A Dynamically Tuned Sorting Library *

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

Abstract

Empirical search is a strategy used during the installation of library generators such as ATLAS, FFTW, and SPIRAL to identify the algorithm or the version of an algorithm that delivers the best performance. In the past, empirical search has been applied almost exclusively to scientific problems. In this paper, we discuss the application of empirical search to sorting, which is one of the best understood symbolic computing problems. When contrasted with the dense numerical computations of ATLAS, FFTW, and SPIRAL, sorting presents a new challenge, namely that the relative performance of the algorithms depend not only on the characteristics of the target machine and the size of the input data but also on the distribution of values in the input data set.

Empirical search is applied in the study reported here as part of a sorting library generator. The resulting routines dynamically adapt to the characteristics of the input data by selecting the best sorting algorithm from a small set of alternatives. To generate the run time selection mechanism our generator makes use of machine learning to predict the best algorithm as a function of the characteristics of the input data set and the performance of the different algorithms on the target machine. This prediction is based on the data obtained through empirical search at installation time.

Our results show that our approach is quite effective. When sorting data inputs of 12M keys with various standard deviations, our adaptive approach selected the best algorithm for all the input data sets and all platforms that we tried in our experiments. The wrong decision could have introduced a performance degradation of up to 133%, with an average value of 44%.

1 Introduction

One of the most serious difficulties in the implementation of effective code generators is the lack of a comprehensive methodology to drive optimization transformations. There is still much to be learned about how to identify the program transformations that should be applied to obtain the best performance on a particular target machine. The difficulty increases when runtime adaptation techniques are applied to improve performance by taking into account the characteristics of the input data set.

In this paper, we present and evaluate a strategy for the automatic generation of sorting libraries that involves static and dynamic tuning to obtain the best possible performance. Our library generator, like most other experimental library generators has an installation phase which uses *empirical search* [5, 12] to identify from a set of algorithms and versions of algorithms the one that that performs best on the machine where the library is being installed. Typically, empirical search generates different versions of one or more algorithms and executes them on the target machine. By measuring execution time, empirical search identifies the best version.

Three well known library generators are ATLAS [12], FFTW [3], and SPIRAL [13]. SPIRAL and FFTW generate signal processing libraries. They use empirical search to select an optimal FFT formula. ATLAS generates linear algebra routines. The kernel of ATLAS is matrix multiplication. During the installation phase, ATLAS uses empirical search to identify the best version of a tiled matrix multiplication algorithm. These versions are determined by the parameters of a few transformations, including tiling and unrolling. For the numerical algorithms implemented by the three generators just mentioned, the best shape is usually determined by the characteristics of the target machine and the size of the input data, and not by the characteristics of the input. In contrast, the performance of many sorting algorithms is influenced by the distribution of the values to be sorted. Therefore, code that dynamically adapts to the characteristics of the input has a significant advantage.

In this project, we faced two main difficulties. One was the lack of a precise formulation of the impact of memory hierarchy on the performance of the different sorting algorithms. The performance of sorting algorithms has usually been studied assuming a flat memory although in today's computers the memory hierarchy has a significant impact on performance. The second difficulty, already mentioned, was that the characteristics of the input data set impacts the performance of the sorting algorithm. Most sorting algorithms do not adjust to the input data set and pure algorithms such as radix sort and quicksort are not optimal for all possible inputs. For example, as will be shown below, multiway merge sort performs very well on some data sets where radix sort performs poorly and vice versa.

We take into account the first one of these difficulties by including in the set of algorithms that can be selected at run time a memory-hierarchy-conscious sorting algorithm based on multiway merge sort. Using empirical search, this algorithm is adjusted by our library generator to the memory hierarchy of the target ma-

chine from the register level to the L2 cache. To deal with the effect of the input data set, we propose a runtime adaptation mechanism that selects a sorting algorithm from a set that includes quicksort, our version of multiway merge sort, and a radix-based sorting algorithm [4]. For this runtime adaptation we use a machine learning strategy applied in combination with empirical search which, as will be seen below, is quite effective in the identification of the best strategy in each case. The techniques developed for the automatic generation of a non-numerical algorithm and the application of machine learning for runtime selection are the two most important contributions of this work. No previous study has tried to dynamically identify which is the best sorting algorithm based on the characteristics of the input data and the architecture of the machine.

The remainder of the paper is organized as follows. In Section 2, we introduce several sorting algorithms including a fast radix-based sorting algorithm and our memory hierarchy conscious sorting algorithm. Section 3 presents the factors that affect the performance of several of the sorting algorithms discussed in Section 2. In Section 4, we discuss the installation phase of our library generator and the runtime mechanism we use to select one of the algorithm candidates. In Section 5, we present our experimental results. Finally, concluding remarks are given in Section 6.

2 Sorting Algorithms

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 [6]. In the recent past, there has been a growing interest on improvements to sorting algorithms that do not affect their asymptotic complexity but nevertheless improve performance by enhancing data locality [4, 7, 8]. The algorithms resulting from these improvements have been called *cache-conscious*.

As mentioned in the introduction, the main focus of this paper is the study of strategies for the optimization of sorting algorithms: empirical search to determine the best form of the algorithm and the use of characteristics of the input data to select the best algorithm at run time. Our runtime selection process makes use of the number of records to sort and a characteristic of the distribution of the keys, called *entropy*, that we define in Section 4.

Figure 1 illustrates the type of studies conducted in the past to compare sorting algorithms. Figure 1 shows the execution time of three sorting algorithms: quicksort (*Quicksort*), multiway merge sort (*Multiway merge*), and a cache-conscious radix sort(*CC-radix*) when applied to data with three different distributions (*Uniform*, *Normal* and *Exponential*). In the figure, the number of keys to sort increases from 128K to 16M keys and the standard deviation remains constant to a value of 512K. The keys are 32 bit integers. Results are shown for an Intel Pentium III Xeon platform. The figure shows that the relative behavior of the algorithms does not change with the number of keys or the distribution. A different perspective is obtained from Figure 2, where the execution time is plotted against the standard deviation. Results are shown for two platforms:Intel Pentium III Xeon and Sun Ultra-Sparc III. For each platform, 2M (figures on the left column) and

16M (figures on the right column) keys are sorted. As Figure 2 shows, the standard deviation and the number of keys to sort has a significant impact on the relative performance of the different sorting algorithms. As Figure 1 shows, evaluating the performance of the different algorithm as a function only of the number of keys usually leads to the conclusion that a particular algorithm is the best across the board. This approach does not take into account that, as Figure 2 shows, the relative performance of the different sorting algorithms also depends on the standard deviation of the input data. As it will be shown later, the general trend of each algorithm is the same for all the platforms we considered. For example, as shown in Figure 2, the execution time of CC-radix sort decreases as the standard deviation increases. However, the cross-point is different in each platform. Before we explain our approach, we present some of the implementation details of the baseline algorithms that we have selected to include in our library.
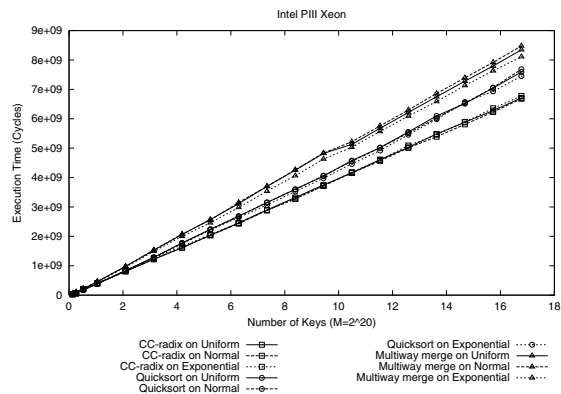

