Optimizing Sorting with Machine Learning Algorithms

Xiaoming Li[†]**, María Jesús Garzarán and David Padua**
[†] Department of Electrical and Computer Engineering, University of Delaware
Department of Computer Science, University of Illinois at Urbana-Champaign
xli@ece.udel.edu, {garzaran, padua}@cs.uiuc.edu

Abstract

The growing complexity of modern processors has made the development of highly efficient code increasingly difficult. Manually developing highly efficient code is usually expensive but necessary due to the limitations of today's compilers. A promising automatic code generation strategy, implemented by library generators such as ATLAS, FFTW, and SPIRAL, relies on empirical search to identify, for each target machine, the code characteristics, such as the tile size and instruction schedules, that deliver the best performance. This approach has mainly been applied to scientific codes which can be optimized by identifying code characteristics that depend only on the target machine. In this paper, we study the generation of sorting routines whose performance also depends on the characteristics of the input data.

We present two approaches to generate efficient sorting routines. First, we consider the problem of selecting the best "pure" sorting algorithm as a function of the characteristics of the input data. We show that the relative performance of "pure" sorting algorithms can be encoded as a function of the entropy *of the input data set. We used machine learning algorithms to compute a function for each target machine that, at runtime, is used to select the best algorithm. Our second approach generalizes the first approach and can build new sorting algorithms from a few primitive operations. We use genetic algorithms and a classifier system to build hierarchically-organized hybrid sorting algorithms capable of adapting to the input data. Our results show that the algorithms generated using this second approach are quite effective and perform significantly better than the many conventional sorting implementations we tested. In particular, the routines generated using the second approach perform better than the most popular libraries available today: IBM ESSL, INTEL MKL and the C++ STL. The best algorithm we have been able to generate is on the average 26% and 62% faster than the IBM ESSL in an IBM Power 3 and IBM Power 4, respectively.*

1 Introduction

Although compiler technology has been extraordinarily successful at automating the process of program optimization, much human intervention is still needed to obtain high-quality code. One reason is the unevenness of compiler implementations. There are excellent optimizing compilers for some platforms, but the compilers available for some other platforms leave much to be desired. A second, and perhaps more important, reason is that conventional compilers lack semantic information and, therefore, have limited transformation power. An emerging approach that has proven quite effective in overcoming both of these limitations is to use library generators. These systems make use of semantic information to apply transformations at all levels of abstractions. The most powerful library generators are not just program optimizers, but true algorithm design systems.

ATLAS [10], PHiPAC [2], FFTW [4] and SPIRAL [11] are among the best known library generators. ATLAS and PHiPAC generate linear algebra routines and focus the optimization process on the implementation of matrix-matrix multiplication. During the installation, the parameter values of a matrix multiplication implementation, such as tile size and amount of loop unrolling, that deliver the best performance are identified using empirical search. This search proceeds by generating different versions of matrix multiplication that only differ in the parameter value that is being sought. An almost exhaustive search is used to find the best parameter values. The other two systems mentioned above, SPIRAL and FFTW, generate signal processing libraries. The search space in SPIRAL or FFTW is too large for exhaustive search to be possible. Thus, these systems search using heuristics such as dynamic programming [4, 6], or genetic algorithms [9].

In this project, we explore the problem of generating high-quality sorting routines. Compared sorting with the algorithms implemented by the library generators just mentioned, the difference is that the performnace of the algorithms they implement is complemently determined by the characteristics of the target machine, but not by characteristics of the input data, except the size of the input data. However, the performance of sorting algorithms depends on other input factors such as the distribution of the input data. In fact, as shown both theoretically and practically, multiway merge sort performs very well on some classes of input data sets while radix sort performs poorly on the same sets. For other data set classes we observe the reverse situation. Hence, the approach of today's generators is useful to optimize the parameter values of a sorting algorithm, but not to select the best sorting algorithm for a given input. What is more, not only do we need to tune individual sorting algorithms for different computer architectures, also we need to find the solution of how to combine different sorting algorithms, because different input classes can be best sorted by

1-4244-0910-1/07/$20.00 ©2007 IEEE

different sorting algorithms.

