A Scalable Auto-tuning Framework for Compiler Optimization

Ananta Tiwari[1], Chun Chen[2*], Jacqueline Chame[3],
Mary Hall[2] and Jeffrey K. Hollingsworth[1]

[1]University of Maryland
Department of Computer Science
College Park, MD 20740
{tiwari, hollings}@cs.umd.edu

[2]University of Utah
School of Computing
Salt Lake City, UT 84112
{chunchen, mhall}@cs.utah.edu

[3]University of Southern California
Information Sciences Institute
Marina del Ray, CA 90292
jchame@isi.edu

Abstract

We describe a scalable and general-purpose framework for auto-tuning compiler-generated code. We combine Active Harmony's parallel search backend with the CHiLL compiler transformation framework to generate in parallel a set of alternative implementations of computation kernels and automatically select the one with the best-performing implementation. The resulting system achieves performance of compiler-generated code comparable to the fully automated version of the ATLAS library for the tested kernels. Performance for various kernels is 1.4 to 3.6 times faster than the native Intel compiler without search. Our search algorithm simultaneously evaluates different combinations of compiler optimizations and converges to solutions in only a few tens of search-steps.

1 Introduction

The complexity and diversity of today's parallel architectures overly burdens application programmers in porting and tuning their code. At the very high end, processor utilization is notoriously low, and the high cost of wasting these precious resources motivates application programmers to devote significant time and energy to tuning their codes. This tuning process must be largely repeated to move from one architecture to another, as too often, a code that performs well on one architecture faces bottlenecks on another. As we are entering the era of petascale systems, the challenges facing application programmers in obtaining acceptable performance on their codes will only grow.

To assist the application programmer in managing this complexity, much research in the last few years has been devoted to *auto-tuning* software that employs empirical techniques to evaluate a set of alternative mappings of computation kernels to an architecture and select the mapping that obtains the best performance. Auto-tuning software can be grouped into three categories: (1) self-tuning library generators such as ATLAS, PhiPAC and OSKI for linear algebra and FTTW and SPIRAL for signal processing [21, 3, 20, 9, 22]; (2) compiler-based auto-tuners that automatically generate and search a set of alternative implementations of a computation [7, 24, 11]; and, (3) application-level auto-tuners that automate empirical search across a set of parameter values proposed by the application programmer [8, 16]. What is common across all these different categories of auto-tuners is the need to search a range of possible implementations to identify one that performs comparably to the best-performing solution. The resulting search space of alternative implementations can be prohibitively large. *Therefore, a key challenge that faces auto-tuners, especially as we expand the scope of their capabilities, involves scalable search among alternative implementations.*

As we look to the future, full applications will likely include a mix of auto-tuning software from the above

*This work was done when the author was at USC/ISI.

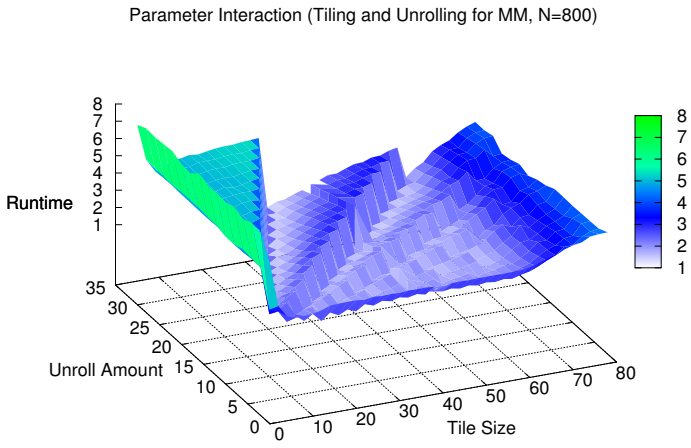


Figure 1. Parameter Search Space for Tiling and Unrolling (Figure is easier to see in color).

three categories: automatically-generated libraries, compiler-generated code and application-level parameters exposed to auto-tuning environments. Thus, applications of the future will demand a cohesive environment that can seamlessly combine these different kinds of auto-tuning software and that employs scalable empirical search to manage the cost of the search process.

In this paper, we take an important step in the direction of building such an environment. We begin with Active Harmony [8], which permits application programmers to express application-level parameters, and automates the process of searching among a set of alternative implementations. We combine Active Harmony with CHiLL [5], a compiler framework that is designed to support convenient automatic generation of code variants and parameters from compiler-generated or user-specified transformation recipes. In combining these two systems, we have produced a unique and powerful framework for auto-tuning compiler-generated code that explores a richer space than compiler-based systems are doing today and can empower application programmers to develop self-tuning applications that include compiler transformations.

A unique feature of our system is a powerful parallel search algorithm which leverages parallel architectures to search across a set of optimization parameter values. Multiple, sometimes unrelated, points in the search space are evaluated at each timestep. With this approach, we both explore multiple parameter interactions at each iteration and also have different nodes of the parallel system evaluate different configurations to converge to a solution faster. In support of this search process, CHiLL provides a convenient high-level scripting interface to the compiler that simplifies code generation and varying optimization parameter values.

The remainder of the paper is organized into five sections. The next section motivates the need for an effective search algorithm to explore compiler generated parameter spaces. Section 3 describes our search algorithm, which is followed by a high-level description of CHiLL in section 4. In section 5, we give an overview of the tuning workflow in our framework. Section 6 presents an experimental evaluation of our framework. We discuss related work in section 7. Finally, section 8 will provide concluding remarks and future implications of this work.

2 Motivation

Today's complex architecture features and deep memory hierarchies require applying nontrivial optimization strategies on loop nests to achieve high performance. This is even true for a simple loop nest like Matrix Multiply. Although naively tiling all three loops of Matrix Multiply would significantly increase its performance, the performance is still well below hand-tuned libraries. Chen et al [7] demonstrate that automatically-generated optimized code can achieve performance comparable to hand-tuned libraries by using a more complex tiling strategy combined with other optimizations such as data copy and unroll-and-jam. Combining optimizations, however, is not an easy task because loop transformation strategies interact with each other in complex ways.

