A Comparison of Search Heuristics for Empirical Code Optimization

Keith Seymour [†1], Haihang You [†2], Jack Dongarra [†*‡3]

[1] seymour@eecs.utk.edu [2] you@eecs.utk.edu [3] dongarra@eecs.utk.edu

[†] *Electrical Engineering and Computer Science Department*
University of Tennessee
Knoxville, TN USA

[*] *Oak Ridge National Laboratory*
Oak Ridge, TN USA

[‡] *University of Manchester*
Manchester, M13 9PL, UK

***Abstract*—This paper describes the application of various search techniques to the problem of automatic empirical code optimization. The search process is a critical aspect of auto-tuning systems because the large size of the search space and the cost of evaluating the candidate implementations makes it infeasible to find the true optimum point by brute force. We evaluate the effectiveness of Nelder-Mead Simplex, Genetic Algorithms, Simulated Annealing, Particle Swarm Optimization, Orthogonal search, and Random search in terms of the performance of the best candidate found under varying time limits.**

I. Introduction

Modern CPU design has been driven by a balance of many factors, such as cost, power consumption, heat, and performance, which leads to many slight differences in their characteristics – clock speed, the number of cores per chip, existence of hyper-threading, cache size and associativity, number of functional units, latencies, etc. Traditionally, to achieve the best performance, the developer had to hand-tune the code with these characteristics in mind and would have to repeat the process for each target architecture. The hand-tuning process is very time consuming, often non-portable, and requires the kind of expertise that only a limited number of programmers possess. Meanwhile, the compiler community has developed optimization techniques to transform programs written in high-level languages to run efficiently on these modern architectures [1], [2]. Some of these program transformations include loop blocking[3], [4], loop unrolling[1], loop permutation, fusion and distribution[5], [6]. To select parameters for transformations such as blocking and unrolling, most compilers use analytical models. This is commonly referred to as model-driven optimization. While compiler optimizations are certainly beneficial (and require essentially no effort from the user), the compiler models may not be accurate or up-to-date with the newest hardware, leading to code that does not achieve peak performance. In contrast with the model-driven approach, empirical optimization techniques generate a large number of code variants with different parameter values for a given algorithm, for example matrix multiplication. All these candidates are run on the target machine and the one that gives the best performance is picked. This helps to cope with differences in CPU characteristics by adapting the tuning to the results obtained. To target a new architecture, the tuning process is simply performed on that machine. With this empirical optimization approach ATLAS[7], [8], PHiPAC[9], and FFTW[10] successfully generate highly optimized libraries for dense, sparse linear algebra kernels, and FFT respectively.

One requirement of empirical optimization methodologies is an appropriate search heuristic, which automates the search for the optimal implementation [7], [8]. Theoretically the search space could be infinite, but in practice it can be limited based on specific information about the hardware for which the software is being tuned. For example, ATLAS bounds NB (blocking size) such that $16 \leq NB \leq min(\sqrt{L1}, 80)$, where L1 represents the L1 cache size, detected by a micro-benchmark. Usually the bounded search space is still very large and it grows exponentially as the number of dimensions in the search space increases. In order to find optimal cases quickly, certain search heuristics need to be employed. The goal of our research is to provide a general search infrastructure and heuristics that can be applied to many empirical optimization tasks. In this paper, we present the results of applying several such search techniques to the empirical optimization of two dense linear algebra routines using four different search spaces.

II. Empirical Tuning Infrastructure

Current empirical optimization techniques such as ATLAS and FFTW can achieve good performance in part because the algorithms to be optimized are known ahead of time, so problem-specific techniques can be applied. In our research, we would like to address this limitation by applying the techniques used in ATLAS to the optimization of arbitrary code. Since the algorithm to be optimized is not known in advance, it will require compiler technology to analyze the source code and generate the candidate implementations. The ROSE project[11], [12] from Lawrence Livermore National

Laboratory provides, among other things, a source-to-source code transformation tool (LoopProcessor) that can produce blocked and unrolled versions of arbitrary C input code. The POET project [13] provides similar functionality except that it takes as input a specification of the valid transformations and allows generating parameterized source code as output. Since the input specification already indicates the valid transformations, there is no analysis overhead, thus the code generation is much faster than with the ROSE LoopProcessor. For that reason, the results in this paper were obtained using POET, but the generated code is essentially the same.

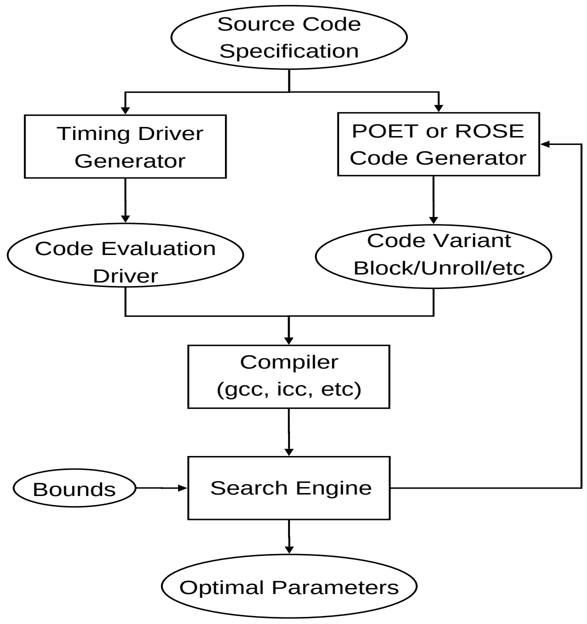

Fig. 1. GCO Framework

Combined with our search techniques and other infrastructure, we can use POET (or any other parameterized code generator) to perform empirical code optimization. To perform the evaluations, we have developed a test infrastructure that automatically generates a timing driver for the optimized routine based on a simple description of the arguments. We refer to the overall system formed by combining of all these parts *Generic Code Optimization* (GCO). As illustrated in Figure 1, the code is first fed into POET or the ROSE LoopProcessor for optimization and separately fed into the timing driver generator which generates the code that actually runs the optimized code variant to determine its performance. The performance results are then fed back into the search engine. Based on these results, the search engine will adjust the parameters used to generate the next code variant. The initial set of parameters could be estimated based on the characteristics of the hardware (e.g. cache size) or could be selected based on the rules of the search technique being used. The search engine also decides when to stop the search process, whether dictated by the user (time limits) or dictated by the results (lack of improvement).