In this project, we study the problem of how to automatically generate highly efficient sorting routines for different computer architectures. Our study has two phases. In the first phase, we try to answer the question how to select, at runtime, the best "pure" sorting algorithm for a given input set to be sorted. In the second phase, we extend and generalize our first approach through the development of a new library generator that produces implementations of composite sorting algorithms in the form of hierarchy of *sorting primitives* whose particular shape ultimately depends on the architectural features of the target machine and the characteristics of the input data. We make three contributions in this project:

- The development of machine-learning techniques that predict and select, based on the characteristics of the input data, the best "pure" sorting algorithm at runtime.
- The definition of sorting primitives by combining which we can represent a large collection of new composite sorting algorithms.
- The development of genetic algorithms and classifier systems that during the installation time searches for the function that maps the characteristics of the input and the architectural features of the target machine to the best composite sorting algorithm.

Our results show that our approach is very effective. The best algorithm we have generated is on the average 36% faster than the best "pure" sorting routine, being up to 45% faster. Our sorting routines perform better than all the commercial libraries that we have tried including IBM ESSL, INTEL MKL and the STL of C++. On the average, the generated routines are 26% and 62% faster than the IBM ESSL in an IBM Power 3 and IBM Power 4, respectively.

The rest of this paper is organized as follows. Section 2 outlines the machine learning techniques that we develop to select "pure" sorting algorithms at runtime. Section 3.1 discusses the primitives that we use to build sorting algorithms. Section **??** explains how to use genetic algorithms to generate a classifier system for sorting rountines. Section 4 shows performance results. Finally Section 5 presents out conclusion.

2 Selection the Best Pure Sorting Algorithm as a Function of the Input Data

Sorting is one of the topics that has been studied most extensively in Computer Science. A large number of sorting algorithms have been proposed and their asymptotic complexity, in terms of the number of comparisons or number of iterations, has been carefully analyzed [7]. However, performance of sorting algorithms on real machines is less well understood especially when complex factors such as locality and instruction level parallelism are taken into account.

Our first approach consists in generating code which at run time selects the fastest among several versions of sorting as a function of characteristics of the input. The first step is to optimize each algorithm. We do this using empirical search to determine the best form of each sorting algorithm. However, no matter how aggressively we tune a sorting algorithm, there is no single sorting algorithm that performs better than all other sorting algorithms for all input sets. Therefore, the second step is to compute a function that maps the characteristics of the input data to one of the algorithms. This function, called *Selection Function* is used at execution time to decide which version of sorting is to be executed each time the generated sorting routine is invoked.

We use versions of quicksort, radix sort, and multiway merge sort as the main alternatives from where the runtime selection will be made. We also considered other sorting algorithms, but we found that in none of them perform better than the best of the first three algorithm. Quicksort and multiway merge sort are comparison-based algorithms, while radix sort is a radix-based algorithm.

2.1 Tuning Each Sorting Algorithm

As mentioned above, our first approach starts with the tuning of each of the three sorting algorithms mentioned above through empirical search. For *quicksort*, we applied all optimizations suggested by Sedgewick [8] and then empirically searched for the best values of two parameters: the number of pivots, and the segment size at which recursion stops and a straight line sorting is applied. The first parameter influence the level of the recursion at which the segments resulting from quicksort fit into cache. The second parameter influences the number of registers are used at the leaf level. These parameter, therefore, influence the performance of quicksort.

For *radix sort* we use the implementation of Jiménez et al [5], which is an improved *Reverse Radix Sort* [7]. The key improvement of CC-radix sort to the Reverse Radix Sort is that CC-radix sort recursively checks if the data structures to sort the keys (the original vector of records, the destination vector, and the counters) fit in the cache. If they do, the simple radix sort algorithm is used. If, however, the data structures do not fit in the cache, the algorithm partitions the bucket into sub-buckets using one step of *reverse sorting*.

In *multiway merge sort* the keys are partitioned into p subsets. In our implementation, the subsets were sorted using CC-radix sort. The subsets are merged using a heap or priority queue [7]. We empirically searched for the best values of two parameters, the *fan-out* and the *heap size*, of multiway merge sort. The fan-out influences spatial locality and heap size influence the number of active input element which reside in cache. Thus, by empirically searching the best value of heap size, we can maximize the locality of multiway merge sort.

2.2 Learning the Selection Function

The performance of a sorting algorithm depends on architectural factors such as cache size, cache line size, and number of registers. Performance also depends on characteristics of the input data that are only known at run time like the number of keys to be sorted, the degree to which the keys are already sorted, and the distribution of the values of the keys. The relation between architectural and in-

put data factors and the values of an algorithm parameters (e.g. height of the heap in multiway merge sort) is not well understood and is quite complex.