Different loop optimizations usually have different

goals, and when combined they might have unexpected (and sometimes undesirable) effects on each other. Even optimizations with similar goals but targeting different resources, such as unroll-and-jam plus scalar replacement targeting data reuse in registers, and loop tiling plus data copy for reuse in caches, must be carefully combined. Unroll-and-jam generally has more impact on performance than tiling for caches, since reuse in registers reduces the number of loads and stores. In addition, in architectures with SIMD units, unroll-and-jam can be used to expose SIMD parallelism. The unroll factors must be tuned so that reuse and SIMD are exploited without causing register spilling or instruction cache misses. On the other hand, tiling plus data copying for reuse in caches changes the iteration order and data layout, and may affect reuse in registers and SIMD parallelism. When combining unroll-and-jam and tiling, both unroll and tile sizes must be tuned so that performance gains are complementary. Figure 1 illustrates these complex interactions by showing the performance of square matrix (of size 800×800) multiplication as a function of tiling and unrolling factors. Tiling factors range from 2 to 80 and unrolling factors from 2 to 32. We see a corridor of best performing combinations along the x-y diagonal where tiling and unrolling factors are equal, and smaller corridors when tile factors are multiples of unroll factors. The best performing code variant used a tiling factor of 24 and unrolling factor of 24 and achieves a performance of 845 MFLOPS.

Empirical optimization can compensate for the lack of precise analytical models by performing a systematic *search* over a collection of automatically generated code variants. Each variant exposes a set of parameters that controls the application of different transformation strategies. Parameter configurations for variants serve as points in the search space and the objective function values[1] associated with the points are gathered by actually running the variants on the target architecture. The success of empirical search is largely driven by how well the chosen *search* algorithm navigates the search space. The search space shown in Figure 1 is not smooth and contains multiple minimas and maximas. The best and the worst configurations are a factor of six different.

Active Harmony, an automated performance tuning infrastructure supporting both online and offline tuning for scientific applications[2], provides a selection of search algorithms designed specifically to deal with search spaces where the explicit definition of the objective function is not available. Finding a good set of loop transformation parameters is a good example of the type of search that the Harmony system is designed to address.

In the next section, we describe our parameter tuning algorithm for compiler generated parameter spaces.

Algorithm 1 : PRO for Compiler Optimization

Start with initial simplex with vertices $\{v_0^0, \cdots, v_0^n\}$ and evaluate $f(v_0^j), \quad j = 0, \cdots, n$ in parallel.
$k = 0$
while Stopping Criteria Not Valid **do**
  Reorder simplex vertices, so that $f(v_k^0) \leq \cdots \leq f(v_k^n)$
  Compute n reflection pts $r_k^j = \Pi\left(2v_k^0 - v_k^j\right)$, and function values $f(r_k^j), \quad j = 1, \cdots, n$ in parallel. {*Reflection step*}
  $l = \arg\min_j f(r_k^j)$ {*Most Promising Point*}
  if $f(r_k^l) < f(v_k^0)$ **then**
    Compute n expansion pts $e_k^j = \Pi\left(3v_k^0 - v_k^j\right)$, and function values $f(e_k^j), \quad j = 1, \cdots, n$ in parallel. {*Expansion checking step*}
    if $f(e_k^l) < f(r_k^l)$ **then** {*Accept Expansion*}
      $v_{k+1}^j = e_k^j \quad j = 1, \cdots, n$
    else {*Send HALT signal to all processes and accept reflection*}
      $v_{k+1}^j = r_k^j \quad j = 1, \cdots, n$
    end if
  else {*Accept shrink*}
    Compute $\Pi\left(v_{k+1}^j = 0.5v_k^0 + 0.5v_k^j\right)$, and $f(v_{k+1}^j) \quad j = 1, \cdots, n$ in parallel. {*Shrink step*}
  end if
  k = k+1
end while

3 Parameter Tuning Algorithm

As previously shown, the loop transformation parameters interact with each other in complex ways. The search algorithm used to explore the parameter spaces of compiler-optimized computations must take into account such interactions and be able to tune the parameters simultaneously. The simultaneous tuning, however, leads to added dimensions in the search space. For our purposes, we use a modified version of the Parallel Rank Ordering (PRO) algorithm proposed by

[1]The objective function values associated with points in the search space can be any desired metric of performance (for example - time per timestep, MFLOPS, cache utilization etc.).

[2]Online tuning refers to adapting performance related parameters during runtime. Offline tuning refers to tuning for parameters that can be selected at compile/launch time but remain fixed throughout the execution.

Tabatabaee et al [19]. Although the original PRO algorithm can effectively deal with high-dimensional search spaces with unknown objective functions, there are two main differences between the type of search PRO was designed for and the type of search we want to conduct. First, PRO was designed for online tuning of SPMD-based parallel applications while our approach needs an offline search. Secondly, Tabatabaee et al only looked at (hyper) rectangular search spaces instead of the more general parameter space used in our compiler optimization. In addition, we modified the initial simplex construction method to better suit our goal of using all available parallelism. We describe each modification in detail later in this section. We will refer to the modified algorithm as PRO-C (PRO for Compiler Optimization).

The parameter tuning algorithm is given in Algorithm 1. For a function of N variables, PRO-C maintains a set of kN points forming the vertices of a simplex in an N-dimensional space. Each simplex transformation step[3] (lines 5, 8 and 15) of the algorithm generates up to $kN-1$ new vertices by reflecting, expanding, or shrinking the simplex around the best vertex. After each transformation step, the objective function value, f, associated with each of the newly generated points are calculated in parallel. The reflection step is considered *successful* if at least one of the $kN-1$ new points has a better f than the best point in the simplex. If the reflection step is not successful, the simplex is shrunk around the best point. A successful reflection step is followed by expansion check step (line 9). If the expansion check step is successful, the expanded simplex is accepted. Otherwise, the reflected simplex is accepted and the search moves on to the next iteration. A graphical illustration for reflection, expansion and shrink steps are shown in Figure 2 for a 2-dimensional search space and a 4-point simplex. In the remainder of this section, we describe the modifications that we made to the original PRO algorithm to make it suitable for searching compiler generated parameter spaces.

