© 2006 by Xiaoming Li. All rights reserved.

MACHINE LEARNING TECHNIQUES FOR CODE GENERATION AND OPTIMIZATION

BY

XIAOMING LI

B.S., Nanjing University, 1998
M.E., Nanjing University, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

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 thesis we study machine learning techniques that extend empirical search to the generation of algorithms whose performance depends on both the input characteristics and the architecture of the target machine. More specially, we target our study on sorting and recursive matrix-matrix multiplication, which are two fundamental algorithm problems.

We observe that various sorting algorithms perform differently depending on input characteristics. We first study if it is possible to predict and select the best sorting algorithm for a specific input. We develop a machine-learning based technique to find the mapping from architectural features and input characteristics to the selection of best algorithm. The mapping is used at runtime to make selection of sorting algorithms. Experiments show that our approach always predict the best sorting algorithm and the runtime overhead due to the selection is below 5%.

Built the first study that selects a "pure" sorting algorithm at the outset of the computation as a function of the input characteristics, we develop 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 thesis 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.

We follow a similar approach and use a classifier learning system to generate high performance libraries for matrix-matrix multiplication. Our library generator produces matrix multiplication routines that use recursive layouts

and several levels of tiling. Our approach is to use a classifier learning system to search in the space of the different ways to partition the input matrices the one that performs the best. As a result, our system will determine the number of levels of tiling and tile size for each level depending on the target platform and the dimensions of the input matrices.

To Hui and Parents.

Acknowledgments

I am greatly indebted to two high-school teachers because their guidance and encouragements changed how I looked at myself. Mr. Zhidong Wang, my physics advisor in junior high school, was the first one who made me believe I could do something related to intelligence. I never forget his encouragements. Mr. Changfan Zhen, my physics advisor in high school, taught me an important lesson of thinking - learn from error. Also I learned from him how to appreciate this world from more than one perspective.

Prof. David Padua, my advisor, has guided me through my graduate study. I particularly appreciate his foresight into research problems, which I hope I have stolen some from him. Moreover, I have learned from him how to make vague ideas into solid research. I will remember his meticulous demand for clear understanding in my future career.

María Jesús Garzarán has contributed to the ideas behind this thesis. She has also helped me making a better presentation of work. My thesis could not be in today's shape without her dedication.

Many thanks go to Prof. Keshav Pingali and Prof. Darko Marinov. It has been rewarding experience working with them.

I would also like to thank the members of my prelim and final committee, Prof. Marc Snir, Prof. Gerald DeJong, Prof. Luddy Hurrison and Prof. Keshav Pingali. I got valuable suggestions from the committee.

This thesis is dedicated to my parents and my wife Hui. My mother raised me up in a difficult condition and determined to give me good education. Hui has always been at my side and encouraging. I could have never made it this far without her.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ATLAS	Automatically Tuned Linear Algebra Software.
BLAS	Basic Linear Algebra Subroutines.
FFT	Fast-Fourier Transform.
MMM	Matrix-matrix multiplication.

Chapter 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.

Significant gaps exist between the performance of manual tuning and compiler optimization. Figure 1.1 compares the performance of compiler optimized matrix-matrix multiplication and that of hand tuned version by a human expert on Intel Xeon platform. The x-axis is matrix size. The y-axis is MFLOPS. Matrix-matrix multiplication stands as one of the most important computation routines and is widely used as a benchmark to measure the efficiency of compilers. Extensive efforts have been made by the compiler community for compiler optimizations that are effective on matrix-matrix multiplication kernels. However, we see a difference about 60 times between what the best compiler can deliver and what a human expert achieves. We also observe similar results of comparison for other important computation routines, such as Fast Fourier Transform (FFT).