Figure 1. Effect of the distribution and number of keys on the performance of sorting algorithms

In this work, 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 heap sort and merge sort, but we found that in none of the cases we evaluated either of these two algorithms performs 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. Our experiments also show that *insertion sort* and *sorting networks* can sort small amounts of data very efficiently because they can exploit the locality in the cache or at the register level. These two algorithms are used in conjunction with the other three algorithms as will be described below. Throughout the remainder of this section, we will assume that the records to be sorted contain only the key and that these keys are of fixed length.

2.1 Quicksort

Quicksort is an in-place divide-and-conquer algorithm. The algorithm is based on a partition procedure which, given a set of records stored in consecutive locations, chooses a key as a pivot and rearranges the records in such a way that the record containing the pivot is placed in its final position, the records with keys smaller than or equal to the pivot are placed before the pivot, and the records with larger keys are placed after the pivot. The algo-
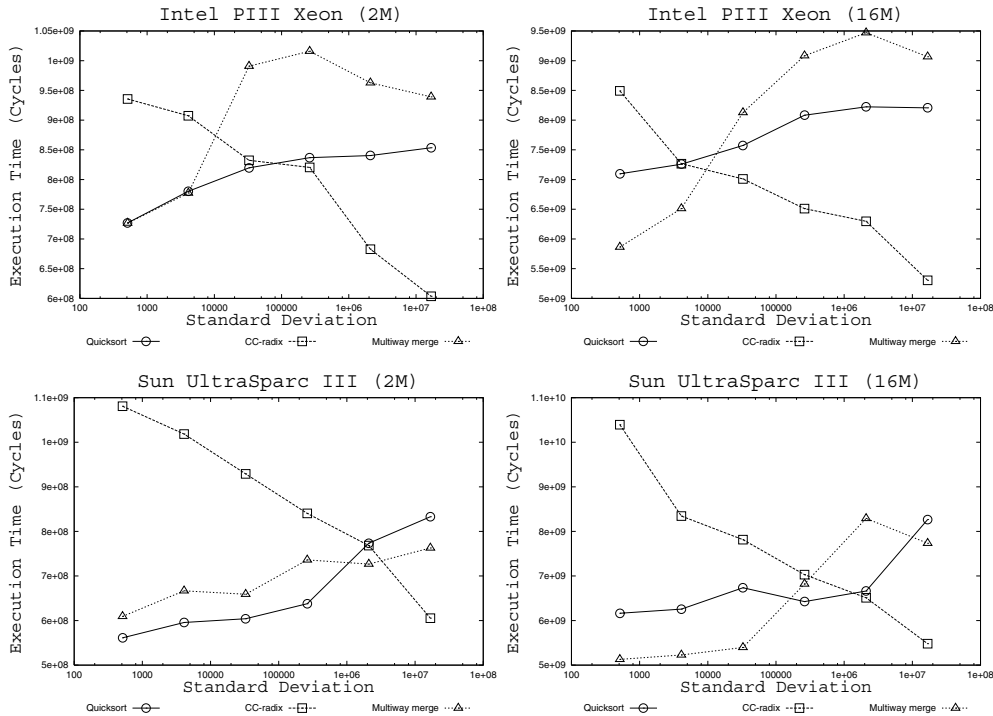
Figure 2. The effect of the standard deviation on the performance of the sorting algorithms. Plots on the left column correspond to 2M keys, while plots on the right column correspond to 16M keys.

rithm recursively works on the region to the left of the pivot and on the region to its right.

Sedgewick [11] suggested several optimizations to Quicksort that we implemented for this study: 1) place a value bigger than the pivot on the rightmost position of the vector and a value smaller than the pivot on the leftmost position to avoid having to check the vector index at each step; 2) proceed iteratively rather than recursively; 3) use the median of the first, the middle and the last keys as the pivot; 4) use insertion sort for small partitions. Sedgewick suggests not to sort immediately "small" unsorted partitions generated by quicksort, but to do it at a final pass that applies insertion sort to the whole vector of records. We implemented this last optimization as one of the alternatives to be evaluated by empirical search.

One of the advantages of Quicksort is that it does not require additional data structures for sorting, since the sorting is done in-place on the input vector of records. The number of comparisons executed by Quicksort is on the average $O(N \log_2(N))$, where N is the number of keys to sort. Quicksort has the best average execution time although its worst case can be $O(N^2)$.

2.2 A Cache-Conscious Radix Sort

Radix sort is the most important non-comparison algorithm [4]. If the keys to be sorted are b-bit integers and the radix sort algorithm uses radix 2^r, the b bits representing an element can be viewed as a set of $\lceil b/r \rceil$ digits of r bits each. The algorithm proceeds in $\lceil b/r \rceil$ phases. The ith phase sorts the key on the value of the ith radix 2^r digit of the keys. The keys are totally sorted after $\lceil b/r \rceil$ phases.

For radix sort we use the implementation of Jiménez et al [4]. To sort the keys in each phase, their implementation relies on a counting algorithm [6] that proceeds in three steps for each phase. First, a vector containing the histogram of the number of records with each value of the digit is computed. Thus, during phase i, the first step computes vector element $v(j)$ which contains the number of keys whose ith digit s equal to j with $0 \leq j \leq 2^r - 1$. Next, an accumulation step computes the partial sum $\sum_{j=0}^{k} v(j)$ with $0 \leq k \leq 2^r - 1$. Finally, a movement step reads the records from the original vector S and moves them to a destination vector. The position where a key is written in the destination vector is indicated in the partial sums for each value of the digit to be sorted. Once a key is moved from the original vector to the destination vector, the corresponding counter is incremented. The original vector and the destination vector interchange their roles in consecutive phases.

The complexity of radix sort is only $O(N)$, where N is the number of keys to sort. Thus, the main advantage of radix is in the number of instructions it executes. It has, however, the disadvantage that its data locality is not very good. To overcome this problem, Jiménez et al [4] have proposed a cache-conscious radix sort (CC-radix sort) algorithm shown in Figure 3.

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 *reverse sorting*. In reverse sorting, a bucket is partitioned using the highest order digit of the key that has not been used yet to partition the data. Keys in each sub-bucket are later sorted using a

```
CC-radix(bucket)

if fits_in_cache (bucket) then
    Radix_sort (bucket)
else
    sub-buckets = Reverse_sorting(bucket)
    for each sub-bucket in sub-buckets
        CC-radix(sub-buckets)
    endfor
endif
```

Figure 3. Pseudocode for CC-radix

simple radix sort by all the lower order digits, but the sub-buckets are already sorted with each other by the higher order digits that were used to create them.