III. Search Space

Before introducing the search techniques, we discuss the code to be tuned and the search spaces involved. The two routines we will optimize are matrix-matrix multiplication and matrix-vector multiplication (both dense). We start with a naïve C implementation, which can be fed directly into the ROSE LoopProcessor or converted by hand to a POET specification. Either way, the result will be a transformed C implementation, which is compiled with the user's choice of compiler.

TABLE I
Summary of the Search Spaces

Code	Dimension	Bounds
Matrix-matrix	i, j, and k loop blocking	2 - 128
	Unroll Amount	2 - 128
Matrix-matrix	i loop blocking	2 - 128
	j loop blocking	2 - 128
	k loop blocking	2 - 128
	Unroll Amount	2 - 128
	Loop order	1 - 6
Matrix-matrix	i loop blocking	2 - 128
	j loop blocking	2 - 128
	k loop blocking	2 - 128
	Unroll Amount	2 - 128
	Loop order	1 - 6
	Compiler flags	1 - 3
	Compiler flags	1 - 4
	Compiler flags	1 - 3
	Compiler flags	1 - 3
	Compiler flags	1 - 2
	Compiler flags	1 - 2
	Compiler flags	1 - 2
	Compiler flags	1 - 2
	Compiler flags	1 - 2
	Compiler flags	1 - 2
	Compiler flags	1 - 2
	Compiler flags	1 - 2
	Compiler flags	1 - 2
	Compiler flags	1 - 2
	Compiler flags	1 - 2
	Compiler flags	1 - 2
Matrix-vector	i loop blocking	2 - 128
	j loop blocking	2 - 128
	Unroll Amount	2 - 128
	Loop order	1 - 2

Given the same code, we can define the search space in different ways. See Table I for a summary of the search spaces. For the matrix multiplication case, we have defined three search spaces. The first search space has two dimensions: blocking and unrolling. Having a single blocking dimension means that all loops are blocked at the same amount. The next search space includes separate blocking dimensions, unrolling, and loop order. The loop order dimension represents the possible reorderings of the loops (e.g. 1 = ijk, 2 = ikj, etc.). The last search space for the matrix multiplication case includes 16 dimensions of compiler flags. For the matrix-vector multiplication case, we have defined one search space similar to the matrix-matrix case. The bounds of each dimension are sometimes rather arbitrary, but in other cases, the bounds are dictated by the capabilities of the code generator or the characteristics of the code being tuned.

The two dimensional search space for the matrix-matrix case was chosen to be feasible to exhaustively search so that we can compare the results of the search techniques with the real optimum point within the search space. The platform used for the experiments is a 2.66GHz Intel Xeon X5355 running Fedora Core 6 (kernel 2.6.22-10-perfctr), PAPI 3.6.0, and using gcc 4.1.2 for all compilations. Although the CPU is quad-core, we are only tuning single-threaded performance.

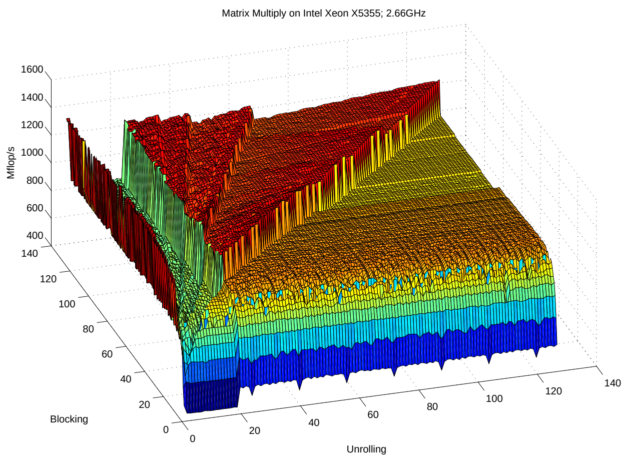

Fig. 2. Two-dimensional Search Space of Matrix Multiply

Figure 2 shows the results of running an exhaustive search over both dimensions of our search space (block sizes up to 128 and unrolling up to 128). The x and y axes represent block size and unrolling amount, while the z axis represents the performance in Mflop/s of the generated code. The code being optimized is an implementation of square matrix-matrix multiplication (N=400). The dimension size is relatively small to allow faster evaluations at each point while being large enough to ensure repeatable timings. In general, we see the best results along the blocking axis with a low unrolling amount as well as along the diagonal where blocking and unrolling are equal, but there are also peaks along areas where the block size is evenly divisible by the unrolling amount. The best performance was found with block size 80 and unrolling amount 2. This code variant ran at 1459 Mflop/s compared to 778 Mflop/s for the naïve version compiled with gcc.

