Optimizing Sorting with Genetic Algorithms *

Xiaoming Li, María Jesús Garzarán and David Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
{xli15, garzaran, padua}@cs.uiuc.edu
http://polaris.cs.uiuc.edu

Abstract

The growing complexity of modern processors has made the generation of highly efficient code increasingly difficult. Manual code generation is very time consuming, but it is often the only choice since the code generated by today's compiler technology often has much lower performance than the best hand-tuned codes. A promising code generation strategy, implemented by systems like ATLAS, FFTW, and SPIRAL, uses empirical search to find the parameter values of the implementation, such as the tile size and instruction schedules, that deliver near-optimal performance for a particular machine. However, this approach has only proven successful on scientific codes whose performance does not depend on the input data. In this paper we study machine learning techniques to extend empirical search to the generation of sorting routines, whose performance depends on the input characteristics and the architecture of the target machine.

We build on a previous study that selects a "pure" sorting algorithm at the outset of the computation as a function of the standard deviation. The approach discussed in this paper uses genetic algorithms and a classifier system to build hierarchically-organized hybrid sorting algorithms capable of adapting to the input data. Our results show that such algorithms generated using the approach presented in this paper are quite effective at taking into account the complex interactions between architectural and input data characteristics and that the resulting code performs significantly better than conventional sorting implementations and the code generated by our earlier study. In particular, the routines generated using our approach perform better than all the commercial libraries that we tried including 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.

*This work was supported in part by the National Science Foundation under grant CCR 01-21401 ITR; by DARPA under contract NBCH30390004; and by gifts from INTEL and IBM. This work is not necessarily representative of the positions or policies of the Army or Government.

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 [21], PHiPAC [2], FFTW [7] and SPIRAL [23] 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 [7, 12], or genetic algorithms [19].

In this paper, we explore the problem of generating high-quality sorting routines. A difference between sorting and the algorithms implemented by the library generators just mentioned is that the performance of the algorithms they implement is completely determined by the characteristics of the target machine and the size of the input data, but not by other characteristics of the input data. However, in the case of sorting, performance also depends on other factors such as the distribution of the data to be sorted. In fact, as discussed below, multiway merge sort performs very well on some classes of input data sets while radix sort performs

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

poorly on these sets. For other data set classes we observe the reverse situation. Thus, 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. To adapt to the characteristics of the input set, in [14] we used the distribution of the input data to select a sorting algorithm. Although this approach has proven quite effective, the final performance is limited by the performance of the sorting algorithms - multiway merge sort, quicksort and radix sort are the choices in [14] - that can be selected at run time.

In this paper, we extend and generalize our earlier approach [14]. Our new library generator produces implementations of composite sorting algorithms in the form of a hierarchy of *sorting primitives* whose particular shape ultimately depends on the architectural features of the target machine and the characteristics of the input data. The intuition behind this is that different sorting algorithms perform differently depending on the characteristic of each partition and as a result, the optimal sorting algorithm should be the composition of these different sorting algorithms. Besides the sorting primitives, the generated code contains *selection primitives* that dynamically select the composite algorithm as a function of the characteristics of the data in each partition. During the installation time, our new library approach searches for the function that maps the characteristics of the input to the best sorting algorithms using genetic algorithms [3, 8, 16, 22]. Genetic algorithms have also been used to search for the appropriate formula in SPIRAL [19] and for traditional compiler optimizations [4, 6, 20].

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 discusses the primitives that we use to build sorting algorithms. Section 3 explains why we chose genetic algorithms for the search and explains some details of the algorithm that we implemented. Section 4 shows performance results. Section 5 outlines how to use genetic algorithms to generate a classifier sytem for sorting routines, and finally Section 6 presents out conclusion.

2 Sorting Primitives

In this section, we describe the building blocks of our composite sorting algorithms. These primitives were selected based on experiments with different sorting algorithms and the study of the factors that affect their performance. A summary of the results of these experiments is presented in Figure 1, which plots the execution time of three sorting algorithms against the standard deviation of

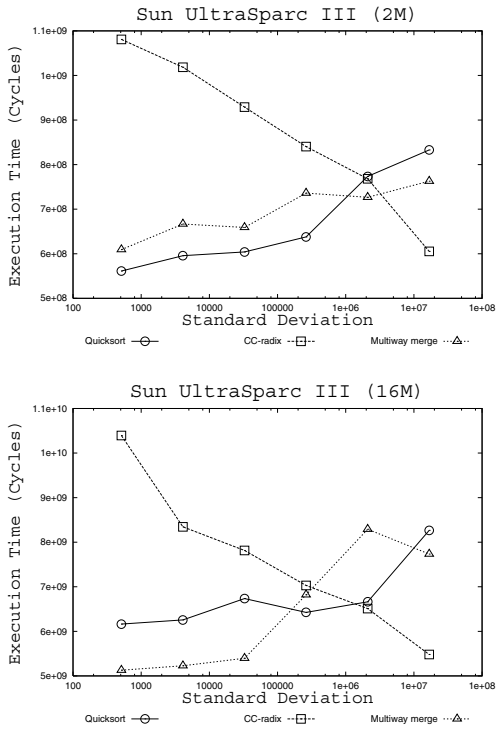

Figure 1: Performance impact of the standard deviation when sorting 2M and 16M keys.

the keys to be sorted. Results are shown for Sun UltraSparc III, and for two data sets sizes, 2 million(M) and 16 million(M). The three algorithms are: quicksort [10, 17], a cache-conscious radix sort (CC-radix) [11], and multiway merge sort [13]. Figure 1 shows that for 2M records, the best sorting algorithm is either quicksort or CC-radix, while, for 16M records, multiway merge or CC-radix are the best algorithms. The input characteristics that determine when CC-radix is the best algorithm is the standard deviation of the records to be sorted. CC-radix is better when the standard deviation of the records is high because if the values of the elements in the input data are concentrated around some values, it is more likely that most of these elements end up in a small number of buckets. Thus, more partition passes will have to be applied before the buckets fit into the cache and therefore more cache misses are incurred during the partitioning. Performance results on other platforms show that the general trend of the algorithms is always the same, but the performance crossover point occurs at different points on different platforms.

