

Algorithm	Direct	Jacobi	SOR	Multigrid
Complexity	n^2	n^2	$n^{1.5}$	n

Dependencies for SOR

There are different implementations of data dependencies for SOR, and we implement Red-Black ordering. Figure 4.1 shows the classification of cells into red and black (shown in white for clarity) depending on whether they are updated using neighboring values from the previous or current iteration. Each cell depends on its neighbors, as indicated by the arrows in the figure.

During the first half of an iteration, the red cells are updated using the black cells' values from the previous iteration. During the second half of the iteration, the black cells are updated using the red cells' values from the current iteration.

PetaBricks supports this complex dependency pattern by splitting the matrix into two temporary matrices each half the size of the input. One temporary matrix contains only red cells, the other only black cells. Each iteration of SOR then involves updating each matrix in turn. Arranging the data in such a manner leads to better cache behavior since memory is accessed in a dense fashion.

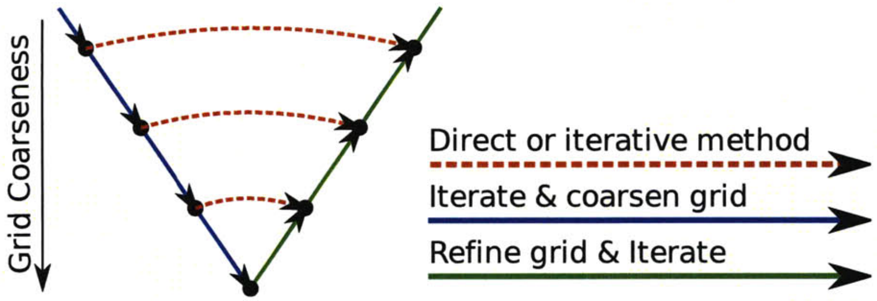


Figure 4.2: Choices in the multigrid algorithm. The diagonal arrows represent the recursive case, while the dotted horizontal arrows represent the shortcut case where a direct or iterative solution may be substituted. Depending on the desired level of accuracy a different choice may be optimal at each decision point

Multigrid

Multigrid is a recursive and iterative algorithm that uses the solution to a coarser grid resolution (by a factor of two) as part of the algorithm. 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 (4.1).

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

Figure 4.3: Pseudo code for MULTIGRID-SIMPLE.

The full multigrid algorithm requires the use of a sequence of k V-cycles of increasing refinement run in succession. In this section, we will focus on tuning a single V-cycle; the methods employed can be extended to tune a full multigrid algorithm. The pseudo code for this is shown in Figure 4.3. At the recursive call on line 6, the PetaBricks compiler can make a choice of whether to continue making recursive calls to multigrid (shown as the solid diagonal arrows) or take a shortcut by using the direct solver or one of the iterative solvers at the current resolution (shown as the dotted horizontal arrows). Figure 4.2 shows these possible paths of the multigrid algorithm.

The idea of choice can be implemented by defining a top level function POISSON, which makes calls to either the direct, iterative, or recursive solution, and having MULTIGRID call POISSON. The pseudo code for this is shown in Figure 4.4.

```
POISSON($x, b$)
1: either
2:     Solve directly
3:     Use an iterative method
4:     Call MULTIGRID for some number of iterations
5: end either
MULTIGRID($x, b$)
1: if $N = 3$ then
2:     Solve directly
3: else
4:     Iterate using some iterative method
5:     Compute the residual and restrict to half resolution
6:     On the coarser grid, call POISSON
7:     Interpolate result and add correction term to current solution
8:     Iterate using some iterative method
9: end if
```

Figure 4.4: General pseudo code for choices in POISSON and MULTIGRID.

Making the choice in line 1 of POISSON 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 parameters such as number of iterations or weights). To make a fair comparison between the choices, we must take the accuracy of each choice into account.

In the other algorithms we have examined thus far, the compiler determines which choices to make based solely on some parameters of the input (such as the input size). In autotuning our Poisson solver, we also use the desired accuracy level to make that determination. To that end, the autotuner keeps 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. In the following sections, we assume we have access to representative training data so that the accuracy of our algorithms during tuning closely reflects their accuracy during use.