3.1 Parallelizing Expansion Check Step

Recall that each simplex transformation step generates up to $kN-1$ new vertices. The time required to complete the parallel evaluation of these new vertices is the time taken by the worst performing vertex. The decision to introduce the expansion-check step in

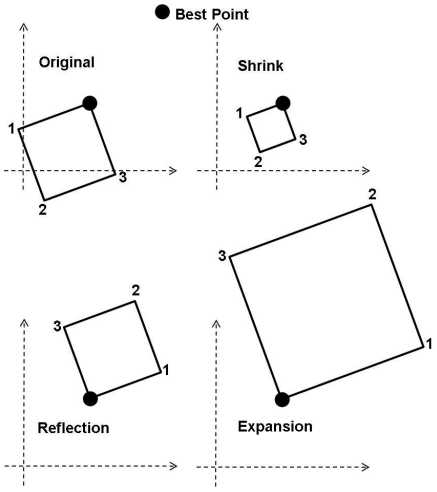


Figure 2. Simplex Transformation steps.

PRO was motivated by the observation that there are some expansion points with very poor performance. For online tuning of SPMD-based parallel applications, such configurations slow down not only the search but also the execution of the application itself. To avoid these time consuming instances, before evaluating all expansion points, PRO first calculates the expansion point performance of only the most promising case[4] at the expense of parallelism. If the expansion checking step is successful, the algorithm performs expansion of other points in the simplex. Assuming we have kN nodes available, each iteration of PRO, therefore, takes at most three search steps (reflection, expansion check and expansion).

In an offline parallel search, however, processors participating in the search are independent, which allows us to take full advantage of the underlying parallelism while still avoiding expansion points with poor performance. To that end, PRO-C evaluates all expansion points and the decision to accept or reject the expanded simplex is based on the performance of the most promising case. If the performance reported by the most promising case is worse than that of the best point in the reflected simplex, our system sends a signal to all the other processors to stop the evaluation of their candidate configurations and accepts the reflected simplex. The expansion of the simplex is accepted if the performance of the most promising case is better than the best vertex in the reflected simplex. With this modification, we not only reduce the number of steps within one iteration of the search algorithm to at

[3]Each simplex transformation is considered to be a *search-step* within one search iteration. One iteration of the search algorithm consists of all the simplex transformations that happen between successive reflection steps.

[4]Most promising point is the point in the original simplex whose reflection around the best point returns a better function value.

most two (reflection-expansion and reflection-shrink) but also increase parallelism.

3.2 Projection Operator for Arbitrary Space

Offline tuning of loop transformation parameters is a constrained optimization problem. Therefore in each step we have to make sure that the computed points are admissible, i.e. they satisfy the constraints. The projection operator, function $\Pi(\cdot)$ (used in the pseudo-code), takes care of this problem by mapping points that are not admissible to admissible points. PRO uses a simple method that independently maps the computed value of the parameter to its lower or upper limit, whichever is closer. This method works well for hyper-rectangular search spaces, but not when we have an arbitrarily shaped space defined by (possibly non-linear) constraints on parameter values. Our projection operator accommodates such arbitrarily shaped spaces by projecting an inadmissible point to its nearest admissible neighbor. We define distance between two points using L_1 distance, which is the sum of the absolute differences of their coordinates. The nearest neighbor of an inadmissible point (calculated in terms of L_1) will thus be a legal point with the least amount of *change* (in terms of parameter values) summed over all dimensions.

Computing the least L_1 distance unfortunately involves finding the nearest neighbors in a high dimensional space, which is a computationally intensive task. After experimenting with multiple nearest-neighbor algorithms, we adopted the Approximate Nearest Neighbor[5] (ANN) [2] algorithm for two reasons. First, for approximate neighbors, ANN has linear space requirements and logarithmic time complexity on the number of points in the search space. Second, an efficient implementation of the ANN library is available [15]. The library supports a variety of metrics to define distance between two points, including L_1 distance metric. We set $\epsilon = 0.5$, which, for L_1 distance, means error of at most one along at most one dimension is tolerated, which is a fairly small price to pay for logarithmic query time.

3.3 Simplex Construction and Size

The initial simplex, with size kN, needs to be non-degenerate so that it can span the whole parameter space; therefore, kN must be at least $N+1$, where N is the number of tunable parameters. For a discrete parameter space, PRO's simplex construction method can generate only up to $2N$ points. In PRO-C, we extend the method to generate points for any $kN \geq N+1$. To exploit all available parallelism, kN can be set to the number of resources/processors available.

Unlike PRO's strategy of starting the search at the center of the search-space (which is hard to ascertain in a high-dimensional constrained space), we randomly select kN points at the start of the algorithm. The first iteration of the algorithm evaluates these random configurations. The initial simplex is constructed by randomly sampling points at distance d (L_1 distance) from the best performing point. The set of search directions/vectors (from the initial best point to the sampled points) generated in this fashion is guaranteed to be a linearly independent set, which is important because this property gives us kN unique parameter interactions.

In section 4, we describe CHiLL - our loop transformation and code generation framework.