It has been known for many years that the performance of Quicksort can be improved when combined with other algorithms [17]. We confirmed experimentally that when the partition is smaller than a certain threshold (whose value depends on the target platform), it is better to use insertion sort or store the data in the registers and sort by exchanging values between registers [14], instead of continuing to recursively apply quicksort. Register sort is a straight-line code algorithm that performs compare-and-swap of values

stored in processor registers [13].

Darlington [5] introduced the idea of sorting primitives and identify merge sort and quicksort as two sort primitives. In this paper, we search 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 a pure sorting algorithm that involves partitioning the data, such as radix sort, merge sort and quicksort. Selection primitives represent a process to be executed at runtime that dynamically decide which sorting algorithm to apply.

The composite sorting algorithm considered in this paper 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 now 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 must be searched 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 that 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 [13]. The merge operation works as follows. At the beginning the leaves of the heap are the first elements of each partition. Then, pairs of leaves are compared, the smaller is promoted to the parent node, and a new element from the partition that contained the promoted element becomes a leaf. This is done recursively until the heap is full. After that, the element at the top of the heap is extracted, placed in the destination vector, a new element from the corresponding subset is promoted, and the process repeats again. Figure 2 shows a picture of the heap.

   The heap is implemented as an array where siblings are located in consecutive positions. When merging using the heap, the operation of finding the child with the smallest key is executed repetitively. If the number of children of each parent is smaller than the number of nodes that fit in a cache line, the cache line will be under-utilized. To solve this problem we use a heap with a fanout that is a multiple of A/r where A is the size of the cache line and r the size of each node. That is, each parent of our heap has A/r children [14]. This takes maximum advantage of spatial locality. Of course, for this to be true, the array structure implementing the heap needs to be properly aligned.

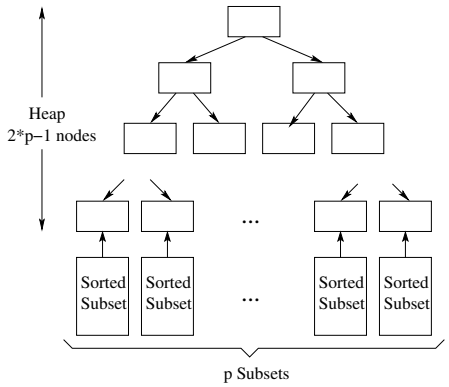

Figure 2: Multiway Merge.

   The DP primitive has two parameters: $size$ that specifies the size of each partition, and $fanout$, that 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 records 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 records will be distributed into 2^r sub-buckets based on the value of a digit of r bits. Our implementation relies on the counting algorithm [13] which, for each digit, proceeds in three steps: the first step computes a histogram with the number of keys per bucket, the second computes partial sums that identify the location in the destination vector where each bucket starts, and a final step moves the keys from the source vector to the destination one.

   The DR primitive has a parameter $radix$ that specifies the size of the radix in number of bits. The position of the digit in the record is not specified in the primitive, but is determined at run time as follows. Conceptually, a counter is kept for each partition. The counter identifies the position where the digit to be used for radix sort starts. Every partition that is created inherit the counter of its parents. The counter is initialized at zero and is incremented by the size of the radix (in number of bits) each time a DR primitive is applied.

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 of each possible value and reserve the corresponding 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 for each possible value 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 partition is sorted. We call them leaf primitives.

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

   This primitive specifies that the DV primitive must be applied recursively to sort the partitions. However, when the size of the partition is smaller than a certain threshold, this LDV primitive uses an in-place register sorting algorithm to sort the records in that partition. LDV has two parameters: np, which specifies the number of pivots as in the DV primitive, and $threshold$, which specifies the partition size below which the register sorting algorithm is applied.

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

   This primitive specifies that the DR primitive is used to sort the remaining subsets. LDR has two parameters: $radix$ and $threshold$. As in LDV, the $threshold$ is used to specify the size of the partition where the algorithm switches to register sorting.

Notice that although the number and type of sorting primitives could be different, we have chosen to use these six because they represent the pure algorithms that obtained better results in our experiments. Other sorting algorithms such as shell sort never obtained the performance of the sorting algorithms selected here. However, they could be included in our framework.

All the sorting primitives have parameters whose most appropriate value will depend on architectural features of the target machine. Consider, for example, the DP primitive. The $size$ parameter is related to the size of the cache, while the $fanout$ is related to the number of elements that fit in a cache line. Similarly, the np and $radix$ of the DV and DR primitives are related to the cache size. However, the precise value of these parameters cannot be easily determined a priori. For example, the relation between np and the cache size is not straightforward, and the optimal value may also vary depending on the number of keys to sort. The parameter $threshold$ is related to the number of registers.

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-partition of a given partition. Based on the results shown in Figure 1, these selection primitives were designed to take into account the number of records in the partition and/or their standard deviation. These selection primitives are:

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

   As shown in Figure 1, the number of records to sort is an input characteristic that determines the relative performance of our sorting primitives. 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)

   Besides the size of the partition, the other input characteristic that determines the performance of the above sorting primitives is the standard deviation. However, instead of using the standard deviation to select the different paths to follow we use, as was done in [14], the notion of entropy from information theory.