Other implementation details of Jiménez's implementation of CC-radix sort (which is the one that we use in our experiments) are: 1) proceed iteratively instead of recursively; 2) compute the histogram of all the digits of a bucket the first time that radix sort is applied. This reduces the amount of reads of the bucket although it requires as many vectors of counters as digits remain to sort in the corresponding bucket; 3) set the number of bits ,r, that determine the radix 2^r for all reverse sorting instances so that $r \leq \log_2 S_{TLB} - 1$, where S_{TLB} is the number of the TLB entries. The reason for this constraint is as follows. Assume that the number of keys to be sorted is larger than the memory that can be represented at a given time in the TLB, and that the values of the digits are uniformly distributed. Then, if $r > \log_2 S_{TLB} - 1$, the addresses of the 2^r buckets could not be represented in the TLB at the same time. Since locality in the values of a digit is not to be expected, the number of TLB misses could be high.

2.3 Multiway Merge Sort

In multiway merge sort the keys are partitioned into p subsets, which are sequences that are sorted during a first phase. In our experiments, the subsets were sorted using CC-radix sort. The subsets are merged using a heap or priority queue [6]. At the beginning, the leaves in the heap are the first elements of all the subsets. Then, during a second phase, pairs of leaves are compared and the larger/smaller is promoted to the parent node, and a new element from the subset of the promoted element becomes a leaf. This is done recursively until the heap is full. After that, the element in the top of the heap is extracted, placed in the destination vector, a new element from the corresponding subset is promoted and the process is repeated. Figure 4 shows a picture of the heap.

The heap contains $(2 * p - 1)$ nodes. Each node contains a key, and a pointer to the subset where the key comes from. In addition to the heap, this algorithm requires a source vector and a destination vector. This algorithm exploits data locality very efficiently. Notice that during the first phase the only elements that need to be in cache are the records in each subset. During the second phase, only the nodes in the heap $(2 * p - 1)$ need to be in memory. Once an element has been sorted using the heap and moved to the destination vector, it will not be accessed again. The total number of operations is proportional to $N * log_2(2 * p - 1)$.
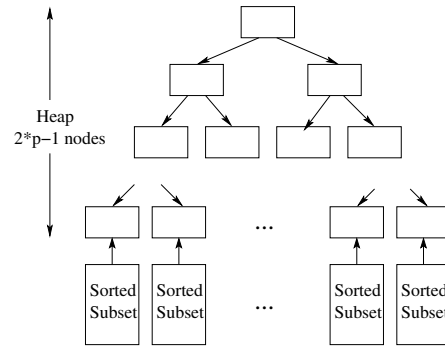

Figure 4. Multiway merge sort.

2.4 Insertion Sort

When the number of keys to sort is small, an algorithm that is known to be very efficient is insertion sorting. For each key $K(i)$ in the vector to be sorted from left to right, the algorithm scans through the keys to the left of $K(i)$ and inserts the key into place by successively moving keys that are bigger than $K(i)$ up to make room.

This algorithm was used in our experiments after the recursive partitions of quicksort or CC-radix sort have produced unsorted sequences with a small number of elements. This algorithm can be very fast for small number of elements. The number of operations in the best case (when keys are in the right order) is proportional to $O(N)$, the worst case (when keys are in the reverse order) is proportional to $O(N^2)$

2.5 Sorting Networks

Sorting networks can also be used to sort small amount of data in the small partitions left by quicksort o CC-radix sort. The network is implemented in software by performing in sequence each layer of the network on processor registers. This algorithm can perform better than insertion sort since sorting networks have a fixed complexity of $O(N \log_2 N)$, as opposed to the worst case of $O(N^2)$ for insertion sort. There are other sorting algorithms that also have a complexity of $O(N \log_2 N)$, but sorting networks are more appropiate to handle small partitions. Figure 5-(a) shows a diagram of a sorting networks similar to those used by Knuth [6], and Figure 5-(b) shows the corresponding code to sort eight elements.

3 Factors

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 input data factors and the values of an algorithm parameters (e.g. height of the heap in multiway merge sort) is usually complex. As discussed in the next section, in this study we used empirical search to determine the value of the algorithm parameters. In other words, our system tries during installation different shapes of the sorting algorithms on the machine where
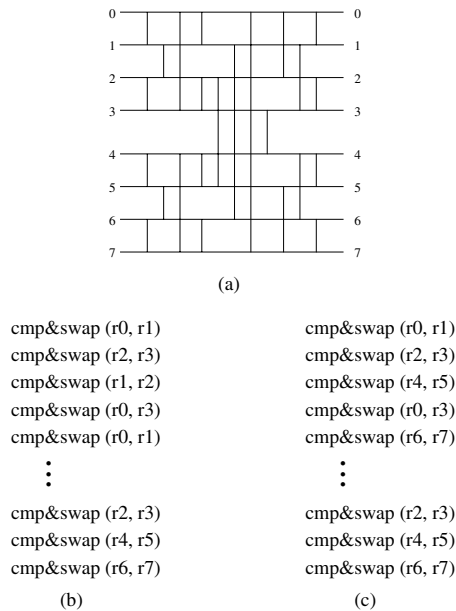
4

```
cmp&swap (r0, r1)        cmp&swap (r0, r1)
cmp&swap (r2, r3)        cmp&swap (r2, r3)
cmp&swap (r1, r2)        cmp&swap (r4, r5)
cmp&swap (r0, r3)        cmp&swap (r0, r3)
cmp&swap (r0, r1)        cmp&swap (r6, r7)
      ⋮                        ⋮
cmp&swap (r2, r3)        cmp&swap (r2, r3)
cmp&swap (r4, r5)        cmp&swap (r4, r5)
cmp&swap (r6, r7)        cmp&swap (r6, r7)
      (b)                      (c)
```

Figure 5. Sorting Network.(a)-Diagram of a sorting network. (b)-Code corresponding to (a). (c)-Code equivalent to (b), but with a different scheduling.

the library is to execute using input data sets with different characteristics. By measuring execution time it identifies the best values for the parameters that determine the shape of the algorithms. The values of some of these parameters are only a function of the target machine, while other parameters depend on the characteristics of the input data set and therefore the value of these parameters can only be decided at runtime, once the data to be sorted is known. It is not always obvious which parameters can be decided statically and which are a function of the input data.

The architectural and input data characteristics influence the parameters through empirical search. Much of this search could be avoided if we could express the values of the algorithm parameters as expressions of values of the architectural features as we did for the ATLAS system [14], but this has not been accomplished and remains an open problem.

3.1 Architectural Factors

In this section we discuss three architectural factors: cache size, the number of registers, and the size of the cache line.

3.1.1 Cache Size

A well-known transformation that has been used to enhance data locality of numerical computations is loop tiling. This transformation divides the (multi-dimensional) iteration space into smaller blocks or tiles with the goal of maximizing data reuse by ensuring that each tile fits in the data cache and by reordering the computation so that several accesses to the same tile are executed consecutively [2].

