An Effective Empirical Search Method for Automatic Software Tuning*

Haihang You† Keith Seymour† Jack Dongarra‡

May 10, 2006

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, 9], PHiPAC[10], and FFTW[11] 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[12].

One requirement of empirical optimization methodologies is an appropriate search heuristic, which automates the search for the most optimal available implementation [8, 9]. 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 method that can apply to any empirical optimization system. The Nelder-Mead simplex method[13] is a well-known and successful nonderivative direct search method for optimization. The Genetic Algorithm [14] is another well-known technique for optimization. We have applied these methods to ATLAS, replacing the global search of ATLAS with the simplex method and the GA. This paper will show experimental results on four different architectures to compare these search techniques with the original ATLAS search both in terms of the perfor-

*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

mance of the resulting library and the time required to perform the search.

This paper is organized as follows. In Section 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. In Section 5, we describe the generic code optimization system and experimental results. Related work is presented in Section 6. Finally, conclusions are provided in Section 7.

2 Simplex Method

To solve the minimization problem:

$$min\ f(x)$$

Where $f : \mathbf{R}^n \rightarrow \mathbf{R}$, and gradient information is not computationally available, Spendley, Hext, and Himsworth[15] 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.

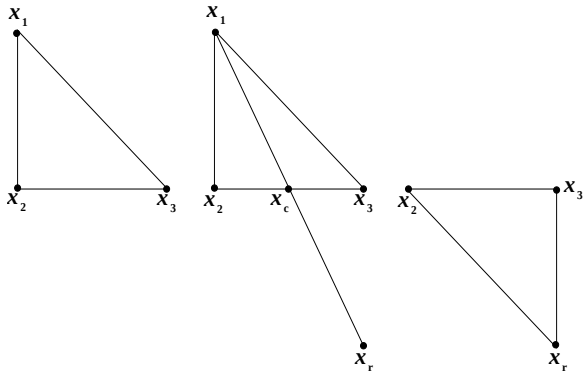

Figure 1: Original simplex in $\mathbf{R}^2$ where $f(x_1) \geq f(x_2) \geq f(x_3)$; Reflect x_1 through x_c, the centroid of x_2 and x_3, to x_r; The new simplex consists of x_2, x_3 and x_r.

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

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 [13], 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

In this section, we briefly describe the structure of ATLAS and then compare the effectiveness of its search technique to the simplex and GA methods.

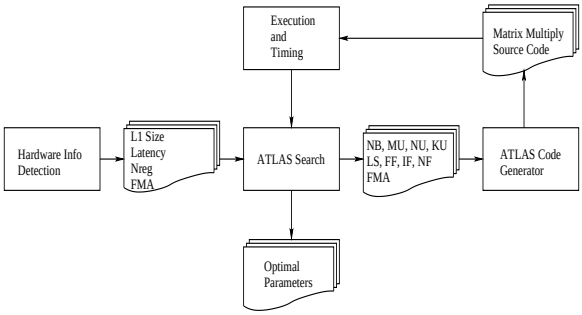

Figure 2: ATLAS with global search

4.1 Structure of ATLAS

Figure 2 depicts the structure of ATLAS [8, 9]. 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. 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 [12]. For an 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-dimensional search for the optimal value of x_i and it uses previously found optimal values for $x_1, x_2, \cdots, x_{n-1}$.

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

4.2 Applying Simplex Search to ATLAS

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 search methods on four different platforms. This shows that the simplex method can find parameters with better performance. Figure 4 compares the total time spent by each of the search methods on the search itself. The Itanium2 search time (for all search techniques) is much longer than the other platforms because we are using the Intel compiler, which in our experience takes longer to compile the same piece of code than the compiler used on the other platforms (gcc). Figures 6 through 9 show the comparision of the performance of matrix multiply on different sizes of matrices using the ATLAS libraries generated by the Simplex search and the original ATLAS search.