Examining the plot, we notice two obvious characteristics. First, the triangular area on the right is typically low and flat. This area represents all the points where the unrolling amount is larger than the block size. A possible reason the performance is relatively low in this area is because the unrolled portion will not be used when the unroll amount is larger than the block size. It falls through to the clean-up loop, which is not unrolled at all in the source. The second glaring characteristic of the plot is the trough running along the blocking dimension. To investigate this, we picked one block size (80) and measured various CPU performance counters at different unrolling amounts. We found two events that had a correlation with the drops in performance. First, Figure 3 shows the number of branch mispredictions. As previously mentioned, when the unroll amount becomes larger than the block size, it falls into the clean-up loop, which corresponds with the drop in performance at unroll amount 81, but it does not correspond with the area of low performance from unroll amounts 3 to 20. However, we found a second event that does correspond with that area. Figure 4 shows the number of times the Reservation Station (RS) is full. The RS is responsible for buffering instructions until they can be sent to one of the functional units. So, a high number of RS Full events can signify pipeline stalls due to cache misses or due to poor instruction scheduling. Our measurements did not indicate a correlation with cache misses, so the performance drop seems to be a result of the interaction between the way we are transforming the C code and the compiler used to generate the executable. To verify this, we ran the unroll tests with different compilers, as shown in Figure 5. The interesting thing is that gcc 4.3 and icc 9.1 do not exhibit the same drop in performance above unroll amount 80 as gcc 4.1 does. Perhaps these compilers are better at optimizing the simple clean-up loop than the big unrolled loops, although for smaller unroll amounts, there is a performance benefit. This illustrates an important point about empirical tuning - as much as you are tuning for the architecture, you are also tuning for the compiler as well.

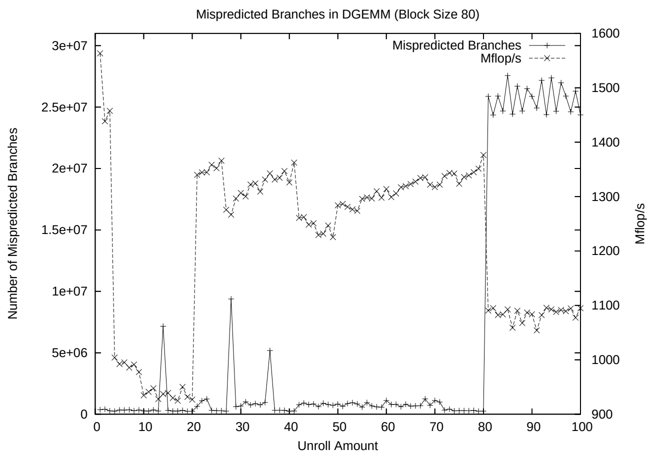

Fig. 3. DGEMM Branch Misses

While the exhaustive search we have done can reveal a lot of interesting things about the search space, it is not usually feasible due to the large amount of time required, especially as new dimensions are added to the search space. Consequently our research involves investigating various search techniques to find an optimal set of parameters without performing an exhaustive search.

IV. Search Techniques

Essentially in GCO, we are trying to solve an optimization problem of the function:

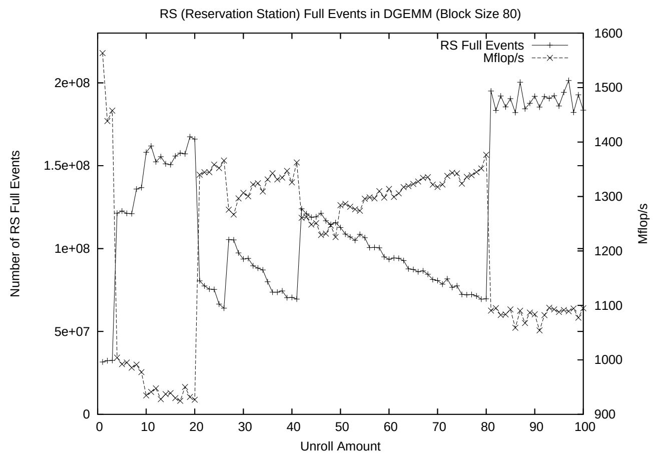

Fig. 4. DGEMM RS Full Events

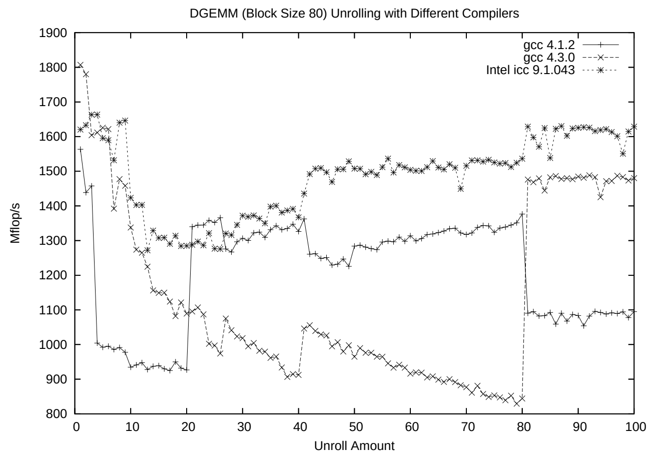

Fig. 5. DGEMM Performance for Different Compilers

$$f(x_1, x_2, \cdots, x_n)$$

The parameters x_1 through x_n represent the code generation options, such as block size and unrolling amount. Typically these are integer values, but in some cases could be real numbers. The value of the function is the performance of the code generated using that set of parameters. Performance can be evaluated in many ways, but the results presented in this paper are based on using PAPI [14] to measure floating point operations per second.