4 CHiLL: A Framework for Composing High-Level Loop Transformations

Automatic tuning requires a compiler to be able to generate different codes rapidly during the search by adjusting parameter values, without costly compiler reanalysis. It also demands that the compiler have a clean interface to a separate parameter search engine. CHiLL [5, 6], a polyhedral loop transformation and code generation framework, provides such capability for composing high-level loop transformations with a script interface to describe the transformations and search space to the search engine. Polyhedral representation of loops facilitates compilers to compose complex loop transformations in a mathematically rigorous way to ensure code correctness. However, existing polyhedral frameworks are often too limited in supporting a wide array of loop transformations (for both perfect and imperfect loop nests) required to achieve high-performance on today's computer architectures. CHiLL employs new design features such as *iteration space alignment* and *auxiliary loops* to greatly expand the capability of polyhedral framework. Further, its high-level script interface allows compilers or application programmers to use a common interface to describe parameterized code transformations to be applied to a computation, whose parameters can be instantiated by an external search engine to find the best-performing implementation. We now briefly describe CHiLL's new features.

[5]Given any $\epsilon > 0$, a $(1+\epsilon)$-nearest neighbor of q is a point $p\prime \in S$ s.t.

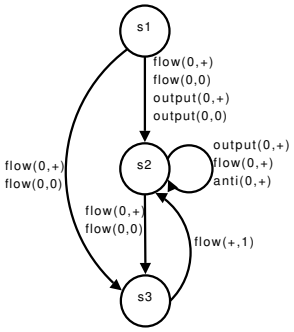
$$\frac{dist(p,q)}{dist(p\prime,q)} \leq 1+\epsilon$$

```
  DO I=2, N
s1   SUM(I)=0
    DO J=1, I-1
s2     SUM(I)=SUM(I)+A(J,I)*B(J)
s3   B(I)=B(I)-SUM(I)
```

(a) Original code

$IS_1 : \{[i,j] \mid 2 \leq i \leq N \wedge j = 1\}$
$IS_2 : \{[i,j] \mid 1 \leq j < i \leq N\}$
$IS_3 : \{[i,j] \mid 2 \leq i \leq N \wedge j = i - 1\}$

(b) Aligned iteration spaces



(c) Dependence graph

$t_{s_1} : \{[*,i,*,j,*] \rightarrow [0,i,0,j,0]\}$
$t_{s_2} : \{[*,i,*,j,*] \rightarrow [0,i,1,j,0]\}$
$t_{s_3} : \{[*,i,*,j,*] \rightarrow [0,i,2,j,0]\}$

(d) Transformation relations to generate the original loop nest in (a)

Figure 3. Representing Loop Nests and Transformations.

4.1 Polyhedral Representation

In a polyhedral representation, a loop nest is represented by the collection of iteration spaces of the statements inside the loop nest. Each statement has its own iteration space, derived from its enclosing loops respectively. Thus for imperfect loop nests the number of dimensions of the iteration spaces of individual statements derived initially may be different. Additional *iteration space alignment* brings each statement to be represented in a same unified iteration space. To generate imperfectly nested transformed loops, *auxiliary loops* are added to determine lexicographical order among loops at each loop level. We will discuss both concepts in detail below.

Iteration space alignment can be thought as a generalization of code sinking and loop fusion. For an imperfect loop nests such as in Figure 3(a), CHiLL extracts the iteration space for each statement as in Figure 3(b). Note that in CHiLL's representation every statement in the loop nest has the same number of dimensions in its iteration space. Although `s1` and `s3` are only surrounded by one loop `I`, their iteration spaces are still 2-dimensional; more precisely, each represents a line *aligned* in a 2-dimensional iteration space. Once the iteration spaces of all statements are aligned in the same iteration space, CHiLL can transform perfect and imperfect loop nests in a systematic way and the legality of a transformation can be determined in the same way as perfect loop nests, i.e., from data dependences (e.g. 3(c)) prior to the transformation. The complete algorithm for iteration space alignment can be found in [5].

Auxiliary loops are introduced to allow a systematic code generation strategy for both perfect and imperfect loop nests. If the aligned iteration spaces only include dimensions for each loop level, there would be no information available as to the relationship or required execution order among statements or how loops and statements would be organized at a specific loop level. To keep a simple and robust polyhedral scanning strategy for code generation, an auxiliary loop is associated with each loop level in the original nest. Each auxiliary loop carries the execution order of statements and loops at its associated level. An additional auxiliary loop is associated with the statements within the deepest level of the iteration space, and carries the execution order of these statements. By setting different constant integer values for these auxiliary loops, CHiLL establishes the lexicographical order of loops at each loop level as well as the lexicographical order of statements in the innermost loop. So for an n-deep loop nest, we have $(2n+1)$-dimension iteration spaces as $[c_1, l_1, c_2, l_2, \ldots, c_n, l_n, c_{n+1}]$, where c_i's are auxiliary loops. Each loop transformation from an n-deep loop nest to a new m-deep loop nest is represented as a set of relations:

$$t : \{[c_1, l_1, \ldots, c_n, l_n, c_{n+1}] \rightarrow [c'_1, l'_1, \ldots, c'_m, l'_m, c'_{m+1}] | \cdots\}.$$

Figure 3(d) shows the transformation relations to generate the original loop nest, with the initial auxiliary loop values unknown yet. Since only constant values are allowed in auxiliary loops, no loops are generated in the final transformed code.

4.2 Code Transformations - Recipes

CHiLL takes as input the original code and a loop transformation recipe (a CHiLL script) describing how

to optimize the code. Each line of the script describes a transformation to be applied on an existing loop representation. For illustration purposes, we list some most common high-level loop transformations below. As a general rule, each loop transformation affects a set of statements within the specified loop.

permute*([stmt],order)*: the loop order of *stmt* is permuted to the new *order*, which is represented by a sequence of integers identifying the loops. If permute does not have a *stmt* parameter, it indicates that the loop order of all statements should be permuted.

tile*(stmt,loop,size,[outer-loop])*: Tile loop at level *loop* of *stmt* with the tile controlling loop at loop level *outer-loop* (default value 1), with tile size *size*.

unroll*(stmt,loop,size)*: Unroll *stmt*'s loop at level *loop* by unroll factor *size*. For all unrolled statements, the inner loop bodies below loop level *loop* are jammed together.

datacopy*(stmt,loop,array,[index])*: For the specified *array* in *stmt*, a temporary array copy construction is introduced for all *array* accesses touched within loop level *loop*. The *index* (default value 0) specifies which subscript in *array* corresponds to the new temporary array's first index (assuming Fortran array layout). The *array* accesses in *stmt* are replaced by appropriate temporary array accesses.

split*(stmt,loop,condition)*: Split *stmt*'s loop level *loop* into multiple loops according to *condition*. The original *stmt*'s iteration space will satisfy *condition*. The iteration space satisfying the complement of *conditions* will be split into new statements.

nonsingular*(matrix)*: Transform the perfect loop nest according to nonsingular *matrix*. This includes both unimodular and nonunimodular transformations.