Tiling can also be applied to sorting algorithms by partitioning the data in such a way that the subset of data to sort fits in the cache. We discuss next how tiling can be incorporated into each sorting algorithm considered by our library generator. Notice that,

to simplify the discussion, what we call the data includes the keys plus the auxiliary data structures that each algorithm requires.

Quicksort. Lamarca et al [7] evaluate a memory-optimized version of quicksort that they call multi-quicksort. When the number of keys to sort is larger than the cache size, multi-quicksort uses several pivots to divide the set of keys into subsets which are likely to fit in the cache. The main challenge in this algorithm is to choose the pivots to maximize the probability that most subsets would fit in the cache. A drawback is that multi-quicksort cannot be done efficiently in-place and executes more instructions that the base quicksort. The results in [7] show that the execution times of multi-quicksort and our implementation of quicksort, which is based on Sedgewick's proposed optimizations, are very close even for large data sets. Thus, we did not implement multi-quicksort.

One of the optimizations proposed by Sedgewick has a negative effect on cache locality and therefore it could be better not to apply it in some cases. This optimization consists in ignoring small partitions while executing quicksort and applying insertion sort over the whole set of the elements at the very end. This optimization eliminates the overhead of calling the insertion sort procedure during the recursion, and as a result, it reduces the number of executed instructions. However, sorting these small partitions as they appear during the recursion procedure, reduces the number of cache misses because the elements to be sorted are already in the cache [7]. Since it is not clear which of these two strategies results in a lower execution time, we try both optimizations when installing our library and choose the one that results in best performance. By empirical search the installation phase determines the size of the partitions to which insertion sort should be applied.

CC-radix. The CC-radix algorithm exploits data locality by partitioning the data into subsets that fit in the cache. When the set of data to sort is larger than the cache size, CC-radix partitions the data until the bucket fits into the cache. This reduces the number of cache misses, although it requires the execution of more instructions.

Remember that the radix for the partitioning (or reverse sorting) is chosen so that $r \leq \log_2 S_{TLB} - 1$, where S_{TLB} is the number of the TLB entries. While this solves the TLB problem, it may force CC-radix to perform a large number of partitions or reverse sorting steps for large data sets, if $\log_2 S_{TLB}$ is small.

Multiway merge. The multiway merge sort algorithm exploits data locality in a very natural way. It partitions the N keys to sort into p subsets, each containing N/p keys. These subsets are sorted using CC-radix. Then a heap of size $(2 * p - 1)$ is used to sort the keys across subsets. When the number of keys to sort is too large, the algorithm could be applied recursively, but we did not do this in the experiments reported in this paper.

To exploit data locality the value of p should be chosen in such a way that the data sets of size N/p and $(2 * p - 1)$ in the corresponding phases fit in the cache. Choosing a value for p that meets the above conditions would reduce the miss ratio. However, we know that the number of operations executed by the heap sort, the CC-radix and quicksort algorithms used in the multiway merge depends on the characteristics of the data to sort. Our final goal is to improve performance not just to minimize cache misses. Our experiments have shown that the best value of p does not only de-

5

pend on the size of the cache but also on the characteristics of the input data.

Finally, notice that there have been many studies by the compiler community on estimating good tile sizes in the context of general purpose compilers, especially for numerical computing. However, the tile size is in general easier to determine for numerical computations than for sorting. For example, in the case of matrix multiplication the total number of arithmetic operations is practically not affected by the size of the tile. Therefore, the tile size can be chosen just by taking into account the number of cache misses, and as a result, the tile size can be obtained by empirical search when a library is being installed [12] or just by evaluating an expression involving the cache size [14]. However, as we have just outlined,this cannot be done for sorting algorithms because their performance depends on the characteristics of the input data.

3.1.2 Number of Registers

The registers are the highest level of the memory hierarchy. When the number of keys to sort is very small, the sorting operation could be partitioned into tiles that are sorted in place using the processor registers. Figure 5-(b) shows the resulting code when register tiling is applied to a sorting network algorithm that sorts eight elements. Figure 5-(c) shows the same example, but in this case the code has been scheduled so that instructions accessing the same element have at least one independent instruction in between.

The code implementing register tiling should be called when a partition is smaller than a certain threshold. This threshold depends on the number of registers that are available to the programmer. If this program is written in high-level language, this number could be further restricted by the compiler. This number is difficult to determine and therefore we use empirical search when installing the library to obtain this threshold parameter. Furthermore, for each threshold value, we search for the schedule that obtains the best performance by varying the number of independent instructions that are placed between two dependent instructions.

Finally, notice that some architectures have special devices that can be used for sorting. This is the case of some Intel machines, which have a stack of processor registers and the corresponding instructions to handle the stack. In this case, the compiler may translate the compare and exchange instructions into the stack instructions. In this case, the performance of the stack, instead of the number of registers, will determine the performance.

3.1.3 Cache Line Size

When several data elements fit in the same cache line, we may reduce cache misses if, when we access an element, we subsequently access the rest of elements in the cache line. This way we are exploiting spatial locality. Sorting algorithms that scan the data like insert sort or quicksort have high spatial locality, and result in a high cache line utilization. Algorithms like CC-radix sort, exploit this spatial locality when reading the source vector, but not when writing the keys into the destination vector.

In the case of the multiway merge sort, the heap is implemented as an array of nodes where siblings are located in consecutive array elements. When sorting using the heap, there are some operations that execute frequently. One of these operations searches for the

child with the largest/smallest key. Thus, 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. For this reason, we use a heap with a fanout of A/r, that is, each parent has A/r children, where A is the size of the cache line and r is the size of each node. Figure 6 shows a heap where each parent has 4 children, what would result in maximum cache line utilization when the cache line has for example 32 bytes and each node has 8 bytes. Of course, for this to be true, the array structure implementing the heap needs to be properly aligned.
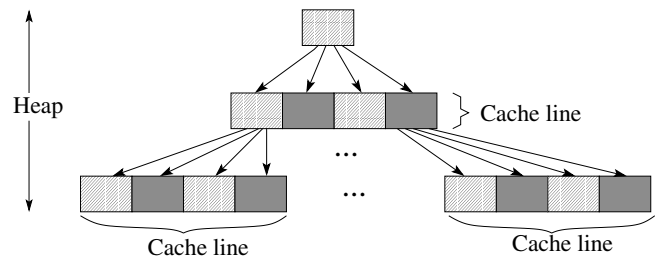

Figure 6. Fill as many as possible child nodes into a cache line.

3.2 Input Data Factors

When evaluating the performance of a sorting algorithm, most studies only consider the algorithm complexity of the average and the worst case for the uniform distribution [6]. However, this may not suffice for the selection of the best sorting algorithm for a particular data set. In the past, the number of keys to sort has been typically used to decide the algorithm to apply, but other characteristics of the input data are usually ignored. Next, we discuss the characteristics of the input data that we have found to affect the performance of the sorting algorithms.