The design a strategy to dynamically select sorting algorithms requires the answers to two questions: (1) Which input data characteristics affect the performance of sorting algorithms ?, and (2) How can we compute a function that maps the characteristics of the input onto the best sorting algorithm ?

In our work, we considered two characteristics of the input data: number of keys to sort, and the entropy of the input data. The number of keys determines if the input set can fit into the cache and, as a result, affect the overall execution time. The other factor, the entropy of the input data can be used to differentiate between the relative performance of radix sort and comparison based sorting algorithms which in our case are quicksort and multiway merge sort. CC-radix sort partitions the data in each bucket that does not fit into the cache. If the values of the elements in the input data are concentrated around some values, it is more likely that most of these elements will end up in a small number of buckets. Thus, more partitions will have to be applied before the buckets fit into the cache. On the other hand, when the elements are spread out, values will be distributed among the buckets and fewer partitions will be necessary to fit the buckets in the cache, and as a result CC-radix sort will perform better.

To compute the function that can predict the relative performance of sorting algorithms based on the characteristics of the input data, we assume that our function is linearly separable in the number of keys and the entropy. Experimental data shows that good results are possible under this assumption. It can also be argued that this is a reasonable assumption by observing that the entropies of the most significant digits are more important (have a bigger weight) than the entropy of the least significant digits. The reason is that if the entropy value of the more significant digits is high, it is more likely that the subsets will fit into the cache, and as a result, partitioning will not have to be applied using the low order digits. The relative weights of the entropy of each digit will depend on the amount of data to sort and the size of the cache. Intuitively, for a given cache size, the more data we sort, the more digits we will need to consider until the data fit in the cache.

The Winnow algorithm is a machine learning technique that can learn linear separatable functions. It seems appropriate to deal with this problem. We first train the Winnow algorithm with training data that consists of input sets with different number of keys and entropies. For each input in the training set we measure the performance of each algorithm. For each size of the input data set, the Winnow algorithm will result in a tuned weight vector. In addition to the weight vector we also keep track of which algorithm was better: CC-radix sort or either quicksort for smaller data set sizes or multiway merge for larger sizes. At runtime, the system computes the entropy vector of all the digit positions of the input data. Then, it computes the inner product of the entropy vector and the weight vectors which are learned by the Winnow algorithm. If the result is larger than the threshold, the prediction is to use CC-radix sort. If however, the value is smaller, the algorithm to use will be either quicksort or multiway merge, depending on the input data set size.

3 Generating Composite Sorting Algorithms

Our second strategy is a generalization of our first approach. Since, as discussed above, different sorting algorithms perform differently depending on the characteristics of the input data; it is natural to expect that performance improvements could result from the application of a composite strategy where different algorithms are applied to each partition generated in a recursive sorting algorithm. Composite sorting algorithm can select different methods of sorting based on the characteristic of each partition. In this section, we first outline a framework that contains both sorting primitives and selection primitives, as well as rules of how to combine the primitives to form composite sorting algorithms. Then we present the machine-learning techniques that we used to search for the best composite sorting algorithm using those primitives.

3.1 Sorting Primitives

The building blocks of our composite sorting algorithms are primitives. These primitives were identified based on experiments with three sorting algorithms: quicksort, CC-radix sort, and multiway merge sort; and the study of the factors that affect their performance. Darlington [3] introduced the idea of sorting primitives and identified two of them: merge sort and quicksort. We use six primitives representing the three pure algorithms mentioned above which are the ones that obtained better results in our experiments. If other sorting algorithms with good performance were identified, they could be easily added to our framework.

In this second approach, we searched for an optimal algorithm by building composite sorting algorithms. We use two types of primitives to build new sorting algorithms: sorting and selection primitives. Sorting primitives represent pure sorting algorithms which involve partitioning the data, such as radix sort, merge sort and quicksort. Selection primitives dynamically decide which sorting algorithm to apply.

3.1.1 Sorting Primitives

The composite sorting algorithms considered in this study assume that the data is stored in consecutive memory locations. The data is then recursively partitioned using one of four partitioning methods. The recursive partitioning ends when a leaf sorting algorithm is applied to the partition. We first describe the four partitioning primitives followed by a description of the two leaf sorting primitives. For each primitive we also identify the parameter values that are to be searched for by the library generator.