In the next section, we describe how CHiLL and Active Harmony frameworks interact with each other to generate a set of alternative implementations of computation kernels and automatically search and select the one with the best-performing implementation.

5 Overall System Workflow

Figure 4 shows the overall workflow of our system. In the proposed framework, code transformation recipes and parameter specifications (i.e. parameter domain and constraints) can be either generated by the compiler automatically or by the users tuning their application code. With this flexibility, our approach can support both fully automated compiler optimizations and user-directed tuning. For our experiments,

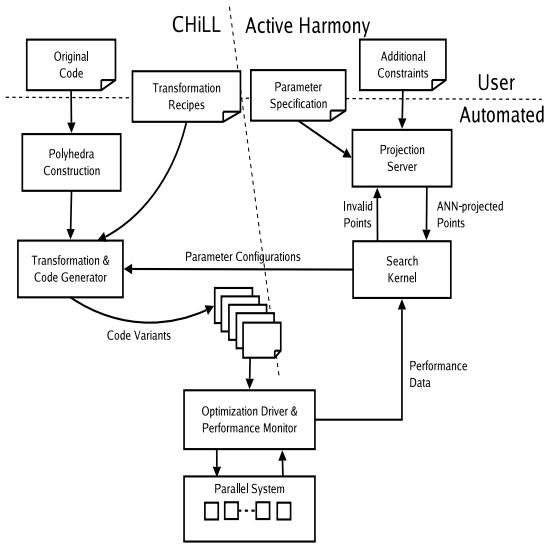


Figure 4. Overall System Workflow Diagram.

we translate loop transformation sequences from the algorithms presented by Chen et al [7] to CHiLL scripts. Specifications for unbound parameters in the scripts are derived using simple heuristics based on architectural parameters (e.g., consider cache capacity to generate constraints for tile-sizes). We elaborate more on parameter specification in the next section. If a user, with domain knowledge, wants more control over what part of the parameter space to focus on, he/she can provide additional constraints to fine-tune the search space. Using the parameter specifications, we normalize the domain of each parameter onto our internal integer based coordinate system. This step is necessary to ensure that the differences in the range of values parameters can take in different dimensions do not unduly influence the L_1 distance metric.

Parameters that appear in one or more constraints are considered to be interdependent and are evaluated as sets. For example, tile-size parameters for multiple loops may appear in one or more cache capacity constraints. A simple constraint solver is then used to enumerate points for each of these sets. Projection of an inadmissible point to a valid point in the search space is done (by the projection server) separately for different groups of parameters.

At each search step, Active Harmony's search-kernel requests CHiLL's code-generator to generate code variants with given sets of parameters for loop transformations. The CHiLL generated code variants are then compiled and run in parallel on the target architecture by the optimization driver. Measured performance values are consumed by the search-kernel to make simplex transformation decisions.

Table 1. Kernels used for experiments

Kernel	*Naive Code*	*Transformation Recipe*	*Constraints*

MM	`DO K = 1, N`
	`  DO J = 1, N`
	`  DO I = 1, N`
	`  C[I,J] = C[I,J]+A[I,K]*B[K,J]`
		`permute([3,1,2])`
		`tile(0,2,TJ)`
		`tile(0,2,TI)`
		`tile(0,5,TK)`
		`datacopy(0,3,2,1)`
		`datacopy(0,4,3)`
		`unroll(0,4,UI)`
		`unroll(0,5,UJ)`
			$TK \times TI \leq \frac{1}{2}\left(\frac{size_{L2}}{2}\right)$
			$TK \times TJ \leq \frac{1}{2}\left(\frac{size_{L1}}{2}\right)$
			$UI \times UJ \leq size_R$
			$TI,\ TJ,\ TK \in [0, 2, 4, \ldots, 512]$
			$UI,\ UJ \in [1, 2, \ldots, 16]$

TRSM	`DO J = 1, N`
	`  DO K = 1, N`
	`  DO I = K + 1,N`
	`  B(I,J) = B(I,J) - B(K,J)*A(I,K)`
		`permute([1,3,2])`
		`tile(0,3,TK)`
		`split(0,2,L3>=L1+TK)`
		`tile(0,3,TI,2)`
		`tile(0,3,TJ,2)`
		`datacopy(0,3,2)`
		`datacopy(0,4,3,1)`
		`unroll(0,4,UJ1)`
		`unroll(0,5,UI1)`
		`datacopy(1,2,3,1)`
		`unroll(1,2,UJ2)`
		`unroll(1,3,UI2)`
			$TK \times TK \leq \frac{1}{2}\left(\frac{size_{L2}}{2}\right)$
			$TK \times TJ \leq \frac{1}{2}\left(\frac{size_{L1}}{2}\right)$
			$TK \times TI \leq \frac{1}{2}\left(\frac{size_{L2}}{2}\right)$
			$UI1 \times UJ1 \leq size_R$
			$UI2 \times UJ2 \leq size_R$
			$TI, TJ, TK \in [0, 2, 4, \ldots, 512]$
			$UI1, UJ1, UI2, UJ2 \in [1, 2, \ldots, 16]$

