An Effective Empirical Search Method for Automatic Software Tuning[*]

Haihang You[†] Keith Seymour[†] Jack Dongarra[‡]

January 21, 2005

Abstract

Empirical software optimization and tuning is an active research topic in the high performance computing research community. It is an adaptive system to generate optimized software using empirically searched parameters. Due to the large parameter search space, an appropriate search heuristic is an essential part of the system. This paper describes an effective search method that can be generally applied to empirical optimization. We apply this method to ATLAS (Automatically Tuned Linear Algebra Software), which is a system for empirically optimizing dense linear algebra kernels. Our experiments on four different platforms show that the new search scheme can produce parameters that can lead ATLAS to generate a library with better performance.

1 Introduction

As CPU speeds double every couple of years following Moore's law[1], memory speed lags behind. Because of this increasing gap between the speeds of processors and memory, in order to achieve high performance on modern systems new techniques such as longer pipeline, deeper memory hierarchy, and hyper threading have been introduced into the hardware design. Meanwhile, compiler optimization techniques have been developed to transform programs written in high-level languages to run efficiently on modern architectures[2, 3]. These program transformations include loop blocking[4, 5], loop unrolling[2], loop permutation, fusion and distribution[6, 7]. To select optimal parameters such as block size, unrolling factor, and loop order, most compilers would compute these values with analytical models referred to as model-driven optimization. In contrast, empirical optimization techniques generate a large number of code variants with different parameter values for an algorithm, for example matrix mulplication. All these candidates run on the target machine, and the one that gives the best performance is picked. With this empirical optimization approach ATLAS[8], PHiPAC[9], and FFTW[10] successfully generate highly optimized libraries for dense, sparse linear algebra kernels and FFT respectively. It has been shown that empirical optimization is more effective than model-driven optimization[11].

An appropriate search heuristic is one requirement of empirical optimization methodologies, which automates the search for the most optimal available implementation [8]. Theoretically the search space is 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 dimension of 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 heuristic that can apply to any empirical optimization system. The Nelder-Mead simplex method[12] is a well-known and successful non-derivative direct search method for optimization. We have applied this method to ATLAS, replacing the global search of ATLAS with the simplex method. Experimental results on four different architectures show that the library generated using the simplex search has better performance than the original ATLAS.

This paper is organized as follows. In Sec-

[*]This work supported in part by the NSF under CNS-0325873.

[†]Department of Computer Science, University of Tennessee, Knoxville

[‡]Department of Computer Science, University of Tennessee, Knoxville and Oak Ridge National Laboratory

tion 2, we briefly introduce the Nelder-Mead simplex method. Section 3 describes the implementation of the algorithm including modifications suitable for empirical optimization applications. Experimental results are presented in Section 4. Finally, conclusions are provided in Section 5.

2 Simplex Method

To solve the minimization problem:

$$min\ f(x)$$

Where $f : \mathbf{R}^n \to \mathbf{R}$, and gradient information is not computationally available, Spendley, Hext, and Himsworth[13] introduced the simplex method, which is a non-derivative based direct search method. In an n-dimension space **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 1.

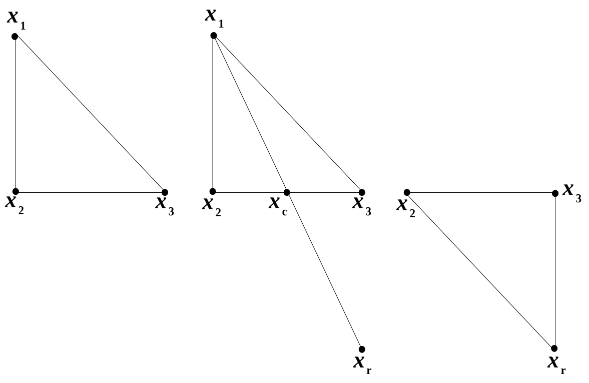

Figure 1: Original simplex in $\mathbf{R}^2$ where $f(x_1) \geq f(x_2) \geq f(x_3)$; Reflect x_1 through x_c, the centroid of x_2 and x_3, to x_r; The new simplex consists of x_2, x_3 and x_r.