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* to be the ratio between the RMS error of its input versus the RMS error of the output compared to optimal. Thus, a higher accuracy algorithm is better.

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 `MULTIGRID`. We also try varying the parameters in the other steps of the algorithm (e.g. the choice of iterative method and the number of iterations in steps 3 and 4 of `POISSON` and steps 4 and 8 of `MULTIGRID`).

Trying all of these possibilities will yield many algorithms that can be plotted as in Figure 4.5(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 for use in step 6 of `MULTIGRID` 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.

The PetaBricks Solution

The PetaBricks compiler 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 4.5(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 $\texttt{POISSON}_i$,

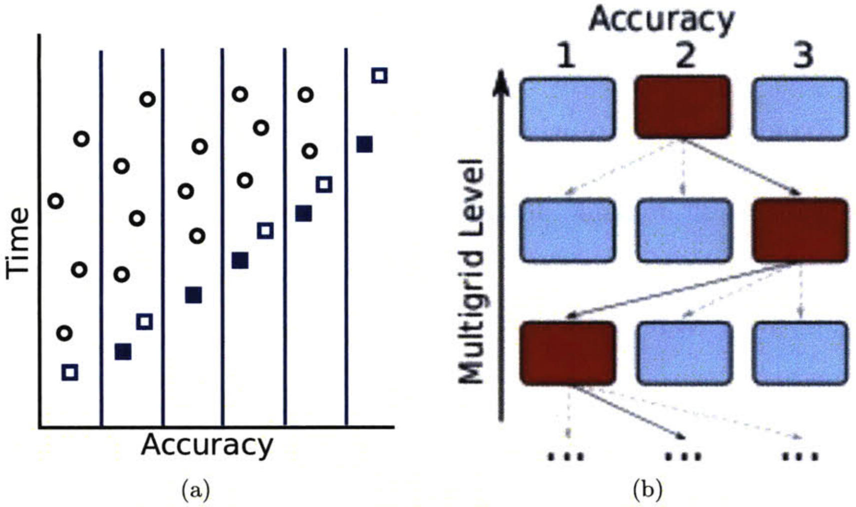


Figure 4.5: (a) Possible algorithmic choices with optimal set designated by squares (both hollow and solid). The choices designated by solid squares are the ones remembered by the PetaBricks compiler, being the fastest algorithms better than each accuracy cutoff line. (b) Choices across different accuracies 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

which achieves accuracy p_i on all input sizes.

To further narrow the search space, we only use SOR as the iteration function since it performs much better than Jacobi for similar computation cost per iteration. In $\texttt{POISSON}_i$, we fix the weight parameter of SOR to ω_{opt}, the optimal value for the 2D discrete Poisson's equation with fixed boundaries [8]. In $\texttt{MULTIGRID}_i$, 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 $\texttt{MULTIGRID}_i$ to one. The resulting accuracy-aware Poisson solver is a family of functions, where i is the accuracy parameter. This family of functions is described in the pseudo code in Figure 4.6

The autotuning process must now determine what choices to make in $\texttt{POISSON}_i$ for each i and for each size input. 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.

```
POISSON$_i(x, b)$
either
    Solve directly
    Iterate using SOR$_{\omega_{\text{opt}}}$ until accuracy $p_i$ is achieved
    For some $j$, iterate with MULTIGRID$_j$ until accuracy $p_i$ is achieved
end either
MULTIGRID$_i(x, b)$
if $N = 3$ then
    Solve directly
else
    Compute one iteration of SOR$_{1.15}$
    Compute the residual and restrict to half resolution
    On the coarser grid, call POISSON$_i$
    Interpolate result and add correction term to current solution
    Compute one iteration of SOR$_{1.15}$
end if
```

Figure 4.6: Pseudo code for family of functions POISSON$_i$ and MULTIGRID$_i$ where i is the required accuracy, as used in the benchmark.