4.3 Applying GA to ATLAS

GA (Genetic Algorithm)[14] is a heuristic 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 set of numbers or a string of characters. It produces the next generation by means of crossover, mutation, and selection of offspring based on fitness. In our case, each member of the population is a set of parameters and the fitness of that member is evaluated by measuring the performance of the code generated using those parameters. 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 an application, the Genetic Algorithm may perform very differently with different choices of operators and values for the crossover rate, mutation rate, initial population, and selection rate.

Similar to the integration of the simplex search described in section 4.2, we replaced the ATLAS search with the GA search. With the same experimental setup, we gathered performance data and search time on the four platforms. We can see in Figure 4 and 5 that the GA search took more time on all platforms. Furthermore, the performance of the library generated using the GA search was worse for most platforms, as shown in Figure 3 and Figures 6 through 9. As we conducted the GA search experiments, we observed that a simple random search was as effective as the GA search. In our experimentation, we were not able to derive a GA strategy that worked better than simplex or the original ATLAS search, but we cannot say no such strategy exists.

5 Generic Code Optimization

Current empirical optimization techniques such as ATLAS and FFTW can achieve good performance because the algorithms to be optimized are known ahead of time. We are addressing this limitation by extending 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

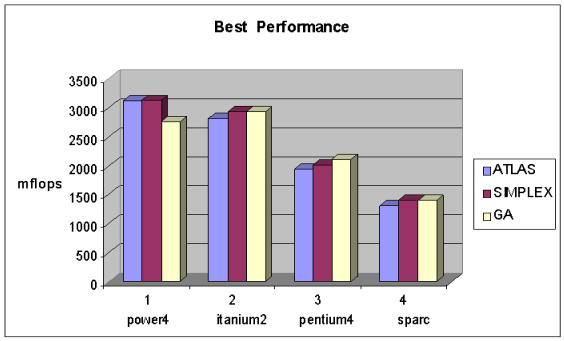

Figure 3: Best performance with input 1000x1000 matrices

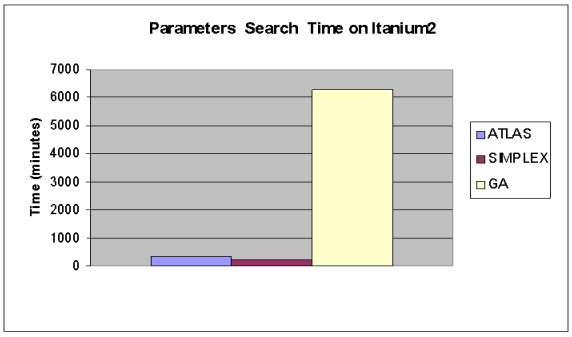

Figure 5: Search time on Itanium2

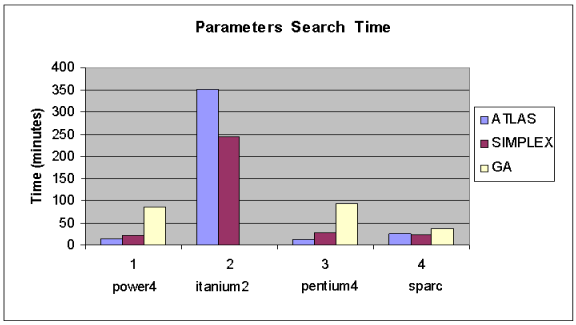

Figure 4: Search time

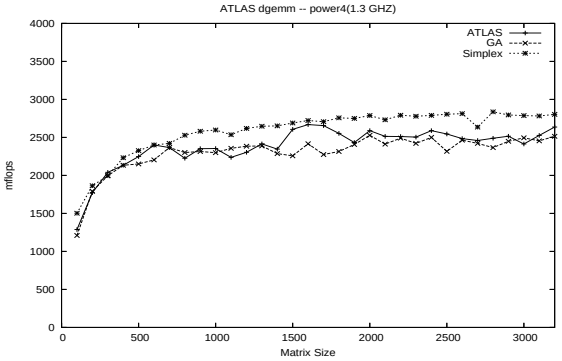

Figure 6: DGEMM on Power 4

project[18, 19] from Lawrence Livermore National Laboratory provides, among other things, a source-to-source code transformation tool that can produce blocked and unrolled versions of the input code. Combined with our search heuristic and hardware information, we can use ROSE to perform empirical code optimization. For example, based on an automatic characterization of the hardware, we will direct their compiler to perform automatic loop blocking at varying sizes, which we can then evaluate to find the best block size for that loop. To perform the evaulations, we have developed a test infrastructure that automatically generates a timing driver for the optimized routine based on a simple description of the arguments.