   There are several reasons to use entropy instead of standard deviation. Standard deviation is expensive to compute since it requires several floating point operations per record. Although, as can be seen in Figure 1, the standard deviation is the factor that determines when CC-radix is the best algorithm, in practice the behavior of CC-radix depends, more than on the standard deviation of the the records to sort, on how much the values of each digit are spread out. The entropy of a digit position will give us this information. CC-radix sort distributes the records according to the value of one of the digits. If the values of this digit are spread out, the entropy will be high and the sizes of resulting sub-buckets will be close to each other, and, as a result, all the sub-buckets will be more likely to fit in the cache. Consequently, each sub-bucket could be completely sorted with few cache misses. If, however, the entropy is low, most of the records will end up in the same sub-bucket, which increase the likelihood that one or more sub-buckets will not fit in cache. Sorting these sub-buckets could require many cache misses.

   To compute the entropy, at runtime we need to scan the input set and compute the number of keys that have a particular value for each digit position. For each digit, the entropy is computed as $\sum_i -P_i * \log_2 P_i$, where $P_i = c_i/N$, c_i is the number of keys with value i in that digit, and N is the total number of keys. The result is a vector of entropies, where each element of the vector represents the entropy of a digit position in the key. We then compute an entropy scalar value S, as the inner product of the

computed entropy vector (E_i) and a weight vector (W_i): $S = \sum_i E_i * W_i$. The resulting S value is used to select the path to proceed with the sorting. The scalar entropy value and the weight vector are the parameter values needed for this primitive. The weight vector measures the impact of each digit on the performance of radix sort. During the training phase, it can be updated with the performance data using the Winnow algorithm. More details can be found in [14].

Type	Prim.	Parameters
Sorting	*DV*	*np*, number of pivots
	DP	*size*, partition size *fanout* of the heap
	DR	*radix* size in bits
	DU	*radix* size in bits
	LDV	*np*, number of pivots *threshold* for in-place register sort
	LDR	*radix* size in bits *threshold* for in-place register sort
Selection	*BS*	n, there are $n+1$ branches *size*, n size-thresholds for the $n+1$ branches
	BE	n, there are $n+1$ branches *entropy*, n scalar-entropy-value-thresholds for the $n+1$ branches and the weight vector.

Table 1: Summary of primitives and their parameters.

The primitives and their parameters are listed in Table 1. We will use the eight primitives presented here (six sorting primitives and two selection primitives) to build sorting algorithms. Figure 3 shows an example where different sorting algorithms are encoded as a tree of primitives. Figure 3-(a) shows the encoding corresponding to a pure radix sort algorithm, where all the partitions are sorted using the same radix of 2^5. Our DR primitive sorts the data according to the value of the left-most digit that has not been processed yet. Figure 3-(b) shows the encoding of an algorithm that first partitions according to the value of the left-most base 2^8 digit and then sorts each resulting bucket using radix sort with radix size of either 2^4 or 2^8 depending on the number of records of each of the buckets produced by the top level radix sort. Radix 2^4 is used when the number of records is less than $S1$ and 2^8 otherwise. Notice that when the resulting partition has fewer than 16 elements the in-place register sorting algorithm is applied. Figure 3-(c) shows the encoding of a more complex algorithm. The input set is initially partitioned into subsets of 32K elements each. For each partition, the entropy is computed as explained above and, based on the computed value, a different algorithm is applied. If the entropy is less than $V1$, a quicksort is applied. This quicksort turns into an in-place register sorting when the partition contains 8 or fewer elements. If the entropy is more than $V2$ (with $V2 > V1$) a radix sort using radix 2^8 is applied. Otherwise, if the entropy is between $V1$ and $V2$, another selection is made based on the size of the partition. If the size is less than $S1$, a radix sort with radix 2^8 is applied. Otherwise a three-way quicksort is applied. At the end, each subset is sorted, but they need to be sorted among themselves. For that, the initial subsets are merged using a heap like the one in Figure 2, with a $fanout$ of 4, which is the parameter value of the DP primitive.

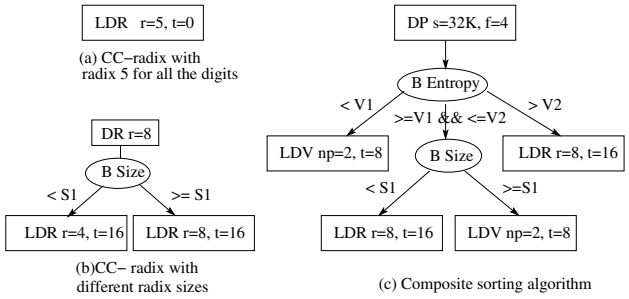

Figure 3: Tree based encoding of different sorting algorithms.

The primitives we are using cannot generate all possible sorting algorithms, but by combining them they can build a much larger space of sorting 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.

3 Gene Sort

In this section we explain the use of genetic algorithms to optimize sorting. We first explain why we believe that genetic algorithms are a good search strategy and then we explain how to use them.

3.1 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 [13]. The studies assume that the time to access each element is the same. This assumption, however, is not true in today's processors that have a deep cache hierarchy 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 2 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 2, the sorting algorithms can be encoded as a tree in Figure 3. Genetic algorithms can be easily used to search in this space for the most appropriate tree shape and parameter values.

- The search space of sorting algorithms that can be derived using the eight primitives in Section 2 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 tree genomes. Each tree genome is an expression that represents a sorting algorithm. The probability that a tree genome is selected for reproduction (called crossover) is proportional to its level of fitness. The better genomes are given more opportunities to produce offsprings. Genetic programming also randomly mutates some expressions to create a possibly better genome.

3.2 Optimization of Sorting with Genetic Algorithms

3.2.1 Encoding

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

3.2.2 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 this paper we use single-point crossover and we choose the crossover point randomly. Figure 4 shows an example of single-point crossover.