3.2.1 Number of keys to sort

One factor could be the number of keys to sort, although as described in the introduction, when considered by itself, it does not affect the relative performance of the sorting algorithms we considered.

3.2.2 Distribution of the input data

One of the statistical properties that determines how data are spread out is the *distribution*. Most of previous studies that have worked on sorting algorithms have only considered uniform distribution. However, other distributions are also important. For example, in the TPC-H database benchmark [1] the exponential and the uniform distribution are used to generate the database. Figure 1 shows the execution time of quicksort, CC-radix and multiway merge with three distributions: normal, uniform and exponential. The standard deviation is constant. As it can be seen, the distribution does not seem to affect much the performance of the different algorithms in the platform shown. In most cases, differences are within 15%. Experiments in other platforms produced the same results.

6

3.2.3 Standard Deviation

Another property is the standard deviation. Figure 2 shows the effect of standard deviation in performance. The keys were generated using a normal distribution. The Figure shows that the execution times of the different algorithms change with the standard deviation. For 2M keys we see that, for small values of standard deviation quicksort is the best algorithm. For large values of standard deviation, CC-radix sort is the best. However, for 16M keys, the best algorithm for small values of standard deviation is multiway merge. CC-radix sort is also the best one for 16M keys as the standard deviation increases.

Let us see why the performance of CC-radix sort depends on the value of standard deviation. 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 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.

Finally, besides the normal distribution that we used to generate the experiments in Figure 2, we also tried exponential and uniform distributions with different numbers of keys and standard deviations, but we did not see variations in the execution times.

Thus, from our experimental results we consider that the characteristics of the input data that determine the behavior of the algorithms in our set are the number of keys to sort and the standard deviation. Our experiments also take us to conclude, that both factors need to be considered when deciding which is the best algorithm. Taking into account a single factor, such as the number of keys could result in a wrong conclusion, as the results from Figure 1 show.

4 Building the Library

In this section, we discuss the procedures followed to install the sorting library and to decide at execution time, based on the characteristics of the input data, which algorithm to execute and the specific configuration this algorithm should assume. The objective of the installation phase is to determine the configuration that each algorithm should assume to deliver the best performance on the particular target machine. In our current implementation this configuration is independent of the input data in the case of quicksort and CC-radix sort and is a function of the characteristics of the input data in the case of multiway merge.

To determine the algorithm parameters that depend on the architecture, the installation phase executes the algorithm for several different values of these parameters and selects the combination of parameter values that delivers the best performance. This approach is similar to that followed by other empirical optimizers like ATLAS [12] or SPIRAL [13]. To determine at run time the best sorting algorithm for a given input data the installation phase learns a function that maps properties of the input data onto the best algorithm. The training of the function is based on the *Winnow machine learning algorithm* [9], which can learn concepts that are linearly separable. The range of the function includes only two

properties of the input data: the number of records to sort and the *entropy*, which we describe below. Information about the distribution is not necessary, since it has very little influence on the relative performance of the algorithms.

We first describe the entropy in Section 4.1, and then the implementation details, including the Winnow algorithm are presented in Section 4.2.

4.1 Entropy

As explained in Section 3.2, when the number of keys is small (usually less than 3 million) the best sorting algorithm is either quicksort or CC-radix sort. For larger numbers, either CC-radix or multiway merge are the best algorithms. Consequently, our runtime selection method will first make use of the number of keys to sort to determine which two algorithm are in the running and then use entropy to determine when to choose CC-radix sort.

The performance of the CC-radix sort is mainly affected by the number of times that a partition ("Reverse Sorting" in Figure 3) needs to be performed before the resulting buckets fit into the cache. This number depends on how many different values each of the digit positions of the key assumes across the entire input data set. If the most significant digit only assumes a few different values, most of the keys will end up in the same subset when applying CC-radix sort. As a result, the data will have to be partitioned again. If, however, the values of the digit are spread out, the keys will go into different subsets, and it is more likely that the data would fit in the cache, making additional partitions unnecessary.

Although the standard deviation is related to the distribution of each digit position, it does not give us precise information. Standard deviation measures the distribution of key values instead of the distribution of each digit position. In addition, the standard deviation is expensive to compute. It requires several operations per key. We can, however, use the notion of entropy from information theory. Thus, we can compute the entropy of each digit position for all the keys in the input data. If the values of a digit position are spread out, the entropy is high. In this case the data would be distributed in many buckets and fewer partitions would be needed to make the data fit in the cache. If, however, the entropy is low, it means the opposite.

To compute the entropy, we need to scan the data and get the number of keys that have a particular value for a particular 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. We obtain a vector of entropies. Each element of the vector represents the entropy of a digit position in the key.

4.2 Implementation Details

In this section we explain the implementations of our library. We explain the empirical search of parameters that depend on the target architecture in Section 4.2.1, the learning procedure used to compute the selection function in Section 4.2.2, and the runtime selection procedure in Section 4.2.3.

IEEE
COMPUTER
SOCIETY

4.2.1 Empirical search of parameters that depend on the architecture

To optimize performance, empirical search is used to determine the exact form that the sorting algorithms should have for the machine where the library is being installed. In the case of quicksort and CC-radix sort, only one configuration is used through out this work. In the case of multiway merge sort, the best configuration is a function of the entropy vector and the number of keys of the input data set. Therefore, at installation time, the best configurations (size of the heap and fanout) are identified for several values of the pair (dataset size, entropy vector). At run time, the system counts the number of records, N, and computes the entropy vector of the input set, E, and selects from the table the configuration of multiway merge sort corresponding to the index of the table that is closest to the pair (N, E).

For quicksort, empirical search is used to determine whether it is better to use insertion sort or sorting networks for small partitions. In addition, empirical search looks for the threshold below which one of these two algorithms is to be applied. Thus, the installation phase of the library generates an input data set and sorts it in four different ways: i) using only quicksort, ii) using quicksort first, leaving partitions smaller than a threshold unsorted and at the end applying insertion sort to the whole array of keys, iii) using quicksort and immediately sorting partitions smaller than the threshold using insertion sort, and iv) quicksort using sorting networks when small partitions are found. In ii), iii) and iv) we need to find a threshold. Thus, we run these options with several thresholds that range from 8 to 32 in steps of 4. The experiment is executed with several input sets, to reduce statistical errors. The option that obtains the best performance will be used to generate the code for quicksort. Notice that our library is written in C, and consequently the architectural features of the machine and the interaction between the compiler and the C code will determine the threshold values that obtain the best performance. Also, when searching for a threshold value for sorting networks, for each threshold we try different schedules (Section 3.1.2). However, the scheduling in the C code will interact with the scheduling that the compiler does.

In the case of CC-radix sort, once the partition fits in the cache, the remaining digits are sorted using a simple radix algorithm. However, if the amount of elements in the set is small enough it can be sorted using insertion sort or sorting networks. Thus, we use empirical search to find out which of these options is the best: i) CC-radix with radix sort, or ii) CC-radix with insertion sort iii) CC-radix with sorting networks. Again, we need to find the threshold value for the size of the set to be sorted. The procedure is similar to the one used in quicksort, and again the option that results in the best performance will be used to generate the code for CC-radix sort.