Jacobi	`DO K = 2, N-1`
	`DO J = 2, N-1`
	`DO I = 2, N-1`
	`A(I,J,K) = C*(B(I-1,J,K)+B(I+1,J,K)+`
	`B(I,J-1,K)+B(I,J+1,K)+`
	`B(I,J,K-1)+B(I,J,K+1))`
		`original()`
		`tile(0, 3, TI)`
		`tile(0, 3, TJ)`
		`tile(0, 3, TK)`
		`unroll(0,5,UJ)`
			$TI, TJ, TK \in [0, 2, 4, \ldots, 512]$
			$UJ \in [1, 2, \ldots, 16]$

6 Experimental Results

In this section, we present an experimental evaluation of our framework. First, we use a Matrix Multiplication kernel to explore the effectiveness of PRO-C on the search space for loop transformation parameters. We study how the size of the initial simplex (and hence the degree of parallelism) affects the convergence and performance of the search algorithm. In the second part, we use our framework to optimize two additional computational kernels - Triangular Solver (TRSM) and Jacobi. The use of linear algebra kernels - Matrix Multiplication and Triangular Solver - was motivated by our goal to compare the effectiveness of our framework to well tuned codes. The results for the Jacobi kernel show that our underlying polyhedral framework is a general-purpose loop transformation tool, which can handle arbitrary code beyond the linear algebra domain. In addition, MM, TRSM and Jacobi all exhibit complex parameter interactions (discussed in section 2) for today's computer architectures. For all the kernels, we provide the original code, the transformation recipe and the constraints on unbound parameters in Table 1.

The experiments were performed on a 64-node Linux cluster. Each node is equipped with dual Intel Xeon 2.66 GHz (SSE2) processors. L1-cache and L2-cache sizes are 128 KB and 4096 KB respectively. We compare the performance of our code versions with those of the native compiler (ifort 10.0.026, compiled with -O3 -xN). When compiling our transformed code, we turn off the native compiler's loop transformations to prevent them from interfering with our optimizations. For Matrix Multiplication and Triangular Solver, we present the performance of ATLAS (version 3.8) self-tuning libraries. In addition to a near exhaustive sampling of the search space, ATLAS uses carefully hand-tuned BLAS routines contributed by expert programmers. To make a meaningful comparison, we provide the performance of the *search-only* version of ATLAS - code generated by the ATLAS Code Generator via pure empirical search. The search-only version was generated by disabling the use of architectural defaults and turning off the use of hand-coded BLAS routines. For all our experiments, unroll factors and tile sizes are constrained by the storage capacity of their associated memory hierarchy levels. In addition, for tile sizes, we use a simple heuristic which tries to fit references with temporal reuse into half of the cache, leaving the other half for references with spatial or no reuse.

6.1 Performance of PRO-C

In this section, we use Matrix Multiplication (MM) to demonstrate the effectiveness of parallel search. The optimization strategy reflected in the transformation recipe in Table 1 exploits the reuse of $C(I, J)$ in registers, and the reuse of $A(I, K)$ and $B(K, J)$ in caches (A and B have the same amount of temporal reuse, carried by different loops). The transformation recipe applies tiling to B in the L1 cache and A in the L2

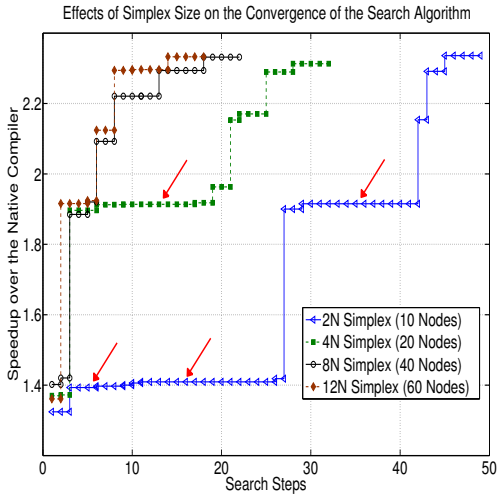


Figure 5. Effects of Different Degree of Parallelism on the Convergence of PRO-C.

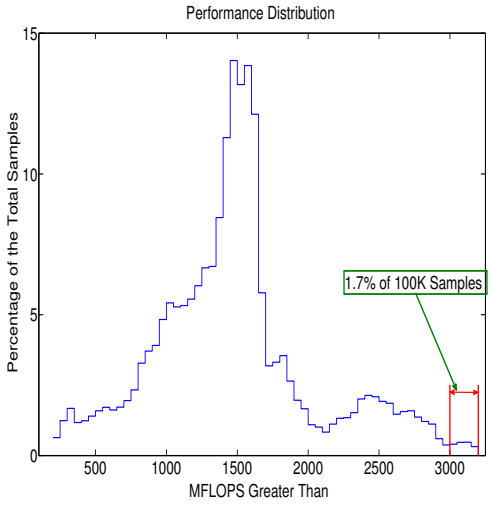


Figure 6. Performance Distribution for randomly chosen MM Configurations

cache. Data copying is applied to avoid conflict misses. In addition, to expose SSE optimization opportunities to the Intel compiler, the copying of A transposes the data into the temporary array. The values for the five unbound parameters TI, TJ, TK, UI and UJ are determined by the search algorithm.

To study the effect of simplex size, we considered four alternative simplex sizes - $2N$ (10 Nodes), $4N$ (20 Nodes), $8N$ (40 Nodes) and $12N$ (60 Nodes), where N is the number of unbound parameters ($N = 5$ for this experiment). Each simplex was constructed around the same initial point, which was randomly selected from the search space at the beginning of the experiment. The search algorithm was run for a square matrix of size 800×800. The results for this experiment are summarized in Table 2.