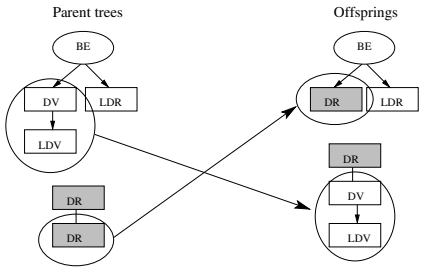

Figure 4: Crossover of sorting trees.

Mutation

Mutation works on a single tree where it produces some changes and 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. Change the values of the parameters in the sorting and selection primitive nodes. The parameters are changed randomly but the new values are close to the old ones.
2. Exchange two subtrees. This type of mutation can help in cases like the one shown in Figure 5-(a) where a subtree that is good to sort sets of less than 4M records is being applied to larger sets. By exchanging the subtrees we can correct this type of misplacement.
3. Add a new subtree. This type of mutation is helpful when more partitioning is required along one path of the tree. Figure 5-(b) shows an example of this mutation.
4. Remove a subtree. Unnecessary subtrees can be deleted with this operation.

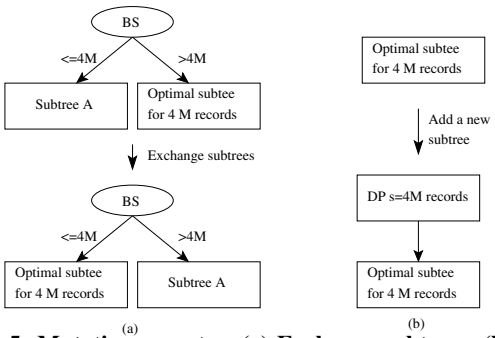

Figure 5: Mutation operator. (a)-Exchange subtrees. (b)-Add a new subtree.

3.2.3 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 other ones. 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

Genetic Algorithm {
P = Initial Population
While (stopping criteria is false) do {
- Apply mutation and crossover and generate set M of k individuals
- P = P ∪ M
- S = Input sets with different sizes and different standard deviations
- Use each genome of P to sort each element of S
- Apply fitness function to remove the k least fit individuals from P.

}}

Figure 6: Genetic Algorithm

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.

3.2.4 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.

Figure 6 shows the code for the steady-state evolution algorithm that we use to generate a sorting routine. 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. 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. The stopping criteria and initial population that we use will be discussed in the next section.

4 Evaluation of Gene Sort

In this section we evaluate our approach of using genetic algorithms to optimize sorting algorithms. In Section 4.1 we discuss the environmental setup. Section 4.2 presents performance results, and Section 4.3 presents the sorting trees produced for each target platform and analyzes their characteristics.

4.1 Environmental Setup

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. Table 3 lists for each platform the main architectural parameters, the operating system, the compiler and the compiler options used for the experiments.

For the evaluation we follow the genetic algorithm in Figure 6. Table 2 summarizes the parameter values that we have used for the experiments. We use a population of 50 sorting trees, and we let them evolve using the steady-state algorithm for 100 generations. However, our experiments (not shown here because of lack of space) show that 30 generations suffice in most cases to obtain a stable solution.

Parameters for Genetic algorithm	
Population Size	50
#*Generations*	100
#*Generated offsprings*	30
Mutation Rate Probability	6%
#*Training input sets*	12

Table 2: Parameters for one generation of the Genetic Algorithm.

Our genetic algorithm searches for both the structure of the tree and the parameter values. Thus, a high replacement rate and a high mutation rate are necessary to guarantee that an appropriate parameter value can be reached through random evolution. We have chosen a replacement rate of 60% which for our experiments, means 30 new individuals are generated through crossover in each generation. The mutation operator changes the new offsprings with a probability of 6%. Also, 12 different input sets are generated in each generation and used to test the 80 sorting trees (50 parents + 30 offsprings). [1]

For the initial population we have chosen trees representing the pure sorting algorithms of CC-radix [11], quicksort [17], multiway merge [13], the adaptive algorithm that we presented in [14], and variations of these. In all the platforms that we tried the initial individuals were quickly replaced by better offsprings, although many subtrees of the initial population were still present in the last generations.

The times to generate the sorting routines vary from platform to platform, and range from 9 hours on the Intel Xeon to 80 hours on the SGI R12000.

4.2 Experimental Results

In this Section we present the performance of sorting routines generated using genetic algorithms. In Section 4.2.1 we compare with other sorting routines and in

[1] We have done experiments varying the parameter values for the genetic algorithm in Table 2, and the results obtained were very similar.

	AMD	Sun	SGI	IBM	IBM	Intel	Intel
CPU	Athlon MP	UltraSparcIII	R12000	Power3	Power4	Itanium 2	P4 Intel Xeon
Frequency	1.2GHz	750MHz	300MHz	375Mhz	1.3GHz	1.5GHz	3GHz
L1d/L1i Cache	128KB	64KB/32KB	32KB/32KB	64KB/64KB	32KB/64KB	16KB/16KB	8KB/12KB (1)
L2 Cache	256KB	1MB	4MB	8MB	1440KB	256KB (2)	512KB
Memory	1GB	4GB	1GB	8GB	32GB	8GB	2GB
OS	RedHat9	SunOS5.8	IRIX64 v6.5	AIX4.3	AIX5.1	RedHat7.2	RedHat3.2.3
Compiler	gcc3.2.2	Workshop cc 5.0	MIPSPro cc 7.3.0	Visual Age c v5	Visual Age c v6	gcc3.3.2	gcc3.4.1
Options	-O3	-native -xO5	-O3 -TARG: platform=IP30	-O3 -bmaxdata: 0x80000000	-O3 -bmaxdata: 0x80000000	-O3	-O3

Table 3: Test Platforms. (1) Intel Xeon has a 8KB trace cache instead of a L1 instruction cache. (2) Intel Itanium2 has a 6MB L3.