For multiway merge, we need to find the size of the heap and the fanout. However, as mentioned above, these values not only depend on the characteristics of the target machine but also on the input data. The installation phase searches for their best value during the learning procedure explained in the next section.

4.2.2 Learning Procedure

After computing the configuration of quicksort and CC-radix sort, the next step of the installation process is to find a function f, that based on the number of keys to sort (N), and the entropy vector (E) predicts the best algorithm among CC-radix, quicksort, or multiway merge. $f : (N, E) \rightarrow \{\text{CC-radix}, \text{Multiway Merge}(N, E), \text{Quicksort}\}$. Here, multiway merge is indexed by N and E because the values of the heap size and fanout of this algorithm will depend on the number of keys to sort, and the entropy vector of the input.

This function is evaluated in two steps. The first step uses the size of the input data to determine whether the comparison should be between quicksort and CC-radix sort or between multiway merge sort and CC-radix sort. The second step makes a binary decision between the pair of sorting algorithms selected by the first part. The second part of the function is learned at installation time using the Winnow algorithm. This algorithm computes weights w_i and a threshold θ such that $\sum_i w_i * E_i > \theta$ if and only if CC-radix sort performs better than the other algorithms (quicksort or multiway merge sort) for input data with the entropy vector E.

We assume that the second step of our function is linearly separable as assumed by the Winnow algorithm. The experimental data presented in the next section show 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 those of the least significant ones. 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 seems appropriate to deal with this problem.

The training data consist of input sets with different number of keys and standard deviations. For each number of keys, we generate input sets with standard deviations of sizes $8^n * 512$, with n ranging from 0 to 5. It is very difficult to generate an input set with a given $entropy$ vector, so we use different standard deviation to control $entropy$ indirectly. Each input set is sorted with all the algorithms: quicksort, CC-radix, and multiway merge sort. In the case of multiway merge sort, for each of these input sets the system empirically searches for the best value for $size\_of\_the\_heap$ and the $fanout$ of the heap. For quicksort, and CC-radix sort we have previously determined the best parameters for these algorithms and these platforms.

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 inputs.

Notice that for each size of the input data in the training set we have searched the value for $size\_of\_the\_heap$ and the $fanout$ for the multiway merge algorithm. As mentioned in Section 4.2.1, we

keep this information in a table indexed by the amount of data to sort and the entropy vector.

4.2.3 Runtime Procedure

At runtime, the system computes the entropy vector of all the digit positions of the input data. Then it computes the inner product (S in Figure 7) of the entropy vector and the weight vector corresponding to the size of the actual input data set. 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. This algorithm we call *Select Algorithm* and it is shown in Figure 7-(a).

When the predicted algorithm is multiway merge we need to access the table that keeps the parameters for $size\_of\_the\_heap$ and the $fanout$. The algorithm is shown in Figure 7-(b)

SELECT_ALGORITHM($Src, W, Threshold, Alg$)
; Src: Input data
; W: Weight vector from Winnow algorithm
; $Threshold$: Threshold
; Alg: Alternative algorithm

Sample the input array and compute the entropy vector E_i
Compute $S = \sum_i W_i * E_i$
if $S \geq Threshold$
    select CC-radix
else
    select Alg

<center>(a)</center>

SELECT_MEMSORT_PARAMETER($S, Smulti$)
; S: the inner-product of E and W as computed by the
              $select\_algorithm$
;Smulti: Vector generated during the learning process.
              For each Si computed during the training process we
              have the $size\_of\_the\_heap$ and the $fanout$. Smulti
              is sorted.

Find $t \in$ Smulti such that t is the closest to S
Use the parameters corresponding to t

<center>(b)</center>

Figure 7. Runtime algorithms. (a)-Select Algorithm (b)- Select Multiway Merge Parameters.

5 Evaluation

In this section we present some measurements of the behavior of our sorting library generator. In Section 5.1, we describe the environmental setup that we used for the evaluation. Section 5.2 presents the performance results, and in Section 5.3 two sensitivity analysis experiments are discussed.

5.1 Environmental Setup

We evaluated our sorting library on seven different platforms: AMD Athlon MP, IBM Power3, Sun UltraSparc III, Intel Pentium III Xeon, SGI R12000, Intel Pentium IV, and Intel Itanium 2. Table 1 lists for each platform the architectural parameters, the version of the operating system, the compiler, and the compiler options used for the experiments.

All experiments sort records with two fields, a 32 bit integer key and a 32 bit pointer. The reason for this choice is that for 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 [10]. We assume that this array has been created before our library routine is called.

During the installation of our sorting library we used training sets of input data with 4M and 16M records with standard deviations of sizes $8^n * 512$, with n ranging from 0 to 5 and a normal distribution. Notice that by varying the standard deviation of the input data we also change the entropy.

For the experiments we used an implementation of CC-radix sort generously provided by the authors of [4].

The installation time of our library varies depending on the platform, but it ranges from 35 minutes in Intel Itanium 2 to 120 minutes in SGI R12000.

5.2 Performance Results

Figure 8 presents plots of the execution time against the standard deviation when sorting 12M records for four different sorting algorithms: *adaptive sort*, which is the algorithm executed when our sorting library is called, quicksort, CC-radix, and multiway merge sort. The performance is measured in cycles per key and results are shown for the seven platforms in Table 1. The input data sets used for the experiments in Figure 8 differ from the data sets used for training during installation in the number of records and the standard deviations. Specifically, the input data sets used for the experiments contain 12M records, and standard deviations of sizes $4^n * 512$, with n ranging from 0 to 8. The weight vector used by our runtime system was computed during training based on input data sets containing 16M records. Also, the table mapping the entropy vector to the best parameters of the multiway merge was computed using a training input of 16M records. The distribution of the data used in the experiments is the same as that of the training set (normal distribution). However, we have found that similar results are obtained with other distributions.

Figure 8 shows that adaptive sort chooses the best algorithm for all the platforms. As discussed below, the performance of adaptive sort is somewhat lower than that of the best algorithm at each point due to the overhead associated with the process of selecting the best algorithm. For the data set sizes of 12M records used in this experiment the trend is the same in all the platforms. In all cases, multiway mergesort is the fastest algorithm when the value of the standard deviation is low. And, as the standard deviation increases, CC-radix sort improves and eventually becomes the faster. Adaptive sort identifies the best algorithm and the cross-point correctly in all platforms. This cross-point is at different values of standard deviation in each platform. Thus, our results indicate that the use of the entropy and the amount of data, together with the Winnow algorithm, systematically leads to the selection of the best sorting algorithm.

The versions of quicksort and CC-radix sort used in the experiments are those obtained statically at installation time using empirical search.