1.1 Library generators

An emerging approach that has proven quite effective in overcoming both of these limitations is to use library generators. Library generation is an emerging technique that automatically produces efficient implementations across a wide range of platforms. It successfully competes with the expensive manual library tuning process and delivers expert-level performance. Library generation has attracted increasing interest from both hardware and software vendors. 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. Compiler research also benefit from the study of library generators. Library generation techniques provide insights on how to build useful models in compilers to optimize performance. Moreover, library generators show how semantic knowledge, which is usually missing in compilers, reduces the gap between the performance of compiler generated code and that of expert written

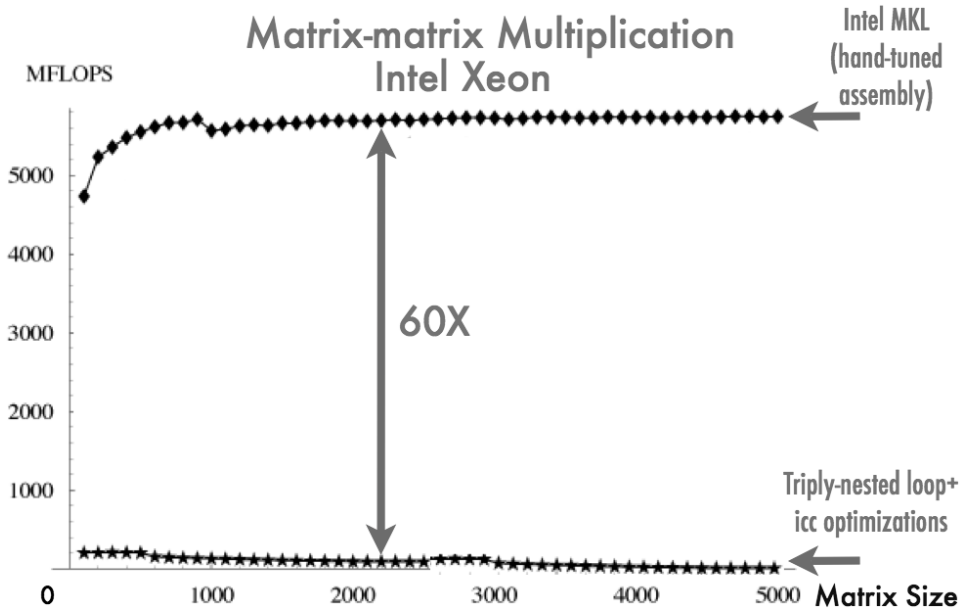


Figure 1.1. Manual tuning vs. compiler optimization

code.

Library generators accept a high level description of the problem. The code generator can synthesize new versions of code from the high-level description. Every such version is executed on the target machine. Architectural features impact the performance of the execution. The code generator will select the version with the best performance and use that version as the implementation of the problem on the target machine. ATLAS [66], PHiPAC [15], FFTW [26] and SPIRAL [69] are among the best known library generators. We use ATLAS and SPIRAL as two examples of how conventional library generators generate efficient code. ATLAS and PHiPAC generate linear algebra routines and focus the optimization process on the implementation of matrix-matrix multiplication. ATLAS has (i) a module that performs empirical search to determine certain parameter values, and (ii) a module that generates code, given these values. 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 algorithm generated using the found parameter values is then compiled and used as the kernel of the Level-3 Basic Linear Algebra Subroutines (BLAS-3).

The other two systems mentioned above, SPIRAL and FFTW, generate signal processing libraries. A signal processing transform can be represented by many different, but mathematically equivalent, formulas. The programs implementing these transforms have different running times. Since the number of formulas can be quite large, exhaustive search is usually not possible. SPIRAL applies a mathematical notation that can represent a wide range of alternative

implementations of a signal processing transformation. The mathematical notation defines the space from where the optimal solution is sought. However, the search space in SPIRAL is too large for exhaustive search to be possible. Thus, these systems search using heuristics such as dynamic programming [26, 33], or genetic algorithms [62], and must therefore accept sub-optimal solutions.

1.2 Contribution

Current systems can not generate code for problems whose performance depends on input characteristics. For instance, the performance of matrix-matrix multiplication does not depend on the input, does not depend on the values that we are multiplying. Thus, library generators, like ATLAS, never took into account input characteristics when generating code. However, we need to learn how to generate code for those problems where the performance depends on the input. The reason is that for such problems, we need to determine a mapping from input characteristics onto performance, which is much more complex than determining a single efficient implementation.