Figure 5 shows the performance of the best point in the simplex across search steps. Search conducted with $12N$ and $8N$ simplices clearly use fewer search steps than the search conducted with smaller simplices. Recall from our discussion in section 2 and from Figure 1 that loop transformation parameter space is not smooth and contains multiple local minimas and maximas. The existence of long stretches of consecutive search steps with minimal or no performance improvement (marked by arrows in Figure 5) in $2N$ and $4N$ cases show that more search steps are required to get out of local minimas for smaller simplices. At the same time, by effectively harnessing the underlying parallelism, $8N$ and $12N$ simplices evaluate more unique parameter configurations (see Table 2) and get out of local minimas at a faster rate.

Table 2. MM Results - Alternate Simplex Sizes

	$2N$	$4N$	$8N$	$12N$
Number of Function Evals.	276	571	750	961
Number of Search Steps	49	32	22	18
Speedup over Native	2.30	2.33	2.32	2.33

Results summarized in Table 2 also show that as the simplex size increases, the number of search steps decreases, thereby confirming the effectiveness of increased parallelism. Using a $12N$ initial simplex, the search converges to a solution 2.7 times faster than using $2N$ initial simplex.

The next question regarding the effectiveness of our framework relates to the quality of the search result. To answer this question, we selected 100,000 uniformly distributed samples from the search space, which has over 70 million total points, and evaluated the performance associated with all the samples. The performance distribution is shown is Figure 6. Approximately 1.7% of the total samples report performance greater than 3 GFLOPS. The best performance (3.22 GFLOPS) was associated with the configuration $TI = 160$, $TJ = 6$, $TK = 162$, $UI = 1$ and $UJ = 6$. For the same problem size, our code delivers 3.17 GFLOPS. The result demonstrates PRO-C's effectiveness on compiler-generated search spaces.

Finally, figure 7 shows the performance of the code variant produced by a $12N$ simplex across a range of

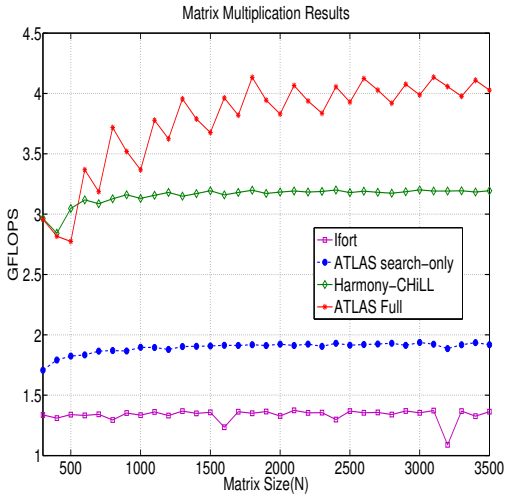


Figure 7. Results for MM Kernel

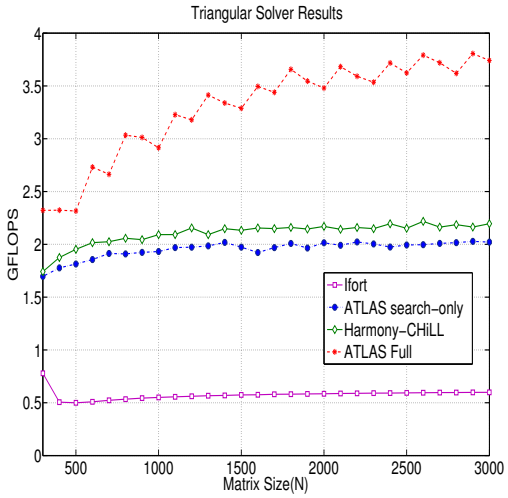


Figure 8. Results for TRSM Kernel

problem sizes along with the performance of native compiler, ATLAS' search-only and full version. Our code version performs, on average, 2.36 times faster than the native compiler. The performance is 1.66 times faster than the search-only version of ATLAS. Our code variant also performs within 20% of ATLAS' full version (with processor-specific hand coded assembly).

6.2 Triangular Solver (TRSM)

The optimization strategy for the TRSM kernel is outlined in its transformation recipe provided in Table 1. Two inner loops are permuted to reuse $B(I, J)$ in registers, and loops I and J are unrolled. For data reuse in cache, loop K is tiled first. The splitting condition is based on the decision to separate read access $B(I, J)$ from write access $B(K, J)$. After splitting, one subloop has non-overlapping read and write accesses and it is optimized in the same way as matrix multiplication. The other subloop has only one non-overlapping read access $A(I, K)$, for which data copy is applied to reduce cache conflict misses caused by this array reference.

Unbound parameters in the transformation recipe TI, TJ, TK, $UI1$, $UJ1$, $UI2$ and $UJ2$ form a seven dimensional parameter space. PRO-C used a 60-point simplex and converged to a solution in 55 steps evaluating 1,579 unique parameter configurations. Figure 8 shows the performance of the code variant along with the performance of the Native compiler and both ATLAS versions. The parameter configuration selected by PRO-C performs, on average, 3.62 times faster than

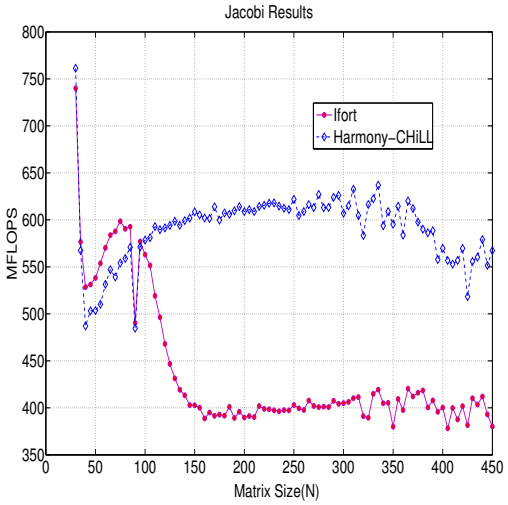


Figure 9. Results for Jacobi Kernel

the native Intel compiler. The performance, on average, is 1.07 times faster than the search-only version of ATLAS. However, ATLAS full-version (with processor-specific hand-tuned assembly) performance is 1.55 times faster than our code-variant.