The use of runtime decision of adaptive sort introduces an average overhead of a 5%. This overhead comes from sampling the input data set, the computation of the vector of entropies, and the prediction. To compute the entropy we used a sample of 1 element

<center>9</center>

	AMD	IBM	Sun	Intel PIII	SGI	Intel PIV	Intel Itanium2
CPU	Athlon MP	Power3	UltraSparc III	PIII-Xeon	R12000	Pentium IV	Itanium 2
Frequency	1.2GHz	375MHz	750MHz	550MHz	300MHz	2GHz	1.5GHz
L1d/L1i Cache	64KB/64KB	64KB/64KB	64KB/32KB	16KB/16KB	32KB/32KB	8KB/12KB	16KB/16KB
L2 Cache	256KB	8MB	8MB	512KB	4MB	512KB	256KB
Memory	1GB	8GB	4GB	1GB	1GB	512MB	8GB
OS	RedHat9	AIX4.3	SunOS5.8	RedHat7.3	IRIX64 v6.5	RedHat7.2	RedHat7.2
Compiler Version	gcc3.2.2	VisualAge c v5	Workshop cc v5.0	gcc3.3.1	MIPSPro cc v7.3.1.1m	gcc3.3.1	gcc3.3.2
Compiler Options	-O3	-O3 -bmaxdata: 0x80000000	-native -xO5	-O3	-O3 -TARG: platform=IP30	-O3	-O3

Table 1. Test Platforms. L1d stands for L1 data cache, while L1i stands for L1 instruction cache. Intel Pentium IV has a 12KB trace cache instead of a L1 instruction cache. Intel Itanium 2 has a L3 cache of 6MB.
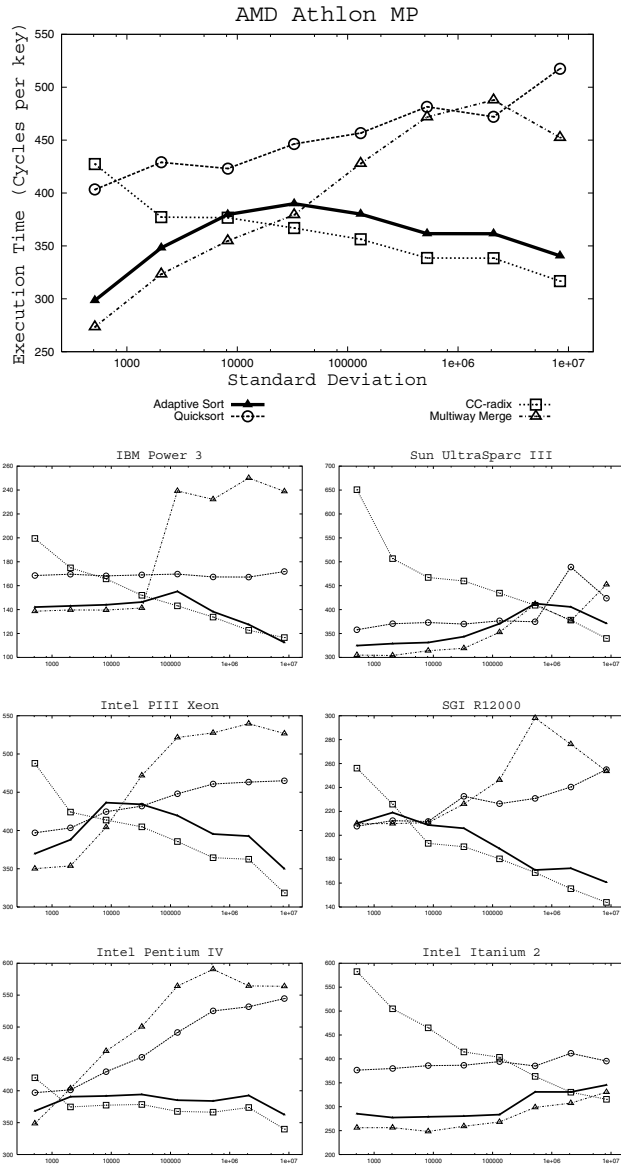

Figure 8. Execution time versus standard deviation for our library and the different sorting algorithms when sorting 12M records.

out of 4 in the input data. Scanning all the data in the input instead of sampling would have resulted in an additional overhead of 2%. This overhead could perhaps be reduced without affecting accuracy by applying more sparse sampling. Also, notice that this overhead pays off since in most situations the wrong decision leads to much lower performance. In addition, when the predicted algorithm is CC-radix sort, most of the operations that cause the overhead could be used to replace some of the operations of CC-radix sort. The reason is that the histogram built to compute the entropy can be reused in CC-radix sort. We did not apply this optimization because for our experiments we used the CC-radix sort provided by Jimenez et al. [4] and we did not modify their code.

If we compare quicksort, CC-radix, and multiway merge sort, we find that quicksort is never the best algorithm when sorting 12M records, although in a few situations it obtains the same performance as one or both of the other two. Multiway merge sort is better than CC-radix sort for small standard deviations because small standard deviation tend to increase the number of keys with the same value. As a result, CC-radix sort tends to execute more partitions to fit the data into the caches. The partition process of CC-radix sort has a high data miss ratio that hinders performance. However, multiway merge sort naturally partitions the data in such a way that the data miss ratio can be kept low. In addition, as the standard deviation decreases, the parent node and the child node in the heap are more likely to have the same value. When that happens, no data movement is necessary. As a result, the number of instructions executed by multiway merge sort is very likely to be small for low values of standard deviation.

As the standard deviation increases, CC-radix sort needs fewer partitions to accommodate the data in the cache and, as a result, it performs better. For multiway merge sort the situation is just the opposite. More keys have different values and the number of operations in the heap increases.

We also ran experiments with fewer records. In particular, for 2M keys, quicksort and CC-radix sort obtained the best performance. In this case, our adaptive sort algorithm was also able identify the best algorithm.

Overall, our adaptive sort algorithm has proven to be very effective to predict the correct algorithm. Results in Figure 8 show that when sorting data inputs of 12M keys with various standard deviations, our adaptive approach selected the best algorithm for all the input data sets and all the platforms. The wrong decision could have introduced a performance degradation of up to 133% with an average value of 44%. This indicates that the technology

IEEE COMPUTER SOCIETY

that we have presented in this paper is well suited to the problem for which it was used.

5.3 Sensitivity Analysis

In this section we study how sensitive the performance is to the variation of the algorithm parameters that are identified at installation time by the empirical search. We present results for the algorithms applied to small partitions by quicksort and for the heap size in the implementation of multiway merge sort.

Table 2 shows the execution time measured in seconds of four different versions of quicksort on three platforms: SGI R12000, Sun UltraSparc III and Intel PIII Xeon. For each platform we show results for two data set sizes, 4M and 16M, and four different options: plain quicksort (*Quicksort*), quicksort with a single insertion sort applied as a single pass at the end (*Insert Sort at the end*), quicksort with insertion sort applied to the small partitions as they appear (*Insert Sort at each partition*), and quicksort with sorting networks applied to small partitions as they appear (*Sorting Networks*). For the last three cases, results are shown for the thresholds that delivered the best result. The thresholds are the values that determine the partition size for which quicksort switches to one of the options just mentioned. The results show that quicksort plus sorting networks is the optimization that delivers the best performance. The only exception occurs Sun UltraSparc III with 16M, where insertion sort at the end obtains slightly better performance. The R12000 processor is the platform where sorting networks obtained the largest improvement, around 22% when compared to plain quicksort. On the average, sorting networks obtained a performance improvement of 15% when compared to plain quicksort.