Performance

The final set of multigrid algorithms produced by the autotuner can be visualized as in Figure 4.5(b). Each of the versions can call any of the other versions during its recursive calls to the lower level, and the optimal path may switch many times between accuracies as we recurse down towards either the base case or a shortcut case.

Figure 4.7 shows the performance of our autotuned multigrid algorithm for accuracy 10^9. The autotuned algorithm uses 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 and iterated calls to Jacobi, SOR, and MULTIGRID-SIMPLE (labeled Multigrid). Each of the iterative methods is run until an accuracy of at least 10^9 is achieved.

The autotuned algorithm shown calls the direct algorithm for small cases up to size $N = 129$, at which point it starts making recursive calls to MULTIGRID. The number of iterations computed at each level of recursion is determined by the autotuner to be optimal given the desired level of accuracy.

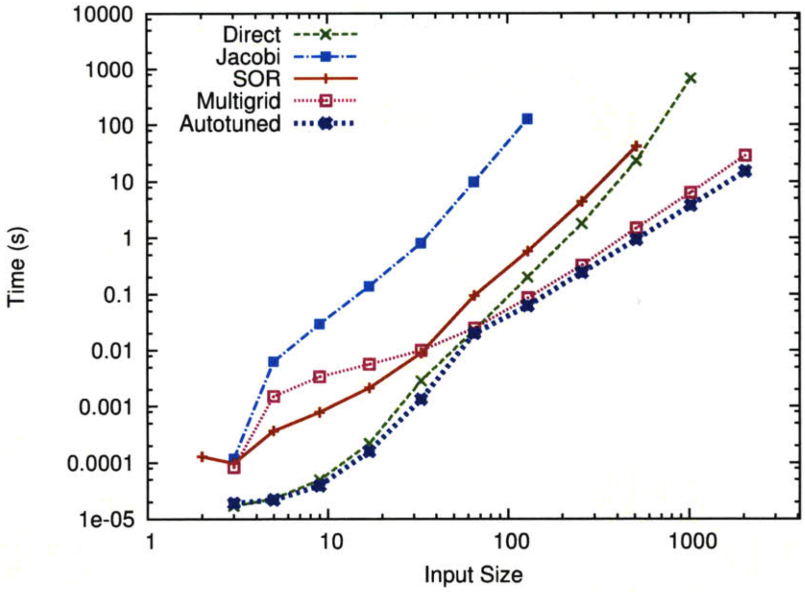


Figure 4.7: Performance for algorithms to solve Poisson's equation up to an accuracy of 10^9 using 8 cores. The iterated SOR algorithm uses the corresponding optimal weight ω_{opt} for each of the different input sizes

4.2 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

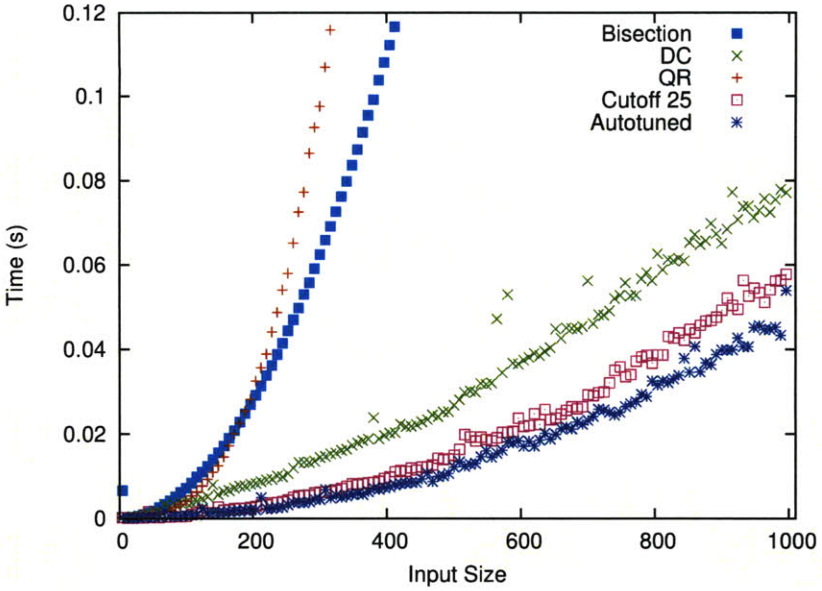


Figure 4.8: Performance for Eigenproblem on 8 cores. "Cutoff 25" corresponds to the hard-coded hybrid algorithm found in LAPACK.

eigenvectors, and the cost associated with each algorithm [8].