4.2.2 with commercial libraries such as INTEL MKL, C++ STL, and IBM ESSL.

4.2.1 Performance Results

We used the genetic algorithm to generate a sorting algorithm to sort 32 bit integer keys. The algorithm has been tuned by sorting input data sets with sizes ranging from 8M to 16M keys, and standard deviations ranging from 2^9 to 2^{23}.

For the experiments in this Section, we sort records with two fields, a 32 bit integer key and a 32 bit pointer. We use this structure because, to minimize data movements of the long records typical of databases, sorting is usually performed on an array of tuples, each containing a key and a pointer to the original record [15, 18]. We assume that this array has been created before our library routines are called.

Figure 7 shows the performance of five different sorting algorithms: quicksort, CC-radix, multiway merge, the adaptive sort algorithm that we presented in [14], and the sorting algorithm generated using the genetic algorithm (*Gene Sort*). For CC-radix sort we use the implementation provided by the authors of [11]. For quicksort, multiway merge sort, and adaptive sort we use the implementations that we presented in [14]. Quicksort and multiway merge sort were automatically tuned to each architectural platform using empirical search to identify parameter values such as fanout or heap size. The adaptive algorithm sorts the input set using the "pure" algorithm that it predicts to be the best out of CC-radix sort, quicksort and multiway merge sort as described in [14]. *Gene Sort* is the algorithm generated using the approach presented in this paper

Figure 7 plots the execution time in microseconds (10^{-6}) per key as the standard deviation changes from 2^9 to 2^{23}. The test inputs used to collect the data in Figure 7 contained 14M records, and standard deviations of sizes $4^n * 512$, with n ranging from 0 to 8. These test inputs were different from the ones used during the training process. For each standard deviation, three different input sets with the same standard deviation were sorted using the five different sorting algorithms. The Figure plots the average of the three running times. Differences between these three running times were smaller than 3%. The test inputs have a normal distribution, which was also the distribution of the training inputs. How-

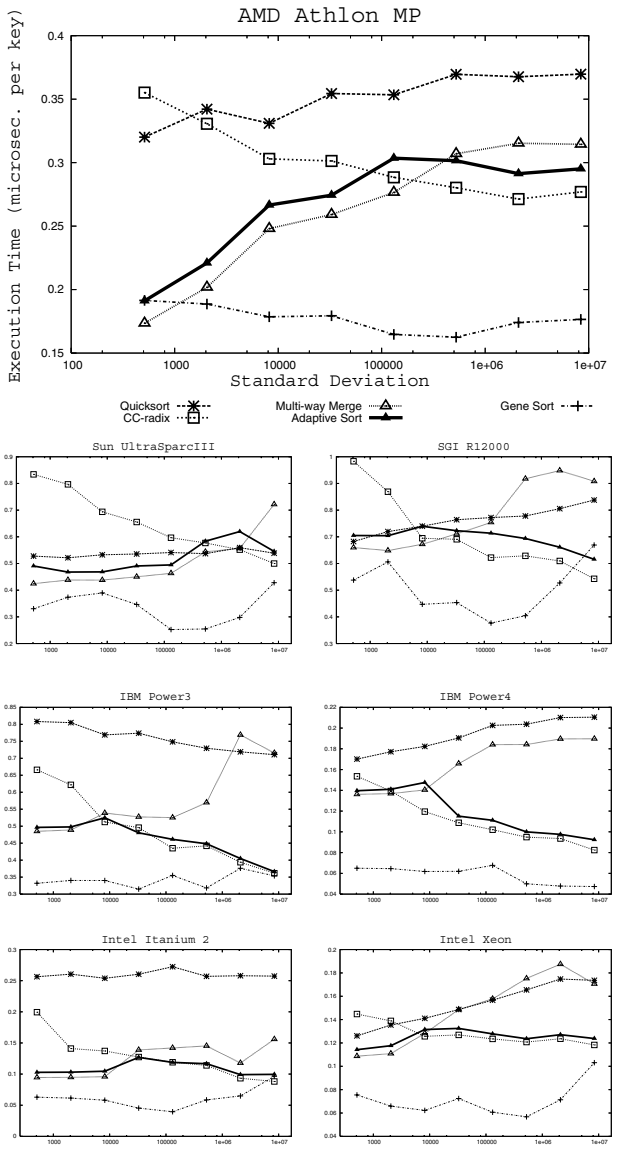

Figure 7: Performance of sorting algorithms as the standard deviation changes

ever, we have run experiments that show that the sorting routines in Figure 7 obtain similar performance when sorting inputs with uniform or exponential distribution. This agrees with the results that we reported in [14].

Figure 7 shows that *Gene Sort* usually performs much better than CC-radix, multiway merge, quicksort or the adaptive sort algorithm. Our adaptive algorithm presented in a previous work [14] predicts correctly the best algorithm among quicksort, CC-radix and multiway merge, but these algorithms usually perform worse than our *Gene Sort* and, as result, the adaptive algorithm cannot outperform the *Gene Sort*. Also, notice that the adaptive algorithm has some overhead over the predicted sorting algorithm since the prediction mechanism needs to compute the entropy of the input set. Our $branch - by - entropy$ primitive also incurs this overhead, but as we will see in Section 4.3, in the end none of the algorithms found by our genetic algorithm used this primitive.

The performance of the *Gene Sort* is slightly worse (less than 7%) than some of the other algorithms in only three cases: on the AMD Athlon for very low values of standard deviation, and on the SGI R12000 and INTEL Itanium 2 for very high values of standard deviation. The fitness function of our genetic algorithm when searching for a general algorithm promotes the sorting algorithm that achieves the best average performance and shows little variation in the execution times (Section 3.2.3). Thus, the use of this fitness function may result in the selection of a sorting algorithm with slightly worse behavior for very low or very high standard deviation since they exhibit very different characteristics from most of the cases. However, our experimental results show that the *Gene Sort* algorithm performs very well across the board. Overall, on Athlon MP, which is the platform with the minimum improvement, the general sorting algorithm obtain a 27% average improvement. On the other platforms, the average improvement over the best of the three "pure" sorting algorithms is 35%.