Figure 10 shows the overall structure of the Generic Code Optimization system. The code is fed into the loop processor for optimization and separately fed into the timing driver generator which generates the code that actually runs the optimized code variant to determine its execution time. The results of the timing are fed back into the search engine. Based on these results, the search engine may adjust the parameters used to generate the next code variant. The initial set of parameters can be estimated based on the characteristics of the hardware (e.g. cache size).

Figure 11 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 code being optimized is a naïve implementation of matrix-vector multiply. In general, we see the best results along the diagonal, but there are also peaks along areas where the block size is evenly divisible by the unrolling amount. The triangular area on the right is typically low because when the unrolling amount is larger than the block size, the unrolled portion will not be used. After running for over 30 hours on a 1.7 GHz Pentium M, the best performance was

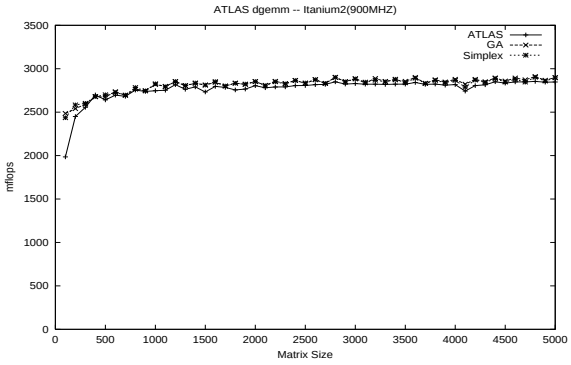

Figure 7: DGEMM on Itanium 2

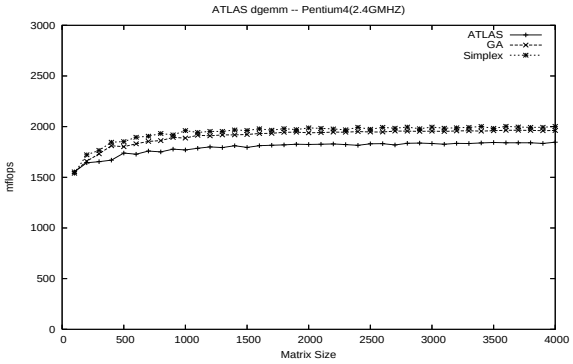

Figure 8: DGEMM on Pentium 4

found with block size 11 and unrolling amount 11. This code variant ran at 338 Mflop/s compared to 231 Mflop/s for the version compiled with gcc.

Of course, due to the large amount of time required, this kind of exhaustive search is not feasible especially as new dimensions are added to the search space. Consequently we are investigating the simplex method as a way to find an optimal set of parameters without performing an exhaustive search. The simplex search technique works the same in this application as it does when applied to ATLAS except in this case we only have two parameters. To evaluate the effectiveness of the simplex search technique, we performed 10,000 runs and compared the results with the true optimum found during the exhaustive search. On average, the simplex technique found code variants that performed at 294 Mflop/s, or 87% of the true optimal performance (338 Mflop/s). At best, the simplex technique can find the true optimum, but that only occurs on 8% of the runs. The worst case was 273 Mflop/s, or 81% of the true optimum. From a