The major goal of this research is to develop techniques that can automatically generate efficient code for problems whose performance depends on input characteristics. We target our research on two problems, sorting and recursive matrix multiplication. Sorting is, without doubt, one of the most fundamental algorithmic problems. It is the core of many applications, such as indexing in database. Sorting has been studied for decades. It attracts research attentions because it is complex to sort effectively while sorting problem can be described in a simple form. The theoretical complexities of sorting is well understood for some input distributions. However, we find that theoretical complexities of sorting algorithms are only partial solutions. There are two reasons why we can not develop efficient sorting algorithms based only on theoretical complexities. First, the theoretical complexity of sorting algorithms has been studied mainly using specific input distributions, including uniform distribution [35] and nearly sorted inputs [23]. The input distribution of sorting problems is not always uniform, for example, database benchmarks [4] feature Normal distribution and Exponential distribution. The distribution impacts the performance of sorting algorithms. It is impossible to derive theoretical complexities for every possible input distribution. The second reason is that theoretical complexity does not represent adequately all the factors that impacts the performance of sorting algorithms on current architectures. When we design efficient sorting algorithms, we need to consider factors, such as cache locality [39] and number of data movements [9].

The other target problem is matrix-matrix multiplication based on recursive layout. Like sorting, matrix-matrix multiplication is another one of the most fundamental algorithmic problems. The input of matrix-matrix multiplication is usually row-major or column-major matrices. Recursive matrix layout organizes matrix elements into hierarchical recursive blocks so that blocks at every level of recursion reside in continuous memory locations [58]. It has been

shown that recursive layout of matrix improves the cache locality of matrix-matrix multiplication naturally and hence accelerates the computation [17, 27]. However, the best recursive layout of matrix-matrix multiplication for specific matrices depends on both the dimensions of the input matrices and the architectural features of the underlying machine. Hence, the search for the best recursive matrix-matrix multiplication differs from the search in ATLAS in that the search should take into account input values. To our knowledge, no previous work addresses the problem of how to find the best recursive layouts for different recursive matrix-matrix multiplication cases.

This thesis presents novel machine-learning based techniques that address two major challenges in code generation: how to adapt code to input characteristics and how to find the most efficient form of code from a large search space. This research follows two natural steps. First we study how to select the best algorithm dynamically for various input characteristics from a small set of predefined candidates. We focus on sorting in this step. Second, we study how to build the most efficient hybrid code by combining different algorithms from bottom up, again, for different input characteristics and architectural features. We use sorting and matrix-matrix multiplication as our target problems.

We first explore the problem of generating high-quality sorting routines. As noted above, the 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 factors such as the distribution of the data to be sorted. In fact, as discussed below, merge sort performs better on some classes of input data sets than radix sort 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, at runtime we use the distribution of the input data to select a sorting algorithm. This approach has proven quite effective, though the final performance is limited by the performance of the sorting algorithms - multiway merge sort, quicksort and radix sort are the choices in [43] - that can be selected at run time. In this thesis we study machine learning techniques to extend empirical search to the generation of sorting routines. We first study if it is possible to predict and select the best sorting algorithm for a specific input because that various sorting algorithms perform differently depending on input characteristics. We develop a machine-learning based technique to find the mapping from architectural features and input characteristics to the selection of best algorithm. The mapping is used at runtime to make selection of sorting algorithms. Experiments show that our approach always predict the best sorting algorithm and the runtime overhead due to the selection is below 5%.

Build on the previous study that selects a "pure" sorting algorithm at the outset of the computation as a function of the input characteristics, we develop 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 thesis 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.

We follow an approach similar to that described in the previous paragraph for soring and use a classifier learning system to generate high performance libraries for recursive matrix-matrix multiplication. Our library generator produces matrix multiplication routines that use recursive layouts and several levels of tiling. Our approach is to use a classifier learning system to search in the space of the different ways to partition the input matrices the one that performs the best. As a result, our system determines the number of levels of tiling and tile size for each level depending on the target platform and the dimensions of the input matrices.