4.2.2 Performance comparison with commercial libraries

In this Section we compare the performance of *Gene Sort* with the sorting routines from the Intel Math Kernel Library 7.0 (MKL), C++ STL and IBM ESSL version 3.2 (IBM Power3) and version 3.3 (IBM Power4). Execution times were measured by sorting inputs with 14M keys. On the IBM platforms (Power3 and Power4) we sort 32 bit integer keys. In the INTEL platforms (Itanium 2 and Xeon) we sort single precision floating point values since INTEL MKL does not sort integers. We can apply the radix based primitives to sort floating point values because using the IEEE 754 standard the relative order of two non-negative floating-point numbers is the same as the order of the bit-strings that represents them [9]. The keys to sort are located in consecutive positions of an array. For the experiments in this section, we did not include the pointers used in the previous section. We re-generated the sorting libraries to take into account the differences (floating-point numbers for INTEL and no pointers).

Figure 8 shows the execution time of *Gene Sort*, INTEL MKL, C++ STL and IBM ESSL sorting routines (the line corresponding to Xsort will be discussed in the next section). For the INTEL platforms we also show quicksort, since the INTEL MKL implements quicksort. To simplify the plots we do not show results for the other sorting routines in Figure 7, but Gene Sort always performs better than any of them.

Gene Sort is faster than the C++ STL in both IBM Power 3 and Power 4. On the IBM Power 4, Gene Sort is much faster than the IBM ESSL sorting routine. However, on the IBM Power 3, the IBM ESSL sorting routine runs faster than Gene Sort. It is noticeable that the IBM ESSL sorting routine requires more cycles on the Power 4 than on the Power3 (170 versus 90 cycles per key). A possible explanation is that the IBM ESSL library was manually tuned for the Power3 and not for Power4. If our assumption is correct, this would show the disadvantage of manual tuning versus the automatic tuning used in our approach. Gene Sort, thanks to automatic tuning, performs about the same in both platforms, although it is outperformed by the IBM ESSL library in the Power3. In Section 5 we present a slightly different approach that generates the Xsort routine. As can be seen in Figure 8 Xsort is faster than any of the commercial libraries that we considered (including the IBM ESSL library in the Power 3). On the average, Xsort is 26% and 62% faster than the IBM ESSL in Power 3 and Power 4, respectively.

On the Intel Itanium 2 and Intel Xeon, our quicksort what was optimized using empirical search is faster than Intel MKL on the two platforms. C++ STL is marginally slower than our quicksort at most points on Intel Xeon but faster than our quicksort on Intel Itanium 2. However, C++ STL is much slower than our Gene Sort in both platforms. Xsort performs, on the average, 56% and 61% faster than the C++ STL in INTEL Itanium 2 and INTEL Xeon, respectively.

4.3 Analyzing The Best Sorting Genomes

Table 4 presents the best sorting genome found in the experiments of Section 4.2.1 for the Gene Sort routines in Figure 7. The string representation of the genome is of the form *(primitive parameters (child 1) (child 2)...)*, where parameters are those shown in Table 1. For example, *(dr 19 (ldr 13 20))* means apply radix sort with radix 19, and then radix sort with a radix 13 and $threshold$ 20 to apply in-place register sort (see Section 2).

An analysis of the results in Table 4, leads us to two main observations: 1) The radix based primitives $Divide - by - Radix$ (DR) and $Dive - by - Radix - Assuming - Uniform - Distribution$ (DU) are the most frequently used primitives. First, the data are partitioned using radix sort with a large radix (15 or larger). Then, they are partitioned again using radix sort with smaller radix. 2) Selection primitives are rare, and only the $Branch - By - Size$

AMD Athlon MP	Sun UltraSparcIII	SGI R12000	IBM Power3	IBM Power4	Intel Itanium 2	Intel Xeon
(dr 15(dr 9 (ldr 5 20)))	(dr 19(du 6 (ldr 7 20)))	(dr 17(dr 6 (ldr 9 5)))	(dr 19(ldr 13 20))	(dr 16(bs 1186587 (ldr 5 20) (du 3(ldr 13 20))))	(dp 246411 4 (ldr 5 20))	(dr 17 (bs 1048576 (ldr 5 20) (du 6 (ldr 9 20))))

Table 4: Gene Sort algorithm for each platform.

(BS) primitive appear in IBM Power 4 and Intel Xeon. The $Branch - By - Entropy$ (BE) primitive does not appear.

We are unable to verify that the sorting genomes in Table 4 are the optimal ones for each platform, since the search space is so large that exhaustive search is not possible. However, we conducted some experiments reported next, to investigate the optimality of the algorithm found.

We did a sensitivity study to verify that the parameters found are the optimal ones. We have taken the sorting genome for the AMD Athlon MP and IBM Power 3 in Table 4 and we have modified them by changing the radix size. When changing the radix size of the first radix our results show that the radix size selected by our genetic algorithm is the one that runs faster, although it may sometimes incur higher cache or TLB misses. Experiments fixing the first radix and changing the value of the second radix also showed that the parameter selected by our genetic algorithm for this second radix was indeed the best. So, it appears that our approach effectively finds the best parameter values at least for the radix based algorithms and the platforms that we examined.

The observation that selection primitives are rare in the sorting algorithms indicates that our genetic algorithm generates code that is unable to adapt to the standard deviation of the input data set. To study how much performance would improve if the generated code would have such adaptation, we modified our system to generate a classifier system.