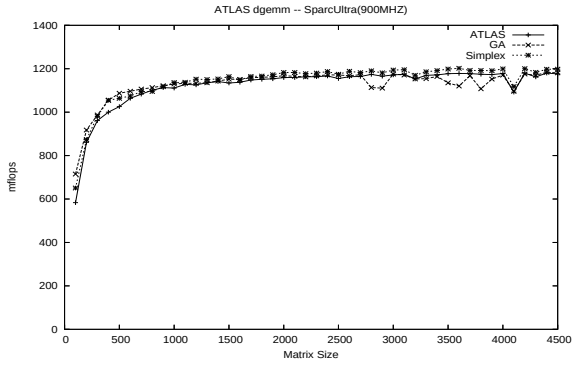

Figure 9: DGEMM on Sparc

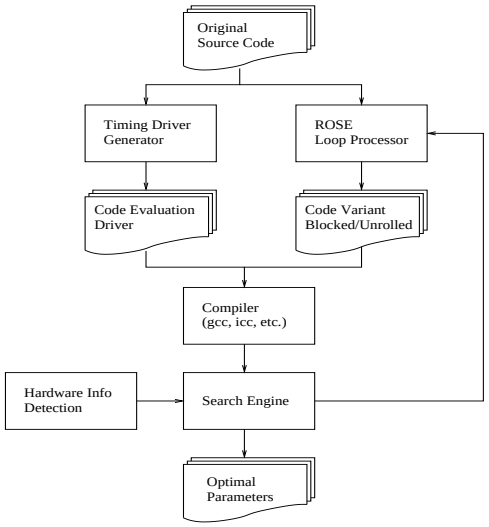

Figure 10: GCO Framework

statistical point of view, the probability of randomly finding better performance than the simplex average case (294 Mflop/s) is 0.079% and the probability of randomly finding better performance than the simplex worst case (273 Mflop/s) is 2.84%. While the simplex technique generally results in a code variant with less than optimal performance, the total search time is only 10 minutes compared to over 30 hours for the exhaustive search.

6 Related Work

There are several projects that adopt an automatic performance tuning strategy to produce highly optimized libraries, but with different approaches to the

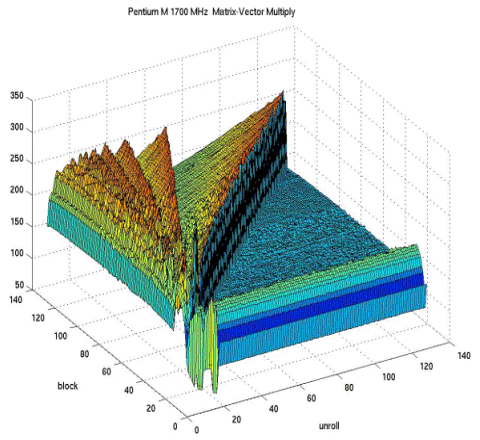

Figure 11: Exhaustive Search of Matrix-Vector Multiply Code

heuristic search. PHiPAC[10] 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 simple heuristic-based search is used for the other levels. ATLAS[8, 9] 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[11] 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[20] 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). However, our goal is to develop a generic search strategy that is effective for a variety of different applications. From our initial experiments with ATLAS and the Generic Code Optimization system, we have found that the simplex method converges relatively fast and produces good results for both.

7 Conclusion

Empirical optimization has been shown to be an effective technique for optimizing code for a particular platform. Since the search heuristic plays such an important role in the system, existing empirical tuning software such as ATLAS [8, 9], PHiPAC [10], and FFTW [11] each have their own search strategy. Our research provides a generic way to search for the optimal parameters and it could be extended to Direct Search Methods such as pattern search methods and methods with adaptive sets of search directions [21].

This paper has demonstrated the effectiveness of the simplex search strategy with ATLAS and the Generic Code Optimization system, but in the future, we would like to evaluate its effectiveness with other tuning systems. Also, while the GA approach did not turn out to be as effective, it has the advantage of being naturally parallellizable. We are planning to implement a parallel version of the GA to be run on a cluster of identical machines.

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.

[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. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, January 2001.

[9] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[10] 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.

[11] 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.

[12] 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.

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

[14] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

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

[16] 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.

[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] Qing Yi and Dan 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.

[19] Dan Quinlan, Markus Schordan, Qing Yi, and Andreas Saebjornsen. Classification and utilization of abstractions for optimization. In *Ths First International Symposium on Leveraging Applications of Formal Methods*, Paphos, Cyprus, Oct 2004.

[20] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[21] 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.