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.

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

```
EIG($T$)
either
    Use QR to find $\Lambda$ and $X$
    Use BISECTION to find $\Lambda$ and $X$
    Recursively call EIG on submatrices $T_1$ and $T_2$ to get $\Lambda_1$, $X_1$, $\Lambda_2$ and $X_2$. Use results to compute $\Lambda$ and $X$.
end either
```

Figure 4.9: Pseudo code for eigenvector solve.

pseudo code for this is shown in Figure 4.9. 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 4.8, 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 [2].

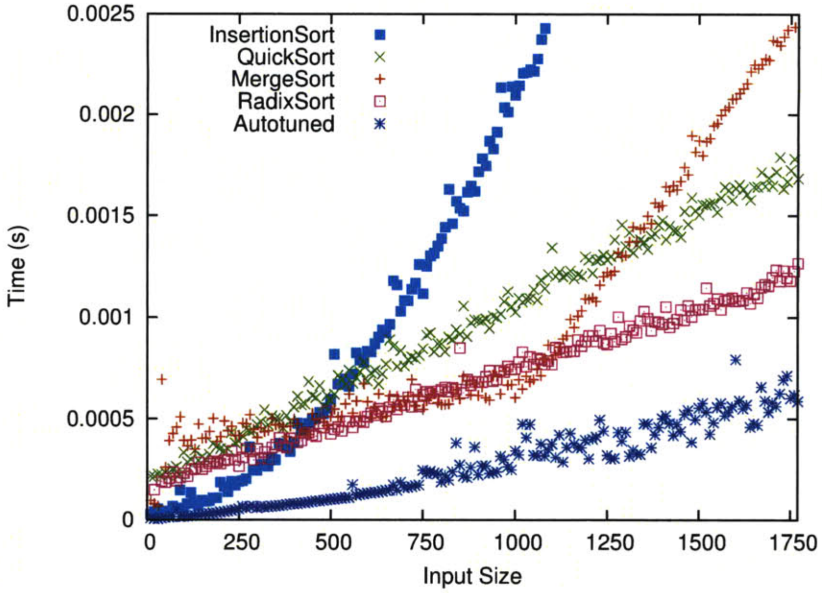


Figure 4.10: Performance for sort on 8 cores.

4.3 Sort

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 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). The concepts behind the choices in sort are discussed in Section 1.1. 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 4.10 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

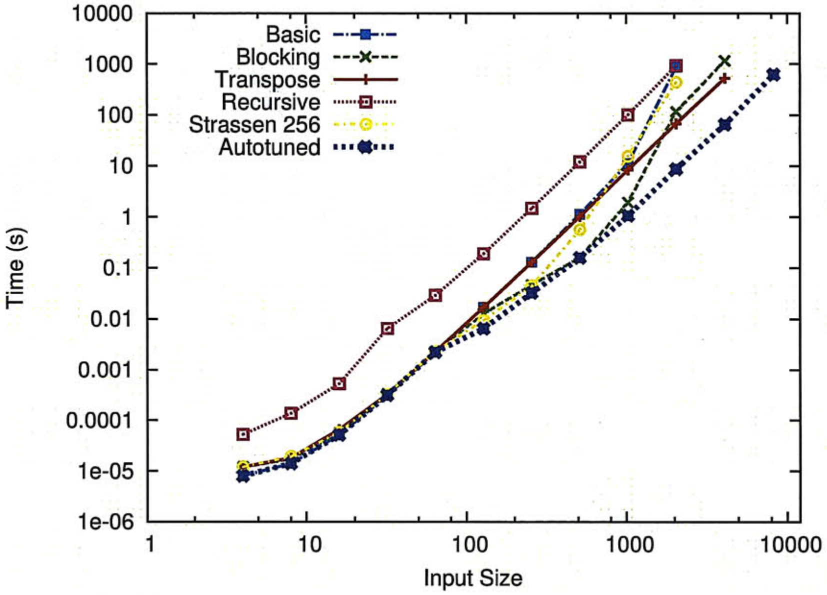


Figure 4.11: Performance for Matrix Multiply on an 8 cores. "Strassen 256" uses strassen algorithm to decompose until n=256 when it switches to basic matrix multiply.

this step ladder performance pattern disappears.