5 Classifier Sorting

In this Section we outline how to use genetic algorithms to generate a classifier system [3, 16, 22] for sorting routines that are suited for different regions of the input space. When generating a classifier the selection of a genome in a region of the input space does not depend on the performance of the genome in a different region.

A classifier system consists of a set of rules. Each rule has two parts, a condition and an action. A condition is a string that encodes certain characteristics of the input, where each element of the string can have three possible values: "0", "1", and "*" (don't care). Similarly, the input characteristics are encoded with a bit string of the same length. If i and c are the input bit string and the condition string respectively, we can define the function $match(p, c)$ as follows:

$$match(i, c) = \begin{cases} true, & \forall(j) i_j = c_j \vee c_j =' *' \\ false, & otherwise \end{cases}$$

If $match(i, c)$ is $true$, the action corresponding to the condition bit string c will be selected. A fitness prediction

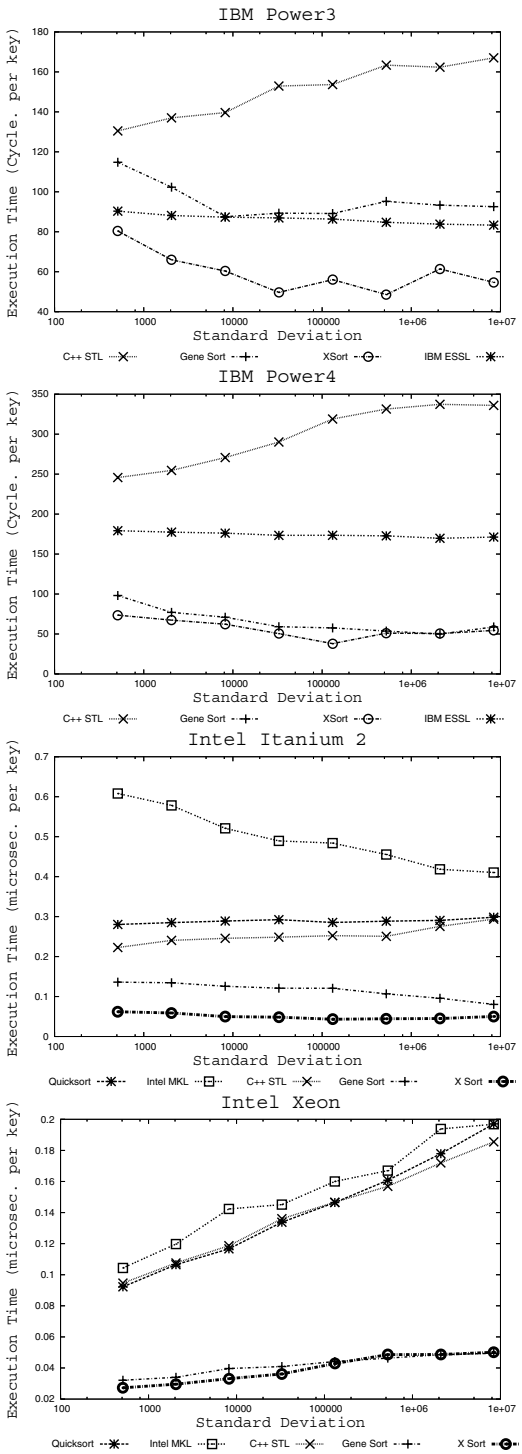

Figure 8: Performance comparison with commercial libraries.

is associated with each rule. For a given input string, there can be multiple matching rules. The rule with the highest predicted fitness will be chosen to act on the input. With the "*" in the condition string, two or more input strings can share the same action. Next we explain how the classifier system is tuned for each platform and input.

5.1 Representation

As we explained in [14] and outlined in Section 2, performance of sorting depends on the number of keys N and the entropy of each digit E_i. Thus, the condition bit string has to encode the different values of these two input characteristics. The number of bits used to encode the input characteristics in the condition bit string will depend on the impact on the performance of each input parameter. So, the more the impact an input parameter has on performance the higher the number of bits that should be used to encode that input parameter.

Our experimental results show that the entropy has a higher impact on performance than the number of keys. As a result, we decided to use two bits to encode the number of keys and four bits to encode the entropy of each digit. Thus, if we assume that N can range from 4M to 16M, the encoding differentials between four regions of length 3M each.

The algorithm selection is done using the rule matching mechanism. As a result, selection primitives are not longer needed to select the appropriate sorting primitives. Thus, the action part of a rule only consists of sorting plans without selection primitives. The sorting genome now has the form of a linear list, not a tree.

Given an input to sort, its input characteristics N and E will be encoded into the bit string i. All the conditions c_j in the rule set of the classifier system will be compared against the input bit string i. All the conditions matching the input bit string i constitute the match set M.

5.2 Training

We train the classifier system to learn a set of rules that cover the space of the possible input parameter values, discover the conditions that better divide the input space and tune the actions to learn the best genome to sort inputs with the characteristics specified in the condition. As before, during the training process inputs with different values of E and N are generated and sorted.

Given a training input, we have a match rule set, which are the set of rules where the condition matches the bit string encoding the input characteristics. We can generate new matching rules applying transformations to both the condition string and the action as described in Section 3.2.2.

We use a classifier fitness based on accuracy [3, 22]. In this type of classifier each rule has two properties, the predicted fitness and the accuracy of the prediction. During the training process, several inputs matching the condition bit string will be sorted using the sorting genome specified in the action part of the rule. The performance obtained will be used to update the accuracy and the predicted fitness. More details of how this algorithm works can be found in [3, 22].