A. Nelder-Mead Simplex Method

Spendley, Hext, and Himsworth [15] introduced the simplex method, which is a non-derivative based direct search method, to solve the minimization problem:

$$min\ f(x)$$

where $f : \mathbf{R}^n \rightarrow \mathbf{R}$, and gradient information is not computationally available. In an n-dimension space $\mathbf{R}$, a simplex is a set of n+1 vertices, thus a triangle in $\mathbf{R}^2$ and a tetrahedron in $\mathbf{R}^3$. The simplex contracts to the minimum by repeatedly comparing function values at n+1 vertices and replacing the vertex with the highest value by reflecting it through the centroid of the rest of the simplex vertices and shrinking. We illustrate the basic idea of the simplex method in Figure 6.

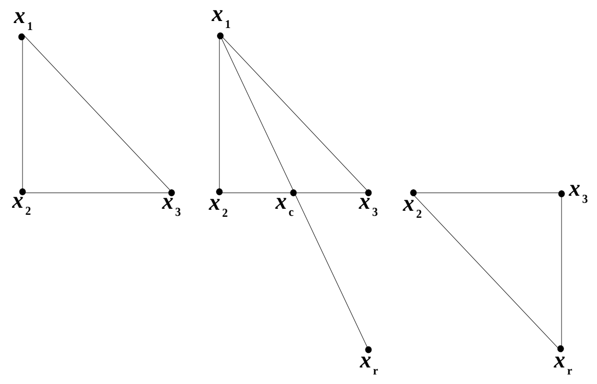

Fig. 6. Original simplex in $\mathbf{R}^2$ where $f(x_1) \geq f(x_2) \geq f(x_3)$; Reflect x_1 through x_c, the centroid of x_2 and x_3, to x_r; The new simplex consists of x_2, x_3 and x_r.

Nelder and Mead improved the method by adding more moves and making the search more robust and faster. We give the description of the Nelder-Mead simplex algorithm [16]:

- Initialize a non-degenerate simplex of n+1 vertices on $\mathbf{R}^n$, compute function value or do a measurement at each vertex, order n+1 vertices by value $f(x_i)$.
- At iteration k, we have:
$$f(x_0^k) \leq f(x_1^k) \leq \cdots \leq f(x_n^k)$$
- Step 1, Calculate centroid:
$$x_c^k = \frac{1}{n}\sum_{i=1}^{n} x_i^k$$
- Step 2, Reflection:
$$x_r^k = x_c^k + \rho(x_c^k - x_n^k), \text{ where } \rho > 0$$
  - If $f(x_0^k) \leq f(x_r^k) < f(x_{n-1}^k)$, replace x_n^k with x_r^k and go to next iteration;
  - Else if $f(x_r^k) < f(x_0^k)$, go to step 3;
  - Else if $f(x_r^k) \geq f(x_{n-1}^k)$, go to step 4.
- Step 3, Expansion:
$$x_e^k = x_c^k + \chi(x_r^k - x_c^k), \text{ where } \chi > 1$$
  - If $f(x_e^k) < f(x_r^k)$, replace x_n^k with x_e^k and go to next iteration;
  - Else replace x_n^k with x_r^k and go to next iteration.
- Step 4, Contraction:
  - If $f(x_r^k) < f(x_n^k)$,
$$x_t^k = x_c^k + \gamma(x_r^k - x_c^k), \text{ where } 0 < \gamma < 1$$
    - If $f(x_t^k) \leq f(x_r^k)$, replace x_n^k with x_t^k and go to next iteration;
    - Else go to step 5.
  - Else
$$x_t^k = x_c^k + \gamma(x_n^k - x_c^k), \text{ where } 0 < \gamma < 1$$
    - If $f(x_t^k) < f(x_n^k)$, replace x_n^k with x_t^k and go to next iteration;
    - Else go to step 5.

- Step 5, Shrink:
  $$x_i^k = x_0^k + \sigma(x_i^k - x_0^k), \text{ where } 0 < \sigma < 1$$

B. Genetic Algorithm

The Genetic Algorithm (GA) is a search method based on the evolutionary process of survival of the fittest. It starts with a population of individuals, each of which is represented by a gene. The gene can be represented as the implementor chooses, but it is typically a bit field, a set of numbers, or a string of characters. In our case, each member of the population is an array of parameters and the fitness of that member is evaluated by measuring the performance of the code generated using those parameters.

The initial population is usually generated randomly, but could be initialized with specific areas of the search space in mind if the problem characteristics are known in advance. In GCO, we start with a random population of 40 candidates. The number is relatively small due to the time required to evaluate each point. If it is too big, we could run out of time before the GA is allowed to evolve much. Given a longer time limit, the population size can be increased.

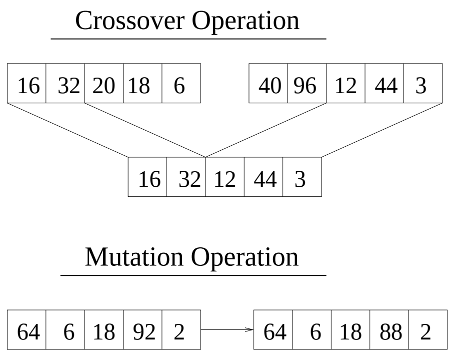

Fig. 7. GA Operations

After the initial population is evaluated, each successive generation is formed by means of several operators.