1. $Divide - by - Value$ (DV)

   This primitive corresponds to the first phase of quicksort which, in the case of a binary partition, selects a pivot

and reorganizes the data so that the first part of the vector contains the keys with values smaller than the pivot, and the second part those that are greater than or equal to the pivot. In our work, the DV primitive can partition the set of records into two or more parts using a parameter np which specifies the number of pivots. Thus, this primitive divides the input set into $np + 1$ partitions and rearranges the data around the np pivots.

2. $Divide - by - position$ (DP)

   This primitive corresponds to multiway merge sort and the initial step breaks the input array of keys into two or more partitions or subsets of the same size. It is implicit in the DP primitive that, after all the partitions have been processed, the partitions are merged to obtain a sorted array. The merging is accomplished using a heap or priority queue [7].

   The DP primitive has two parameters: $size$ which specifies the size of each partition, and $fanout$, which specifies the number of children of each node of the heap.

3. $Divide - by - radix$ (DR)

   The Divide-by-Radix primitive corresponds to a step of the radix sort algorithm. The DR primitive distributes the keys to be sorted into buckets depending on the value of a digit in the record. Thus, if we use a radix of r bits, the keys will be distributed into 2^r sub-buckets based on the value of a digit of r bits. The DR primitive has a parameter $radix$ that specifies the size of the radix in number of bits.

4. $Divide - by - Radix - Assuming - Uniform - Distribution$ (DU)

   This primitive is based on the previous DR primitive, but assumes that a digit is uniformly distributed. The computation of the histogram and the partial sum steps in the DR primitive are used to determine the number of keys containing each one of the digits at a particular position and reserve the necessary space in the output vector. However, these steps (in particular computing the histogram) are very costly. To avoid this overhead, we can assume that a digit is uniformly distributed and that the number of keys containing each digit is the same. Thus, with the DU primitive, when sorting an input with n keys and a radix of size r, each sub-bucket is assumed to contain $\frac{n}{2^r}$ keys. In practice, some sub-buckets will overflow the space reserved, because the distribution of the input vector is not totally uniform. However, if the overhead to handle the cases when there is overflow is less than the overhead to compute the histogram and the accumulation step, the DU primitive will run faster than the DR one. As in DR, the DU primitive has a $radix$ parameter.

   Apart from these primitives we also have recursive primitives that will be applied until the sequence is sorted. We call them leaf primitives.

5. $Leaf - Divide - by - Value$ (LDV)

   This primitive specifies that quicksort must be applied recursively to sort the sequences until they are sorted.

6. $Leaf - Divide - By - Radix$ (LDR)

   This primitive specifies that radix sort is used to fully sort the remaining subsets.

3.1.2 Selection Primitives

In addition to the sorting primitives, we also use selection primitives. The selection primitives are used at runtime to determine, based on the characteristics of the input, the sorting primitive to be applied to each sub-sequence of a given sequence. These selection primitives were designed to take into account the number of keys in the partition and/or their standard deviation. The selection primitives are:

1. $Branch - by - Size$ (BS)

   This BS primitive is used to select different paths based of the size of the partition. Thus, this BS primitive, has one or more ($size_1$, $size_2$, ...) parameters to choose the path to follow. The size values are sorted and used to select $n + 1$ possibilities (less than $size_1$, between $size_1$ and $size_2$, ..., larger than $size_n$).

2. $Branch - by - Entropy$ (BE)

   Like the $Branch - by - Size$ primitive, the $Branch - by - Entropy$ primitive has one or more threshold values, which are the inner-products of entropy and a weight vector, that are used to select the path to proceed with the sorting.

3.2 Using Genetic Algorithms to Optimize Composite Algorithms

We use the eight primitives presented above (six sorting primitives and two selection primitives) to build sorting algorithms. These primitives cannot generate all possible sorting algorithms, but their combination spans a much larger space of algorithms than that containing only the traditional pure sorting algorithms like quicksort or radix sort. Also, by changing the parameters in the sorting and selection primitives, we can adapt to the architecture of the target machine and to the characteristics of the input data.

We combine these primitive using genetic algorithms. Next, we first explain why we believe that genetic algorithms are a good search strategy and then we explain how to use them.