5.3 Runtime

At the end of the training phase, we have a rule set. At runtime, the bit string encoding the input characteristics will be used to extract the match set. Out of these rules, the one with the highest predicted performance and accuracy will be selected, and the corresponding action will be the sorting algorithm used to sort the input. The runtime overhead includes the computation of the entropy to encode the input bit string and the scan of the rule set to select the best rule. In our experiments entropy is computed by sampling one out of four keys of the input. This overhead is negligible compared to the time required to sort input arrays of sizes ($\geq 4M$).

5.4 Experimental Results

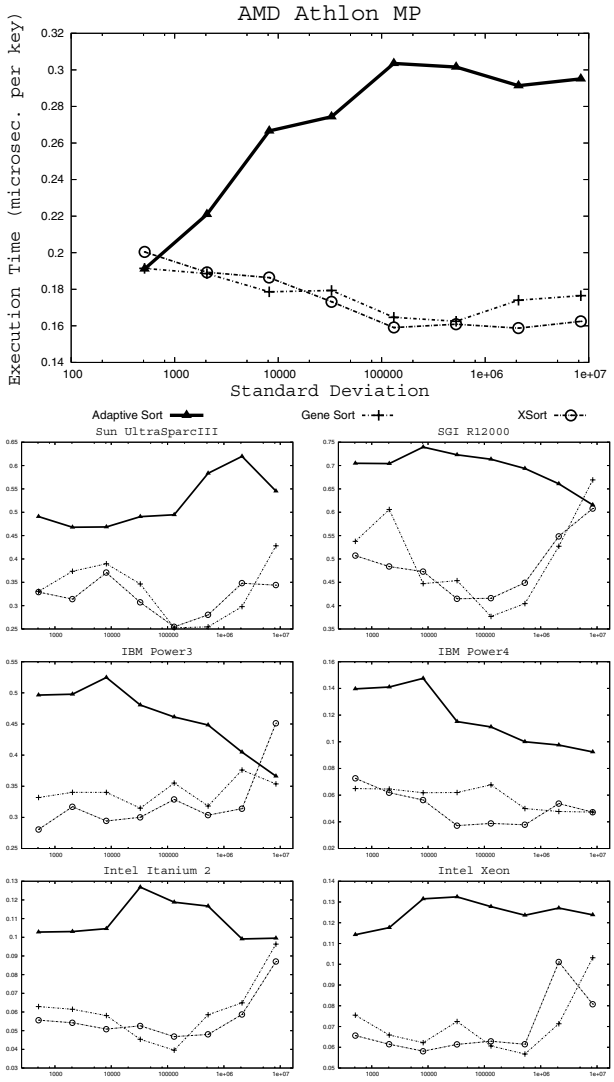

Figure 9: Xsort versus Gene and Adaptive Sort.

	8192	524288	8388608
Sun UltraSparc III	(dr 19 (dr 13))	(dr 21 (ldr 5 20))	(dr 21 (du 5 (ldr 6 20)))
IBM Power3	(dr 22 (du 10))	(dr 19 (du 7 (ldr 6 20)))	(dr 20 (du 10 (ldr 2 20)))

Table 5: Best genomes selected by the classifier sorting library.

Figure 9 compares the execution time of the algorithm generated in the form of a classifier (*Xsort*) versus Gene Sort and Adaptive Sort as the standard deviation changes when sorting 14M records. Xsort is almost always better than Adaptive sort. On the average Xsort is 9% faster than Gene Sort, being up to 12% faster than Gene Sort in the IBM Power4. When compared to Adaptive sort (which is composed of "pure" sorting algorithms), Xsort in on the average 36% faster, being up to 45% faster on Intel Xeon.

Table 5 shows the different sorting genomes found using the classifier for 14M keys and different values of standard deviation for Sun UltraSparc III and IBM Power 3. The table shows that the algorithms are still radix based, but they are different based on the entropy.

6 Conclusion

In this paper, we propose building composite sorting algorithms from primitives to adapt to the target platform and the input data. Genetic algorithms were used to search for the sorting routines. Our results show that the best sorting routines are obtained when using the genetic algorithms to generate a classifier system. 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 generated sorting algorithm is on the average 36% faster than the best "pure" sorting routines on the seven platforms on which we experimented, being up to 42% faster. Our generated routines 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% and 62% faster than the IBM ESSL in an IBM Power 3 and IBM Power 4, respectively.

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] M. V. Butz and S. W. Wilson. An algorithmic description of XCS. *Lecture Notes in Computer Science*, 1996:253+, 2001.

[4] K. Cooper, P. Schielke, and D. Subramanian. Optmizing for Reduced Code Space Using Genetic Algorithms. In *Proc. of the Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 1–9, May 1999.

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

[6] P. K. et al. Finding Effective Optimization Phase Sequences. In *Proc. of the Conf. on Languages, Compilers and Tools for Embedded Systems*, pages 12–23, June 2003.

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

[8] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[9] J. L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[10] C. Hoare. Quicksort. *Computer*, 5(4):10–15, April 1962.

[11] 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.

[12] 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.

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

[14] X. Li, M. J. Garzarán, and D. Padua. A Dynamically Tuned Sorting Library. In *In Proc. of the Int. Symp. on Code Generation and Optimization*, pages 111–124, 2004.

[15] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proc. of the Sigmod Conference*, pages 233–242, 1994.

[16] W. S. Pier Luca Lanzi and S. W. Wilson. *Learning Classifier Systems, From Foundations to Applications*. Springer-Verlag, 2000.

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

[18] A. Shatdal, C. kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. of the 20th Int. Conference on Very Large Databases*, pages 510–521, 1994.

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

[20] M. Stepehnson, S. Amarasinghe, M.Martin, and U. O'Reilly. Meta Optimiization: Improving Compiler Heuristics with Machine Learning. In *Proc. of Programing Language Design and Implementation*, June 2003.

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

[22] S. W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[23] 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.