- The *crossover* operator merges two genes to produce an offspring gene, similar to reproduction in nature. Since the genes are represented as arrays of parameters, we split two genes at a random point in the array and form a new gene from the beginning of one array and the end of the other (see Figure 7).
- The *mutation* operator affects a single member of the population and produces a mutated version of that member. With a low probability, we choose a member and alter one of the parameters slightly (see Figure 7).
- The *selection* operator is responsible for the "survival of the fittest" aspect of the GA. After each generation, it decides which members will be carried over to the next generation. We choose the members with the best performance, excluding the lowest performing members to make room for the new members generated via crossover and mutation.

There is a wide variety of techniques for performing these GA operations. For example, [17] lists fifteen alternative crossover and eight different mutation operations. For a given application, the GA may perform very differently with different choices of operators and values for the crossover rate, mutation rate, initial population, and selection rate.

C. Simulated Annealing

Simulated annealing [18] is a search heuristic based on the annealing of substances such as metals. Since its introduction, it has been used in a variety of applications, including circuit layout and classical optimization problems like traveling salesman. In the annealing process, the substance undergoes a series of heating and cooling cycles to alter its internal structure, which results in a change in the characteristics of the material (e.g. making it less brittle). In simulated annealing, the heating and cooling phases are simulated by means of a global "temperature" setting. When the temperature is high, the system is more likely to allow a change in state, similar to the movement of atoms in the heated material. As it gradually cools, the system should converge to a state of minimum energy. In terms of GCO, that means high temperatures encourage moving to a new part of the search space and choosing new code generation parameters. During the cooling phase, it should become less likely to move, thus converging on the code variant with the best performance.

As with the genetic algorithm, simulated annealing has a number of configurable aspects, which can affect the performance of the search process.

- The *annealing schedule* determines how the temperature is adjusted throughout the simulation. In [18], it is described as a process of moving from a melted state towards a freezing state such that each intermediate temperature is held long enough for the state to reach equilibrium. For GCO, we implement a straightforward annealing schedule by reducing the temperature as the search time limit decreases.
- The *acceptance probability* is a function that determines the probability of accepting a new configuration. In GCO, rather than using a probabilistic function, we use a threshold function.
- The *neighbor selection* function chooses the next potential configuration. In GCO, we choose neighbors somewhat randomly, except that as the temperature decreases, the choices will be closer to the current point.

D. Particle Swarm Optimization

Particle Swarm Optimization has its origins in the simulation of social behavior, particularly the flocking or swarming patterns of birds [19]. For example, it appeared that through some form of social cooperation, many birds were able to flock together and converge on a food source even with no prior knowledge of its location. As applied to the optimization of a function, PSO consists of a population of particles flying through hyperspace, each one represented by a position and velocity. The particle retains a memory of the best position it has visited, but it is also aware of the best position found by neighboring particles and the best position found by any other particle (i.e. the current global best). The particles are simultaneously drawn towards the global and local best points based on some magic numbers that define the relative strength of the different attractions.

In GCO, we define the initial population randomly, with the number of members being based on an estimate of the number of evaluations that can be done before the time limit is exceeded. If the population is too large for the time limit, the particles will not have a chance to move around much. In our implementation, we do not have multiple neighborhoods – the whole population is essentially one neighborhood. At each iteration, the performance of every particle is evaluated and the velocities are updated based on the results according to the following formula.

$$\begin{aligned} v_{i+1} &= w \cdot v_i + c_1 \cdot r_1 \cdot (BestPt_{local} - Pt_i) \\ &\quad + c_2 \cdot r_2 \cdot (BestPt_{global} - Pt_i) \\ Pt_{i+1} &= Pt_i + v_{i+1} \end{aligned}$$

Where v_i is the velocity of particle i, $BestPt_{local}$ is the best point seen by particle i, $BestPt_{global}$ is the best point seen by any other particle, r_1 and r_2 are random numbers uniformly distributed between $[0.0, 1.0)$, w is a weight to influence the importance of the previous velocity, and c_1 and c_2 are weights to influence the importance of the best point seen by the particle versus the best point seen globally.

Although we did not experiment with tweaking the magic numbers, we did have to try a few techniques for handling points that stray out of bounds. There are several techniques described in [20]. You could let the point stay out of bounds, but give the resulting performance an artificially bad value (such as infinity) or simply place it back in the search space randomly. We experimented with placing the point on the boundary where it exited the search space, but they tended to get stuck, so we ended up making them "bounce" a bit off the bound back into the search space.

E. Orthogonal

In orthogonal search, one dimension is optimized while keeping the other dimensions constant. Then each successive dimension is optimized while retaining the best values for the preceding dimensions. Naturally the main problem is determining the order in which the dimensions are optimized. In some systems such as ATLAS, the meaning of each dimension is known ahead of time, so the order can be based on experience and reasoning about the interactions between the transformations. The GCO search engine does not know what the dimensions represent, so it just optimizes them in the order specified by the user. If the time limit has not been exceeded, the search can start over at the first dimension using all the previous best values. If the search converges on one set of parameters and there is still time remaining, it starts a new iteration with randomly chosen parameters. If there is not enough time to complete an entire iteration of the orthogonal search (i.e. one pass through each dimension), the search engine reverts to random search. The determination is made by estimating the average time required per evaluation.

F. Random

In random search, each point is chosen completely at random and its performance has no impact on subsequent selections. Random search is used as a baseline to which we compare the other search techniques. To be considered effective, a search heuristic should be able to do better than a random number generator.

V. Experimental Results

To evaluate the various search techniques, we performed searches of the spaces described in Section III. Unless otherwise specified, the experiments were performed on a 2.66GHz Intel Xeon X5355 running Fedora Core 6 (kernel 2.6.22-10-perfctr), gcc 4.1.2, and PAPI 3.6.0. Each point on the graphs presented here represents the average of 10 runs of each search technique within the specified run-time constraint (either limited by time or by the number of evaluations to be performed).