4.4 Matrix Multiply

The full PetaBricks code for the basic version of matrix multiply can be found in the introduction (Figure 2.1). In addition to that example code we also implemented *Strassen algorithm* (fast matrix multiply). This results in four recursive decompositions and one base case, for a total of five algorithmic choices. 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 4.11 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" dimension shown in Figure 2.1. The other two recursive decompositions are equivalent to blocking and thus are not shown. The autotuned algorithm uses a mixture of blocking, transpose, and the recursive decomposition.

Chapter 5

Results

Figures 4.7, 4.8, 4.10, and 4.11 compare the performance of our autotuned algorithms to implementation that only utilize a single algorithmic choice. In all cases the autotuned algorithm has significant speedup. These 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.

5.1 Autotuning Parallel Performance and Scalability

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

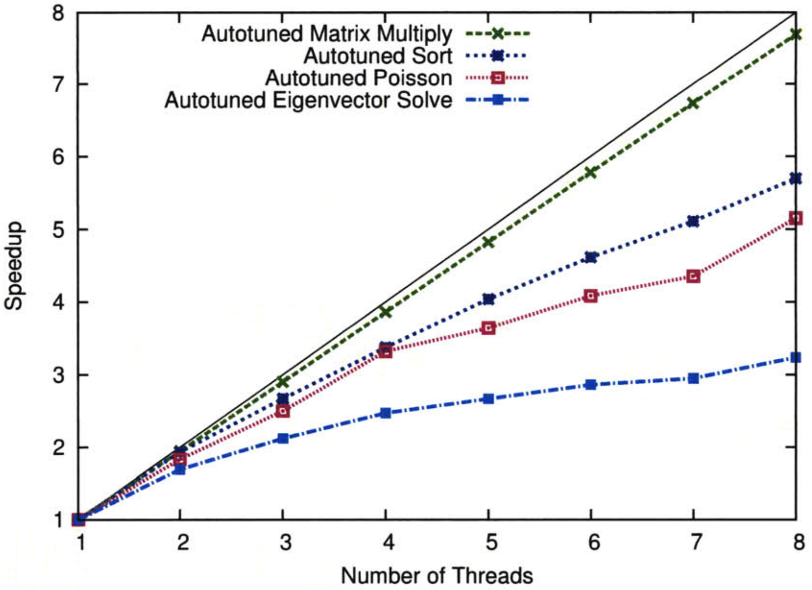


Figure 5.1: **Parallel scalability.** Speedup as more worker threads are added. Run on an 8-way (2 processor × 4 core) x86_64 Intel Xeon System.

1 core (as seen in Table 5.1).

5.2 Effect of Architecture on Autotuning

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 5.1 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.

		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 5.1: 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 5.2.

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 5.2: 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.

Table 5.2 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.

Chapter 6

Related Work

A number of empirical autotuning frameworks have been developed for building efficient, portable libraries in specific domains. PHiPAC [5] is an autotuning system for dense matrix multiply, generating portable C code and search scripts to tune for specific systems. ATLAS [33, 34] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LAPACK. FFTW [11, 12] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [16] for sparse matrix computations, SPIRAL [24, 28, 10] for digital signal processing, UHFFT [1] for FFT on multicore systems, OSKI [32] for sparse matrix kernels, and autotuning frameworks for optimizing sequential [19, 20] and parallel [22] sorting algorithms.

In addition to these systems, various performance models and tuning techniques [35, 31, 6, 38, 18, 39] have been proposed to evaluate and guide automatic performance tuning.