Nelder and Mead improved the method by adding more moves and making the search more robust and faster. We give the discription of the Nelder-Mead simplex algorithm [14]:

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

3 Modified Simplex Search Algorithm

Empirical optimization requires a search heuristic for selecting the most highly optimized code from the large number of code variants generated during the search. Because there are a number of different tuning parameters, such as blocking size, unrolling factor and computational latency, the resulting search space is multi-dimensional. The direct search method, namely Nelder-Mead simplex method [12], fits in the role perfectly.

The Nelder-Mead simplex method is a direct search method for minimizing a real-valued function $f(x)$ for $x \in \mathbf{R}^n$. It assumes the function $f(x)$ is continuously differentiable. We modify the search method according to the nature of the empirical optimization technique:

- In a multi-dimensional discrete space, the value of each vertex coordinate is cast from double precision to integer.
- The search space is bounded by setting $f(x) = \infty$ where $x < l$, $x > u$ and l, u, and $x \in \mathbf{R}^n$. The lower and upper bounds are determined based on hardware information.
- The simplex is initialized along the diagonal of the search space. The size of the simplex is chosen randomly.
- User defined restriction conditions: If a point violates the condition, we can simply set $f(x) = \infty$, which saves search time by skipping code generation and execution of this code variant.
- Create a searchable record of previous execution timing at each eligible point. Since execution times would not be identical at the same search point on a real machine, it is very important to be able to retrieve the same function value at the same point. It also saves search time by not having to re-run the code variant for this point.
- As the search can only find the local optimal performance, multiple runs are conducted. In search space of $\mathbf{R}^n$, we start n+1 searches. The initial simplexes are uniformly distributed along the diagonal of the search space. With the initial simplex of the n+1 result vertices of previous searches, we conduct the final search with the simplex method.
- After every search with the simplex method, we apply a local search by comparing performance with neighbor vertices, and if a better one is found the local search continues recursively.

4 Experiments with ATLAS

Figure 2 depicts the structure of ATLAS [8]. By running a set of benchmarks, ATLAS detects hardware information such as L1 cache size, latency for computation scheduling, number of registers and existence of fused floating-point multiply add instruction.

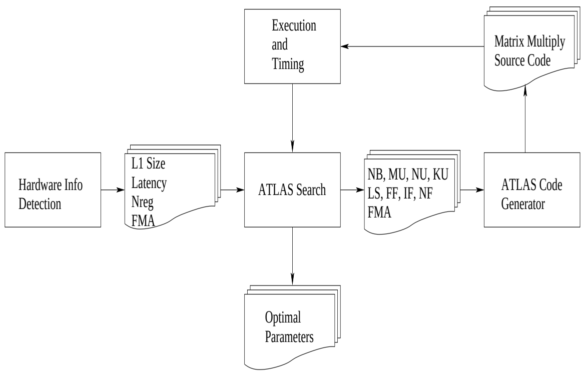

Figure 2: ATLAS with global search

The search heuristics of ATLAS bound the global search of optimal parameters with detected hardware information. For example, NB (blocking size) is one of ATLAS's optimization parameters. ATLAS sets NB's upper bound to be the minimum of 80 and square root of L1 cache size, and lower bound as 16, and NB is a composite of 4. The optimization parameters are generated and fed into the ATLAS code generator, which generates matrix multiply source code. The code is then compiled and executed on the target machine. Performance data is returned to the search manager and compared with previous executions.

ATLAS uses an orthogonal search [11]. For a optimization problem:

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

Parameters x_i (where $1 \leq i \leq n$) are initialized with reference values. From x_1 to x_n, orthogonal search does a linear one-dimension search for the optimal value of x_i, it uses previously found optimal values for $x_1, x_2, \cdots, x_{n-1}$.