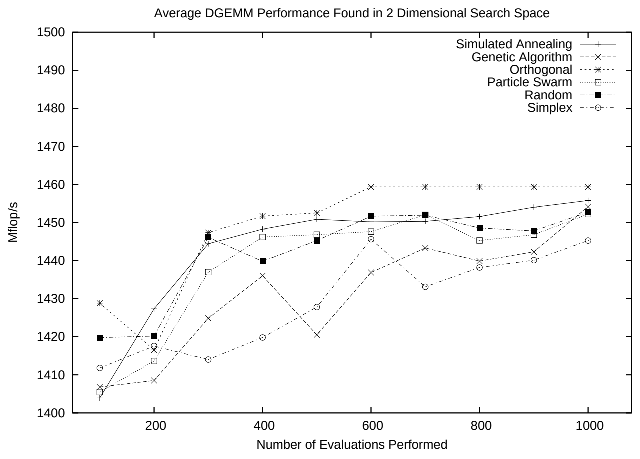

Fig. 8. Average DGEMM Performance Found in 2 Dimensional Search Space

First we examine the results of tuning matrix multiply in the two dimensional search space. The methodology is a bit different in this case compared to the others that will be presented. Since we had already performed an exhaustive search of this space, we decided to run the searches on the data that had already been collected. This would allow comparing the best case found by the search to the absolute best point in the search space, which turned out to be 1459 Mflop/s. Looking at Figure 8, above 600 evaluations, the orthogonal search consistently hits the maximum point. The others are a bit worse, but only by a few percent. Even random search does quite well. We speculated that random performs well because there are a lot of points that perform fairly close to the global maximum. To verify this, we plotted a histogram of the performance of all the points in the two dimensional search space, shown in Figure 9. There are 182 points out of the total 16129 points which have a performance within 5% of the maximum. So if we think of it as a binomial experiment, the probability of finding at least one point within 5% of the maximum by doing 100 evaluations would be around 68%, which closely matches our observations (7 of the 10 runs found points within 5% of the maximum). By the time we reach 500 evaluations, the probability is over 99%. Therefore, it appears that this two dimensional search space is a relatively easy one to search and even a random search is likely to give good results.

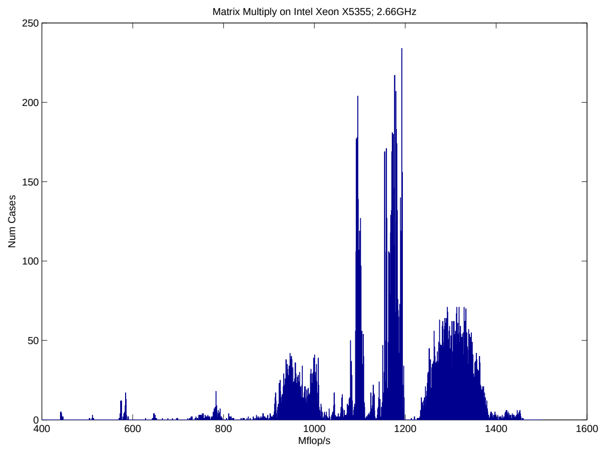

Fig. 9. Histogram of Matrix Multiply Performance (Exhaustive Search)

The next case we will examine is the five dimensional search space for matrix multiply. As summarized in Table I, this search space has separate blocking dimensions, an unrolling dimension, and a loop order dimension. There is very little separation between the various search techniques in this case, as shown in Figure 10. Only Particle Swarm Optimization appears to have a consistent advantage, but even that is quite small (on the order of a few percent). We do not have a visualization of this search space since it is not feasible to exhaustively search it, but one possible reason PSO does a bit better here is because the swarming activity tends to produce a better local search around the maximum points.

The four dimensional matrix-vector multiply case is very similar to the five dimensional matrix multiply case just discussed. As Figure 11 shows, PSO does well and most of the others are grouped together. The main difference in this case is the poor performance of the orthogonal search. It starts off doing well because for short time limits, it estimates that a full orthogonal search is not possible, so it reverts to random search. Thus for the 1 and 5 minute searches, the results are almost identical to the random search results. At 15 minutes and above, there is enough time to do at least one iteration of the orthogonal search, but it picks a bad value for the last dimension, which only has two values (representing the loop order). It starts the search with the last dimension set to the lower bound and optimizes the first three dimensions based on that. So, by the time it reaches the last dimension, the previous dimensions have been optimized to work well with the current

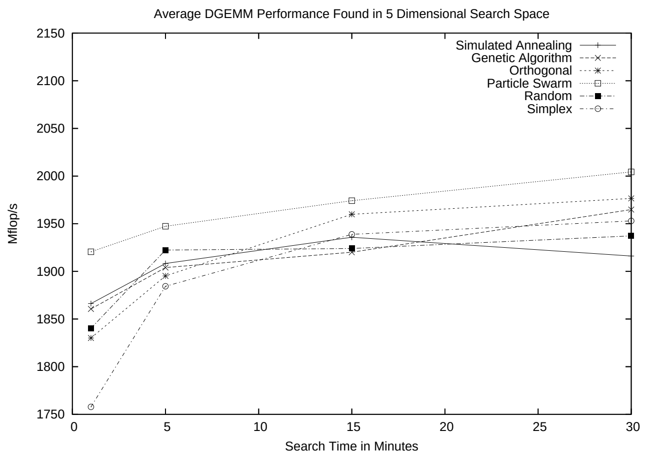

Fig. 10. Average DGEMM Performance Found in 5 Dimensional Search Space

setting, so changing it does not improve the performance. Even when more iterations can be done, it does not improve much because it retains the bad value picked in the first iteration. To measure the effect of the dimension ordering, we ran a quick experiment in which we sorted the dimensions based on size so that the loop order dimension came first. Running the orthogonal search again with a time limit of 15 minutes produced an average performance of 1000.2 Mflop/s versus 774.7 Mflop/s for the unordered dimensions.