3.2.1 Why Use Genetic Algorithms?

Traditionally, the complexity of sorting algorithms has been studied in terms of the number of comparisons executed assuming a specific distribution of the input, such as the uniform distribution [7]. These studies assume that the time to access each element is the same. This assumption, however, is not true for today's processors with their deep cache hierarchies and complex architectural features. Since there are no analytical models of the performance of sorting algorithms in terms of architectural features of the machine, the only way to identify the best algorithm is by searching.

Our approach is to use genetic algorithms to search for an optimal sorting algorithm. The search space is defined by composition of the *sorting and selection primitives* described in Section 3.1 and the *parameter values* of the primitives. The objective of the search is to identify the hierarchical sorting that better fits the architectural features of the machine and the characteristics of the input set.

There are several reasons why we have chosen genetic algorithms to perform the search.

- Using the primitives in Section 3.1, the sorting algorithms can be encoded as a tree where each primitive is represented as a node. The children of selection primitives are the sorting primitives controlled by the selection primitives. The children of the sorting primitives is the selection primitive applied to each segment resulting from the application of the sorting primitive. The leaf sorting primitives are the leaves of the tree. Genetic algorithms can be easily used to search in the space of possible trees for the most appropriate tree shape and parameter values associated to each node.

- The search space of sorting algorithms that can be derived using the eight primitives in Section 3.1 is too large for exhaustive search.

- Genetic algorithms preserve the best subtrees and give those subtrees more chances to reproduce. Sorting algorithms can take advantage of this since a sub-tree is also a sorting algorithm.

In our case, genetic programming maintains a population of trees. Each tree is an expression which represents a sorting algorithm. The probability that a tree is selected for reproduction (called crossover) is proportional to its level of fitness. The better trees are given more opportunities to produce offsprings. Genetic programming also randomly mutates some expressions to create a possibly better tree.

3.3 Optimization of Sorting with Genetic Algorithms

Encoding As discussed above we use a tree based schema where the nodes of the tree are sorting and selection primitives.

Operators Genetic operators are used to derive new offsprings and introduce changes in the population. Crossover and mutation are the two operators that most genetic algorithms use. Crossover exchanges subtrees from different trees. Mutation operator applies changes to a single tree. Next, we explain how we apply these two operators.

Crossover The purpose of crossover is to generate new offsprings that have better performance than their parents. This is likely to happen when the new offsprings inherit the best subtrees of the parents. In our work we used single-point crossover and we chose the crossover point randomly.

Mutation Mutation works on a single tree. It introduces diversity in the population. Mutation prevents the population from remaining the same after any particular generation [1]. This approach, to some extent, allows the search to escape from local optima. Mutation changes the parameter values hoping to find better ones. Our mutation operator can perform the following changes:

1. Randomly change the values of the parameters in the sorting and selection primitive nodes.

2. Exchange two subtrees.

3. Add a new subtree.

4. Remove a subtree. Unnecessary subtrees can be deleted with this operation.

Fitness Function The fitness function determines the probability of an individual to reproduce. The higher the fitness of an individual, the higher the chances it will reproduce and mutate.

In our case, performance will be used as the fitness function. However, the following two considerations have been taken into account in the design of our fitness function:

1. We are searching for a sorting algorithm that performs well across all possible inputs. Thus, the average performance of a tree is its base fitness. However, since we also want the sorting algorithm to consistently perform well across inputs, we penalize trees with a variable performance by multiplying the base fitness by a factor that depends on the standard deviation of its performance when sorting the test inputs.

2. In the first generations, the fitness variance of the population is high. That is, a few sorting trees have a much better performance than the others. If our fitness function was directly proportional to the performance of the tree, most of the offsprings would be the descendants of these few trees, since they would have a much higher probability to reproduce. As a result, these offsprings would soon occupy most of the population. This could result in premature convergence, which would prevent the system from exploring areas of the search space outside the neighborhood of the highly fit trees. To address this problem, our fitness function uses the performance order or rank of the sorting trees in the population. By using the performance ranking, the absolute performance difference between trees is not considered and the trees with lower performance have more probability to reproduce than if the absolute performance value had been used. This avoids the problem of early convergence and of convergence to a local optimum.

Evolution Algorithm An important decision is to choose the appropriate evolution algorithm. The evolution algorithm determines how many offsprings will be generated, how many individuals of the current generation will be replaced and so on.

In this work we use a *steady-state* evolution algorithm. For each generation, only a small number of the least fit individuals in the current generation are replaced by the new generated offsprings. As a result, many of the individuals from the previous population are likely to survive.