	SGI R12000		Sun UltraSparcIII		Intel PIII-Xeon	
	4M	16M	4M	16M	4M	16M
Quicksort	2.960	13.915	1.579	6.498	2.429	10.886
Insert Sort at the end	2.676	11.576	1.406	6.098	2.174	9.759
Insert Sort at each partition	2.859	11.976	1.475	6.415	2.321	10.486
Sorting Networks	2.275	10.994	1.372	6.160	2.081	9.207

Table 2. Execution time in seconds for quicksort and quicksort plus some optimizations using insert sort or sorting networks.

We now discuss how different sizes of the heap affect the performance of multiway merge sort. Notice that for a fixed number of keys to sort, the heap size determines the size of each sorted subset and vice versa. To analyze whether the best size of the subsets depends only on the cache size or on both cache size and input data, we used sets of 12M records generated with two different standard deviations (512 and 32768) and we sorted them using subsets of different sizes. Figure 9 shows the results. For each value of standard deviation the figure shows how the L1 data cache misses, L2 data cache misses, number of instructions executed and total clock cycles change as the size of the subset changes from 32 to 8M of records (notice that the values on the X axis have a log-

arithmic scale). The fanout of the heap is kept constant at two. The experiments were conducted on a R12000 processor and the measurements were done using hardware counters.

Comparing the results in Figure 9 for the two values of the standard deviation we can observe that the plots of L1 cache miss, L2 cache miss, and instruction count behave differently, and as a result the best performance for each value of the standard deviation (shown in the Execution Time plot of Figure 9-(d)) is obtained at different subset sizes. Figure 9 shows that when sorting the same number of records and using subsets of the same size, different standard deviations result in different cache misses, number of executed instructions, and total execution times. These results confirm that, as expected, sorting behaves differently than dense numerical linear algebra. For example, given a tiled implementation of matrix multiplication, if the matrix and tile size are kept constant, the cache misses, the number of executed instructions, and cycles remain constant. Thus, for matrix multiplications, the problem of searching for the optimal tile size can be simplified to that of finding the tile size that minimizes the data cache misses, and therefore a simple expression of the cache size can be used to compute the optimal value of the tile size [14]. In sorting, the problem is more complex, and a more complex approach involving runtime decisions such as the one discussed above is necessary.

Our approach seems to be quite effective. For the example shown in Figure 9 where 12M records are sorted, our runtime algorithms used the size of the subset obtained with a training set of 16M records. For the standard deviation of 512, the library routine selected a size of 16K, which is only 5% slower than the best size. For the standard deviation of 32768, the runtime selected a size of 32K, which is the value that obtained the best performance.

Finally, notice that to verify that the differences in cache misses, instructions and cycles executed depend on the standard deviation, and not on the values to sort, we ran several times the same experiment with different input data sets that had the same standard deviation but different values. We found that the differences for each of the events shown in Figure 9 were always less than 5%.
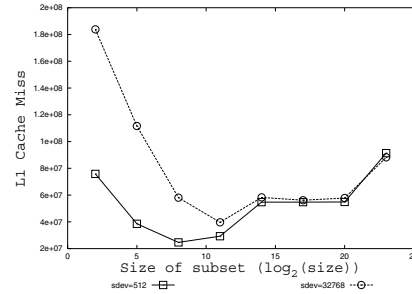
6 Conclusion

In this paper we described an approach for the automatic generation of sorting libraries that produces routines that are highly tuned to the architecture of the target machine and that dynamically selects the best routine based on the characteristics of the input data.

The contribution of this paper is threefold. First, we identify the architectural and runtime factors that affect the performance of the sorting algorithms from a set that includes quicksort, CC-radix and multiway merge. Second, we use empirical search to identify the best shape and parameter values of a sorting algorithm. In addition, we identify those parameters that depend on both, architectural and input characteristics, and show that the use of the entropy is appropiate to select at runtime the best values. Finally we use the Winnow machine learning algorithm and two properties of the input data (entropy and amount of data) to select the best algorithm from the set.
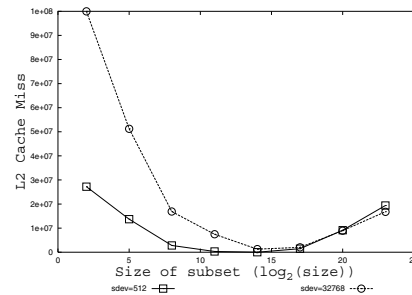
11

Our results show that our approach is quite effective. When sorting data inputs of 12M keys with various standard deviations, our adaptive approach selected the best algorithm in all the cases for the four platforms that we tried. The wrong decision could have introduced a performance degradation of 133% in the worst case and an average of 44%.
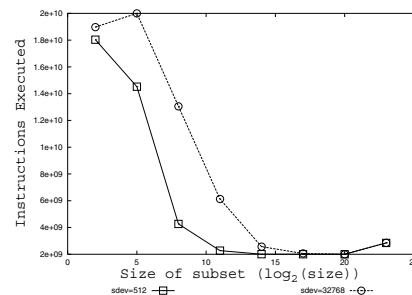
References

[1] *TPC Benchmark H*. Transaction Processing Performance Council (TPC), 2002.

[2] R. Allan and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishing, 2002.

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

[4] 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.

[5] P. Kisubi, P. Knijnenburg, and M. O'Boyle. The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, pages 237–246, 2000.

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

[7] A. LaMarca and R. Ladner. The Influence of Caches on the Performance of Sorting. In *Proceeding of the ACM/SIAM Symposium on Discrete Algorithms*, pages 370–379, January 1997.

[8] J. Larriba-Pey, D. Jiménez-González, and J. Navarro. An Analysis of Superscalar Sorting Algorithms on an R8000 processor. In *International Conference of the Chilean Computer Science Society*, pages 125–134, November 1997.

[9] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[10] 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.

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

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

[13] 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.

[14] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A Comparison of Empirical and Model-driven Optimization. In *Proc. of Programing Language Design and Implementation*, pages 63–76, June 2003.
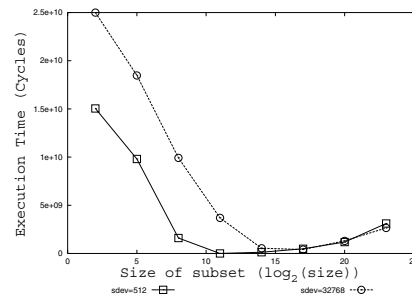
(a) L1 cache misses versus size of the subset.

(b) L2 cache misses versus size of the subset.

(c) Number of instructions executed versus size of the subset.

(d) Cycles executed versus size of the subset.

Figure 9. Effect of varying the size of the subset when using multiway merge to sort 12M records on SGI R12000.In the plots, sdev stands for Standard Deviation. The X axis uses a logarithmic scale.