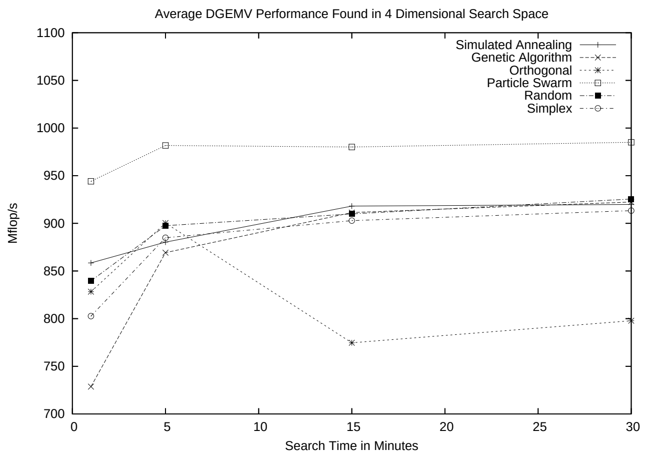

Fig. 11. Average DGEMV Performance Found in 4 Dimensional Search Space

Since we were not getting much separation between the various search techniques, we decided to try to vastly increase the search space to make it more difficult. We had already used all the available code transformation options, so we added 16 dimensions of compiler optimization flags to the mix. Most of these dimensions are small – either the feature is enabled or disabled – but some of them have a range of values (e.g. optimization level 0, 1, 2, 3). From Figure 12, we can see that we did not get the separation we were hoping for. The orthogonal search still shows some suffering because of the small discrete dimensions at the end of the search space.

Simplex can be a little erratic for low time limits (especially when the number of dimensions is high) because it may not be able to complete at least one full iteration of the Simplex algorithm within the allotted time. Unlike orthogonal search, we do not have an easy prediction of the number of evaluations that Simplex will need to do, so we cannot determine when it would be beneficial to revert to random search for low time limits. After some experimentation, it turns out that most of the flags have very little effect on the performance, so that is probably why the search techniques are grouped together once again. The promising aspect of this experiment is that the performance is up to almost 2400 Mflop/s with the optimization flags, versus 2000 Mflop/s without them (the previous experiments were all done using only `-O3`). Most of that gain came from the `-funroll-loops` flag, but it is interesting to note that the selection of compiler flags also affects the selection of the best code generation parameters, so we found that it is useful to tune them in conjunction with each other.

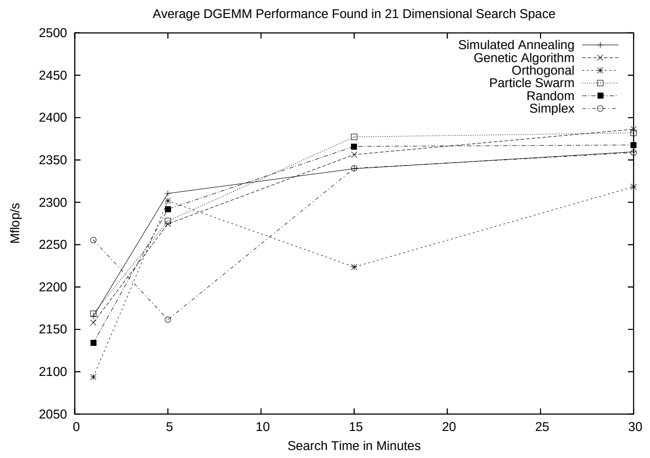

Fig. 12. Average DGEMM Performance Found in 21 Dimensional Search Space

VI. Related Work

There are several projects that adopt an automatic performance tuning strategy to produce highly optimized libraries, but with different approaches to the heuristic search. PHiPAC[9] is a methodology for developing High-Performance linear algebra libraries in ANSI C. It searches for the optimal block sizes starting from register level (L0 cache), then L1 cache, L2 cache, and so on. A random search strategy is used for searching the L0 search space and a heuristic-based search is used for the other levels. ATLAS[7], [8] is an empirical tuning system which generates an optimized BLAS library. ATLAS first bounds the search space based on hardware information detected by microbenchmarks. It then uses an orthogonal search, which starts with an initial set of parameters and searches for the optimal value for one parameter at a time and keeps the rest unchanged. After each one-dimensional linear search, the selected parameter value will be preserved. FFTW[10] generates a highly optimized library for computing the discrete Fourier transform (DFT). Its search strategy is called dynamic programming, which takes advantage of the recursive nature of the problem and solutions of smaller problems can be used to construct solutions of larger problems. SPIRAL[21] generates highly optimized code for a broad set of digital signal processing transforms. It uses dynamic programming primarily, but when that fails, it has several other methods to fall back on (e.g. genetic algorithms and random search). A genetic algorithm approach has also been used for selecting the best sequence of optimizations applied within a traditional compiler [22] and for optimizing the set of flags to specify [23].

Other research has focused on deriving a mathematical model for ATLAS [24] and achieved performance nearly as good as the empirically optimized version on certain platforms. There has also been some research on combining analytic models with an empirical search. In [25], an analytic model of ATLAS was refined with an empirical search, leading to shorter search time than ATLAS with higher performance than the purely analytic approach. Our goal, however, is to develop a generic tuning and search infrastructure that is adaptable to a variety of different applications.