We have replaced the ATLAS global search with the modified Nelder-Mead simplex search and conducted experiments on four different architectures: Pentium 4, Itanium 2, Power 4 and Sparc Ultra. The specifications of these four platforms are shown in Table 1.

Given values for a set of parameters, the ATLAS code generator generates a code variant of matrix multiply. The code gets executed with randomly generated 1000x1000 dense matrices as input. After executing the search heuristic, the output is a set of parameters that gives the best performance for that platform. Figure 3 shows the performance of the best matrix multiply code variant selected by each of the two

Feature	Intel Pentium 4	Intel Itanium 2	IBM Power 4	Sun UltraSparc
Processor Speed	2.4GHz	900MHz	1.3GHz	900MHz
L1 Instruction	12KB	16KB	64KB	32KB
L1 Data	8KB	16KB	32KB	64KB
L2	512KB	256KB	1440KB	8MB
L3	N/A	1.5MB	128MB	N/A
FMA	no	yes	yes	no
OS	Linux	Linux	AIX 5.1	SunOS 5.9
Compiler	gcc 3.3.3	icc 8	xlc 6	gcc 3.2

Table 1: Processor Specifications

search methods on four different platforms. Figure 4 compares the total time spent by each of the search methods on the search itself. We can see that the simplex method can find parameters with better performance. Figures 5 through 8 show the performance of matrix multiply generated using the Simplex search compared with the original ATLAS search.

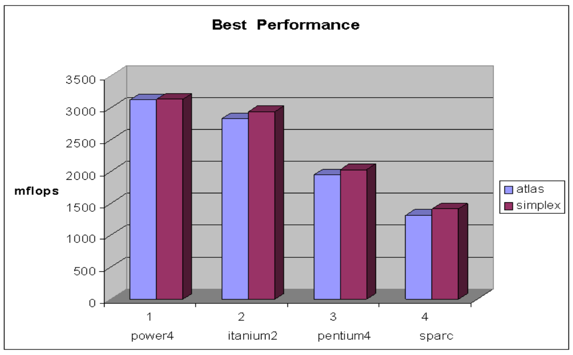

Figure 3: Best performance with input 1000x1000 matrices

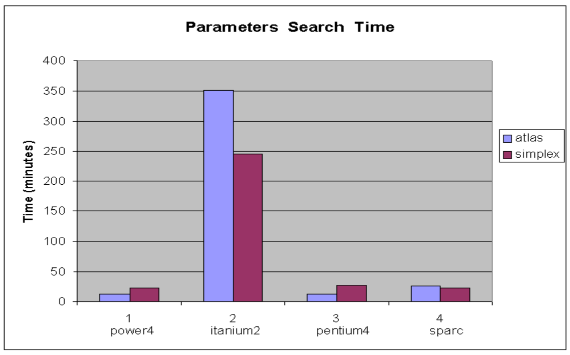

Figure 4: search time

5 Conclusion

Empirical optimization has been shown to be an effective technique for optimizing code for a particular platform. The search heuristic plays an important role in the system. All existing software such as ATLAS [8], PHiPAC [9], and FFTW [10] each has its own search heuristic developed. Our research provides a generic way to search for the optimal parameters. Our research can be extended to GA (Genetic Algorithms) and Direct Search Methods such as pattern search methods, simplex methods and methods with adaptive sets of search directions [15].

References

[1] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 19 April 1965.

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[3] David A. Padua and Michael Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[4] Qing Yi, Ken Kennedy, Haihang You, Keith Seymour, and Jack Dongarra. Automatic Blocking of QR and LU Factorizations for Locality. In *2nd ACM SIGPLAN Workshop on Memory System Performance (MSP 2004)*, 2004.

[5] Robert Schreiber and Jack Dongarra. Automatic Blocking of Nested Loops. Technical Report CS-90-108, Knoxville, TN 37996, USA, 1990.