The steady-state evolution algorithm that we use to generate a sorting routine evolves many generations. In each

generation, a fixed number of new offsprings will be generated through crossover and some individuals will mutate as explained above. The fitness function will be used to select the individuals to which the mutation and crossover operators are applied. Then, several input sets with different characteristics (standard deviation and number of records) will be generated and used to train the sorting trees of each generation. New inputs are generated for each iteration. The performance obtained by each sorting algorithm will be used by the fitness function to decide which are the least fit individuals and remove them from the population. The number of individuals removed is the same as the number generated. In this way, the number of individuals remains constant across generations.

Several criteria can be chosen as stopping criteria such as stop after a number of generations, or stop when the performance has not improved more than a certain percentage in the last number of generations.

4 Experimental Results

We evaluated our approach on seven different platforms: AMD Athlon MP, Sun UltraSparc III, SGI R12000, IBM Power3, IBM Power4, Intel Itanium 2, and Intel Xeon. Our library generator produces code with outstanding performance on all these platforms. We will only discuss the results from our second approach for lack of space and also because our second approach subsumes the first one.

Figure 1 shows the performance of our library on IBM Power3. The figure plots the execution time in CPU cycles per key as the standard deviation changes from 2^9 to 2^{23}. The test inputs used to collect the data in Figure 1 contained 14M records, and standard deviations of sizes $4^n * 512$, with n ranging from 1 to 8.

Figure 1 shows that the full Xsort routine, generated by our second approach, is the is on average 26% faster than the best sorting routine in vendor provided libraries, IBM ESSL.

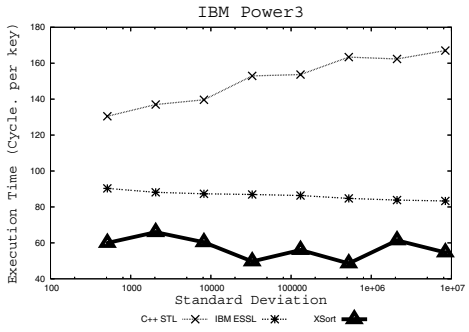

Figure 1: Performance of Xsort.

5 Conclusions

In this paper, we discuss a strategy to build composite sorting algorithms from primitives which are capable of adapting to the target platform and the input data. Genetic algorithms were used to search for the sorting routines. The resulting algorithm is a composite algorithm where a different sorting routine is selected based on the entropy and the number of keys to sort. In most cases, the routines are radix based with different parameters depending on the input characteristics and target machine. The routines generated by our strategy perform better than any commercial routine that we have tried including the IBM ESSL, the INTEL MKL and the STL of C++. On the average, our generated routines are 26% faster than the IBM ESSL on an IBM Power 3 and 62 % on a IBM Power 4.

6 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CSR-0509432. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] T. Back, D. B. Fogel, and Z. Michalewicz. *Evolutionary Computation Vol. I & II*. Institute of Physics Publishing, 2000.

[2] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *Proc.of the 11th ACM International Conference on Supercomputing (ICS)*, July 1997.

[3] J. Darlington. A Synthesis of Several Sorting Algorithms. *Acta Informatica*, 11:1–30, 1978.

[4] M. Frigo. A Fast Fourier Transform Compiler. In *Proc. of Programing Language Design and Implementation*, 1999.

[5] D. Jiménez-González, J. Navarro, and J. Larriba-Pey. CC-Radix: A Cache Conscious Sorting Based on Radix Sort. In *Euromicro Conference on Parallel Distributed and Network based Processing*, pages 101–108, February 2003.

[6] H. Johnson and C. Burrus. The Design of Optimal DFT Algorithms Using Dynamic Programming. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 31:378–387, April 1983.

[7] D. Knuth. *The Art of Computer Programming; Volume3/Sorting and Searching*. Addison-Wesley, 1973.

[8] R. Sedgewick. Implementing Quicksort Programs. *Communications of the ACM*, 21(10):847–857, October 1978.

[9] B. Singer and M. Veloso. Stochastic Search for Signal Processing Algorithm Optimization. In *Proc. of Supercomputig*, 2001.

[10] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.

[11] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and a Compiler for DSP Algorithms. In *Proc. of the International Conference on Programming Language Design and Implementation*, pages 298–308, 2001.