6.3 Jacobi

The transformation recipe provided in Table 1 outlines the optimization strategy we use for this kernel. Since only array B has reuse on three dimensions, the loops are tiled on three dimensions for reuse in L1 or L2 cache. Arrays A and B access data in the loop nest in the same order as the dimensionality of the iteration

space, thus the original loop order is best for spatial reuse in cache and TLB. Finally loop J is unrolled for register reuse. Four unbound parameters in the script TI, TJ, TK and UI form a four-dimensional parameter space.

PRO-C took 23 steps (870 unique function evaluations) to converge to $TI = 0$, $TJ = 22$, $TK = 0$ and $UJ = 1$. The results of $TK = 0$ and $TI = 0$ suggest that no tiling is needed for K and I loops. Tiling only the J loop produces the best performance. Also no unroll is performed. We suspect that the native compiler's scalar replacement cannot take advantage of available register reuse across the I dimension so there is little benefit of unrolling J. Figure 9 shows the performance of our code variant. On average, our code variant performs 1.35 times faster than the native Intel compiler.

7 Related Work

There are many research projects working on empirical optimization of linear algebra kernels and domain specific libraries. ATLAS [21] uses the technique to geneate highly optimized BLAS routines. It uses a near-exhaustive orthogonal search (search in one dimension at a time by keeping rest of the parameters fixed). The OSKI (Optimized Sparse Kernel Interface) [20] library provides automatically tuned computational kernels for sparse matrices. FFTW [9] and SPIRAL [22] are domain specific libraries. FFTW combines the static models with empirical search to optimize FFTs. SPIRAL generates empirically tuned Digital Signal Processing (DSP) libraries. Rather than focussing on one particular domain, our framework aims at providing a general-purpose compiler based approach tuning code.

Recently, many research projects on compiler transformation frameworks have focussed on *facilitating* the exploration of a large optimization space of possible compiler transformations and their parameter values. TLOG [13] is a code generator for parameterized tiled loops where tile sizes are symbolic parameters. Symbolic tile-size enables static or run-time tile size optimization without repeatedly generating the code and recompiling it for each tile size. POET [23] is a transformation scripting language embedded in an arbitrary programming language. It is interpreted by a POET compiler to apply source-to-source code transformations. Interactive Compilation Interface (ICI) [10] provides a flexible and portable interface to internal compiler optimizations so that iterative optimization [1] can be applied at the loop or instruction-level by adjusting optimization decisions externally. WRaP-IT [11] and Petit [12] are both polyhedral loop transformation framework that supports composition of transformations. They support many high-level loop transformations on perfect loop nests in a single transformation step and by composing many low-level transformations on each individual loop, they also support arbitrary loop transformations on imperfect loop nests. LeTSeE [17] is an iteration optimization tool based on the polyhedral model. It finds all legal affine scheduling of a loop nest and explores this space to find the best scheduling and parameter values. Pluto [4] is an automatic parallelization and locality optimization tool also based on the polyhedral model.

There is also some work done in using search techniques to explore compiler generated parameter spaces. Kisuki et al [14] addresses the problem of selecting tile sizes and unroll factors simultaneously. Different search algorithms are used to search the parameter space - Genetic algorithms, Simulated Annealing, Pyramid search, Window search and Random search. Qasem et al [18] use a modified version of pattern-based direct search algorithm to explore the same search space. Our work considers a much broader range of loop transformations. Also Kisuki et al. report converging to a solution in hundreds of iterations. By effectively utilizing the underlying parallel infrastructure, we converge to solutions in a few tens of iterations.

8 Conclusion

In this paper, we integrated the capabilities of Active Harmony and CHiLL to create a unique and powerful framework that is capable of both fully automated code transformation and parameter search as well as user assisted transformation combined with automatic parameter search. The resulting framework employs a parallel search technique to simultaneously evaluate different combinations of compiler optimizations. Our system is demonstrated on three computational kernels for automatic compilation and tuning in parallel to achieve performance that greatly exceeds the Intel compiler, and is comparable to (and sometimes exceeds) the near-exhaustive search of the ATLAS library system.

Our work on this topic is just beginning, in the near term we plan to explore optimizing larger programs within our framework. We also plan to combine our current offline optimization approach with online optimization of application parameters.

Acknowledgements. This work was supported in part by DOE grants DE-CFC02-01ER25489, DE-FG02-01ER25510, DE-FC02-06ER25763, DE-FC02-06ER25765 and DE-FG02-08ER25834, by NSF awards EIA-0080206 and CSR-0615412, and by a gift from In-

tel Corporation.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2004.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[3] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, June 1997.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[5] C. Chen. *Model-Guided Empirical Optimization for Memory Hierarchy*. PhD thesis, University of Southern California, 2007.

[6] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical report, University of Southern California, 2008.

[7] C. Chen, J. Chame, and M. W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2005.

[8] I.-H. Chung and J. K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 30, Washington, DC, USA, 2004. IEEE Computer Society.

[9] M. Frigo. A fast Fourier transform compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[10] G. Fursin and A. Cohen. Building a practical iterative compiler. In *Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART'09)*, Jan. 2007.

[11] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[12] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, University of Maryland at College Park, Mar. 1995.

[13] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. M. Strout. Multi-level tiling: M for the price of one. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[14] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[15] D. M. Mount. `http://www.cs.umd.edu/~mount/ANN/`. [last accessed: Feb 09, 2009].

[16] Y. Nelson, B. Bansal, M. Hall, A. Nakano, and K. Lerman. Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[17] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

[18] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *J. Supercomput.*, 36(2):183–196, 2006.

[19] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel parameter tuning for applications with performance variability. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 57, Washington, DC, USA, 2005. IEEE Computer Society.

[20] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16:521–530, June 2005.

[21] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of Supercomputing '98*, Nov. 1998.

[22] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[23] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[24] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):358–386, Feb. 2005.