There are a number of systems that provide high-level abstractions to ease the burden of programming adaptive applications. STAPL [27] is an C++ template library that support adaptive algorithms and autotuning. Paluska et al. propose a programming framework [23] 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. [3] and Kessler et al. [17] provide a framework for composing parallel algorithmic components.

There is a body of research into applying machine learning to more traditional optimizations such as loop unrolling and instruction ordering. MILEPOST [14] is an extension to GCC that uses machine learning to optimize programs for specific processors. Moss and Page [21] introduce techniques for exploring many different fine grained instruction orderings. Donadio et al. [9] introduced a language for expressing families of loop transformations with different parameters.

ADAPT [29, 30] augments compile-time optimizations with run-time optimizations based on dynamic information about architecture, inputs, and performance. It does not support making algorithmic changes, but instead focuses on lower level compiler optimizations.

Atune-IL [26] allows programmers to annotate their parallel programs with different parameters. Their system then uses autotuning techniques to set these parameters and improve performance.

FLAME [15] 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 [37] 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 [36] 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.

Chapter 7

Future Work

We are continuing to improve the PetaBricks language, expand our benchmark suite, and improve performance. An interesting additional future direction is adding a distributed memory backend to our compiler so that we can run unmodified PetaBricks programs on clusters. Moving to clusters will add even more choices for the compiler to analyze, as it must decide both what algorithm to use and where to run it. A key challenge in this area is autotuning the management of data. Since distributed systems are often heterogeneous, autotuning can offer greater benefits since the trade offs become more complex. Finally, we are also exploring compiler backends for less traditional architectures such as graphics cards and embedded systems.

Chapter 8

Conclusions

Getting consistent, scalable, and portable performance is difficult. The compiler has the daunting task of selecting an effective optimization configuration from possibilities with drastically different impacts on the performance. No single choice of parameters can yield the best possible result as different algorithms may be required under different circumstances. The high performance computing community has always known that in many problem domains, the best sequential algorithm is different from the best parallel algorithm. Varying problem size and data sets will also require different algorithms. Currently there is no viable way for incorporating all these algorithmic choices into a single program to produce portable programs with consistently high performance.

In this paper we introduced 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. We have shown the efficacy of this system by developing a non-trivial suite of benchmark applications. One 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.

Trends show us that programs have a lifetime running into decades while architectures are much shorter lived. With the advent of multicore processors, architectures are experiencing drastic changes at an even faster rate. Under these circumstances, it is a daunting task to write a program that will perform well not only on today's architectures but also those of the future. We believe that PetaBricks can give programs the portable performance needed to increase their effective lifetimes.

Bibliography

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

[2] 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.

[3] Jesper Andersson, Morgan Ericsson, Christoph Kessler, and Welf Lowe. Profile-guided composition. *Lecture Notes in Computer Science*, 4954:157–164, March 2008.

[4] 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.

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

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

[7] 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.

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

[9] 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. Springer Verlag, 2005.

[10] 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.

[11] 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. IEEE, 1998.

[12] 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".

[13] 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.

[14] 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.

[15] 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.

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

[17] 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. IOS Press, 2007.

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

[19] 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.

[20] 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. IEEE Computer Society, 2005.

[21] 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. ACM.

[22] 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.

[23] 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. IEEE Computer Society.

[24] Markus Puschel, Jose M. F. Moura, Jeremy R. Johnson 2and David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Aca Gacic Franz Franchetti, Robbert W. Johnson Yevgen Voronenko, Kang Chen, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. In *Proceedings of the IEEE*, volume 93, pages 232–275. IEEE, Feb 2005.

[25] Richard H. Rand. *Computer algebra in applied mathematics: an introduction to MACSYMA*. Number 94 in Research notes in mathematics. 1984.

[26] 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.

[27] 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. ACM.

[28] 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.

[29] 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.

[30] Michael Voss and Rudolf Eigenmann. High-level adaptive program optimization with adapt. *ACM SIGPLAN Notices*, 36(7):93–102, 2001.

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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. ACM.

[37] 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.

[38] 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. ACM.

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