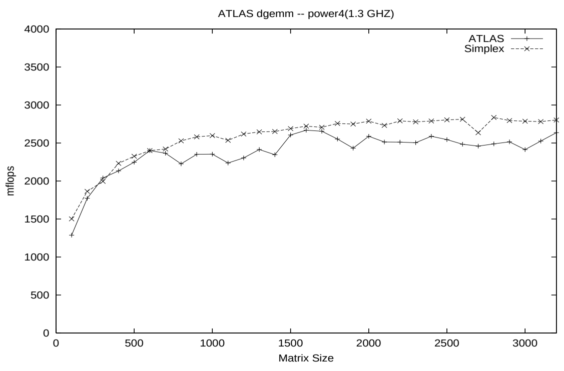

Figure 5: dgemm on Power 4

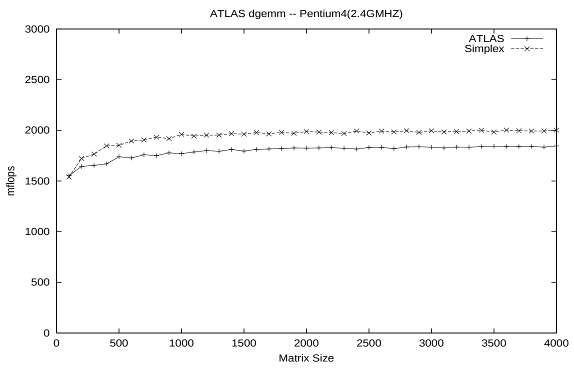

Figure 7: dgemm on Pentium 4

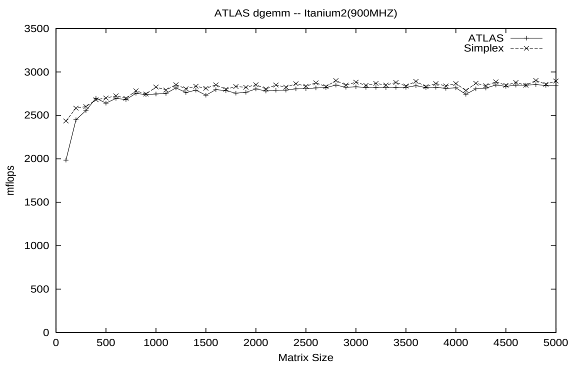

Figure 6: dgemm on Itanium 2

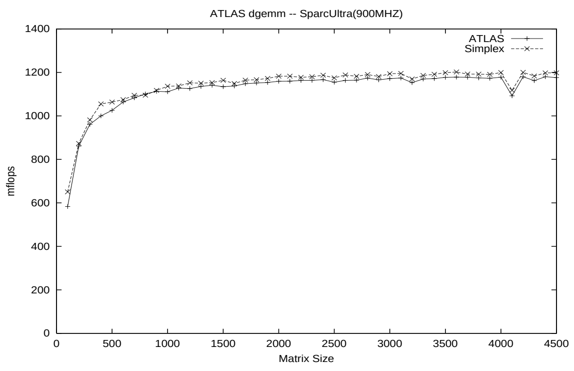

Figure 8: dgemm on Sparc

[6] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, 1996.

[7] Utpal Banerjee. A Theory of Loop Permutations. In *Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing*, pages 54–74. Pitman Publishing, 1990.

[8] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *SC '98: Proceedings of the Proceedings of the IEEE/ACM SC98 Conference*, page 38. IEEE Computer Society, 1998.

[9] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[10] Matteo Frigo and Steven G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[11] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A Comparison of Empirical and Model-driven Optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 63–76. ACM Press, 2003.

[12] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 8:308–313, 1965.

[13] W. Spendley, G.R. Hext, and F.R. Himsworth. Sequential Application of Simplex Designs in Optimization and Evolutionary Operation. *Technometrics*, 4:441–461, 1962.

[14] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence Properties of the Nelder–Mead Simplex Method in Low Dimensions. *SIAM J. on Optimization*, 9(1):112–147, 1998.

[15] Robert Michael Lewis, Virginia Torczon, and Michael W. Trosset. Direct Search Methods: Then and Now. *J. Comput. Appl. Math.*, 124(1-2):191–207, 2000.