VII. Conclusion

In this paper, we have examined a variety of search heuristics and applied them to several practical empirical tuning tasks. The experiments have demonstrated that for a very modest investment in search time (15-30 minutes), the performance can be more than tripled compared to the naïve C implementation. As the code generators for automatic empirical tuning become more sophisticated and the search spaces consequently grow, having effective search techniques will become increasingly important.

The strength of random search (and our subsequent examination of the distribution of performance in the search space) indicates that the search spaces are not incredibly difficult – there are many points with performance within 5% of the true maximum. Even trying to artificially make the search more difficult by adding many dimensions of compiler flags did not result in a big distinction between the random search and other methods, although as we mentioned, many of those dimensions were probably not very influential. PSO had a clear (though modest) advantage in some of the experiments, possibly because the swarming provides a kind of local search functionality towards the end of the cycle. Another observation is that some of the dimensions have their best performance near the bounds and PSO tends to search these areas well due to the way it handles the positioning of particles that go out of bounds. In other cases, the orthogonal search had an advantage, but it suffered when the dimensions were not ordered well.

We have not spent much time trying to tweak the search strategies, so it is possible that with more effort, we could get more of an improvement over random search. Since the tweaks could work well for certain search spaces, but be worse on others, it would be worthwhile to do more experiments to determine a good set of general parameters that would work well for many tuning problems. Based on some initial experiments, a modified orthogonal search that sorts the dimensions based on size could be a strong technique. We are also interested in looking into hybrid and adaptive techniques. The orthogonal search is a rudimentary example – it can adapt to the search time limit by falling back on random search when it estimates that a full iteration cannot be completed. However, perhaps we could use one search to "prime" another one, switch techniques in mid-stream, or combine search techniques in other ways.

Acknowledgment

This work was funded in part by the Department of Energy and the National Science Foundation.

References

[1] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[2] D. A. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Commun. ACM*, vol. 29, no. 12, pp. 1184–1201, 1986.

[3] Q. Yi, K. Kennedy, H. You, K. Seymour, and J. Dongarra, "Automatic Blocking of QR and LU Factorizations for Locality," in *2nd ACM SIGPLAN Workshop on Memory System Performance (MSP 2004)*, 2004.

[4] R. Schreiber and J. Dongarra, "Automatic Blocking of Nested Loops," Knoxville, TN 37996, USA, Tech. Rep. CS-90-108, 1990. [Online]. Available: citeseer.ist.psu.edu/schreiber90automatic.html

[5] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 424–453, 1996.

[6] U. Banerjee, "A Theory of Loop Permutations," in *Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing*. Pitman Publishing, 1990, pp. 54–74.

[7] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project." *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, January 2001.

[8] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick, "Self Adapting Linear Algebra Algorithms and Software," *Proceedings of the IEEE*, vol. 93, no. 2, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[9] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology," in *International Conference on Supercomputing*, 1997, pp. 340–347. [Online]. Available: citeseer.ist.psu.edu/article/bilmes97optimizing.html

[10] M. Frigo and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," in *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, vol. 3. IEEE, 1998, pp. 1381–1384.

[11] Q. Yi and D. Quinlan, "Applying Loop Optimizations to Object-oriented Abstractions Through General Classification of Array Semantics," in *The 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, Sep 2004.

[12] D. Quinlan, M. Schordan, Q. Yi, and A. Saebjornsen, "Classification and Untilization of Abstractions for Optimization," in *Ths First International Symposium on Leveraging Applications of Formal Methods*, Paphos, Cyprus, Oct 2004.

[13] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "POET: Parameterized Optimizations for Empirical Tuning," in *IEEE Parallel and Distributed Processing Symposium*, Long Beach, CA, Mar 2007.

[14] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *The International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.

[15] W. Spendley, G. Hext, and F. Himsworth, "Sequential Application of Simplex Designs in Optimization and Evolutionary Operation," *Technometrics*, vol. 4, pp. 441–461, 1962.

[16] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder–Mead Simplex Method in Low Dimensions," *SIAM J. on Optimization*, vol. 9, no. 1, pp. 112–147, 1998.

[17] D.J.Stewardson, C.Hicks, P.Pongcharoen, S.Y.Coleman, and P.M.Braiden, "Overcoming Complexity via Statistical Thinking: Optimising Genetic Algorithms for use in Complex Scheduling problems via Designed Experiments," in *Manufacturing Complexity Network Conference*, April 2002.

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[19] J. Kennedy and R. Eberhart, "Particle Swarm Optimization," *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, pp. 1942–1948 vol.4, Nov/Dec 1995.

[20] S. Helwig and R. Wanka, "Particle Swarm Optimization in High-Dimensional Bounded Search Spaces," *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, pp. 198–205, 1-5 April 2007.

[21] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proceedings of the IEEE*, vol. 93, no. 2, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[22] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive Optimizing Compilers for the 21st Century," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 7–22, 2001.

[23] "Acovea – Using Natural Selection to Investigate Software Complexities," http://www.coyotegulch.com/products/acovea/.

[24] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A Comparison of Empirical and Model-driven Optimization," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. ACM Press, 2003, pp. 63–76.

[25] A. Epshteyn, M. J. Garzarán, G. DeJong, D. A. Padua, G. Ren, X. Li, K. Yotov, and K. Pingali, "Analytic Models and Empirical Search: A Hybrid Approach to Code Optimization," in *LCPC*, ser. Lecture Notes in Computer Science, E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, Eds., vol. 4339. Springer, 2005, pp. 259–273.