Experiment results show that our machine-learning techniques are very effective. Our adaptive sorting accurately selects the best sorting algorithm at runtime from Quicksort, Radix Sort and Multiway Merge Sort. Furthermore, The best hybrid sorting 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. Our recursive matrix-matrix multiplication library automatically generate highly efficient routines that achieve comparable performance as that of routines found by ATLAS.

1.3 Organization of this thesis

The rest of this thesis is organized as follows. Chapter 2 discusses how we use machine learning techniques to generate sorting algorithms that can select the best from a small set of "pure" sorting algorithms and adapt to input characteristics and architectural features. Chapter 3 discusses our work that how machine learning techniques guide the building of highly efficient hybrid sorting algorithms. Chapter 4 describes our application of machine learning techniques in learning the best recursive layouts for various matrix-matrix multiplication cases and different architectural features. Finally, Chapter 5 concludes this thesis and discuss possible future research directions on applying machine learning techniques to code generators.

Chapter 2

Code Selection

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 chapter, 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* [34, 66] 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 [66], FFTW [26], and SPIRAL [69]. 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.

Selecting the most appropriate algorithm for different instances of inputs have always been an important goal for algorithm designers. Rice proposed the first framework for algorithm selection and provided several general principals of selection in [55]. Our work on sorting is related to Rice's framework in that we also try to determine the input characteristics that affect the performance of sorting algorithms and make a selection based on that characteristics.

In this project, we dynamically select the best sorting algorithm. We faced two main difficulties. One was the

lack of a precise formulation of the tradeoffs between number of operations and the impact of memory hierarchy on the performance of the different sorting algorithms. The performance of sorting algorithms has usually been studied with number of operations that each algorithm executes. However, the memory hierarchy has a significant impact on performance. Though there are theoretical analysis about the impact of memory hierarchy on sorting [7, 8], the tradeoff of these two factors when we change algorithm parameters is difficult to formulate. 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 machine 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 [32]. 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 chapter is organized as follows. In Section 2.1, we introduce several sorting algorithms including a fast radix-based sorting algorithm and our memory hierarchy conscious sorting algorithm. Section 2.2 presents the factors that affect the performance of several of the sorting algorithms discussed in Section 2.1. In Section 2.3, we discuss the installation phase of our library generator and the runtime mechanism we use to select one of the algorithm candidates. In Section 2.4, we present our experimental results. Finally, concluding remarks are given in Section 2.5.

2.1 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 [35]. 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 [32, 39, 41]. The algorithms resulting from these improvements have been called *cache-conscious*.

As mentioned in the introduction, the main focus of this thesis 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 2.2.

Figure 2.1 illustrates the type of studies conducted in the past to compare sorting algorithms. Figure 2.1 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*). They keys are generated using a random generator [3], so the value of a key is independent of the value of a another key. Each key is rounded from a Uniform/Normal/Exponential distribution. 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. For each point we repeat the same experiment three times and use the average. 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 2.2, where the execution time is plotted when we change the standard deviations of Normal distributed inputs. Results are shown for two platforms:Intel Pentium III Xeon and Sun UltraSparc III. For each platform, 2M (graphs on the left column) and 16M (graphs on the right column) keys are sorted. As Figure 2.2 shows, the standard deviation and the number of keys to sort have a significant impact on the relative performance of the different sorting algorithms. If we analyze why the standard deviation affects the performance of those algorithms, we find that it is actually the number of distinct input values that distinguish the performance of comparison-based sorting algorithms and the frequency of each digit value that decide the performance of CC-Radix. We will discuss the relationship between these input characteristics and the performance of sorting in more detail in the next sections of this chapter. What motivates our research is that we can see clearly from Figure 2.1 that evaluating the performance of the different algorithms as a function only of the number of keys usually leads to the conclusion that a particular algorithm is the best across the board. That approach does not take into account that, as Figure 2.2 shows, the relative performance of the different sorting algorithms also depends on the number of distinct input values and frequency of digit values, which is controlled by the standard deviation of the input data of the Normal distribution that we use to generate the keys. Furthermore, 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 2.2, the execution time of CC-radix sort decreases as the standard deviation increases. However, the crossover-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.

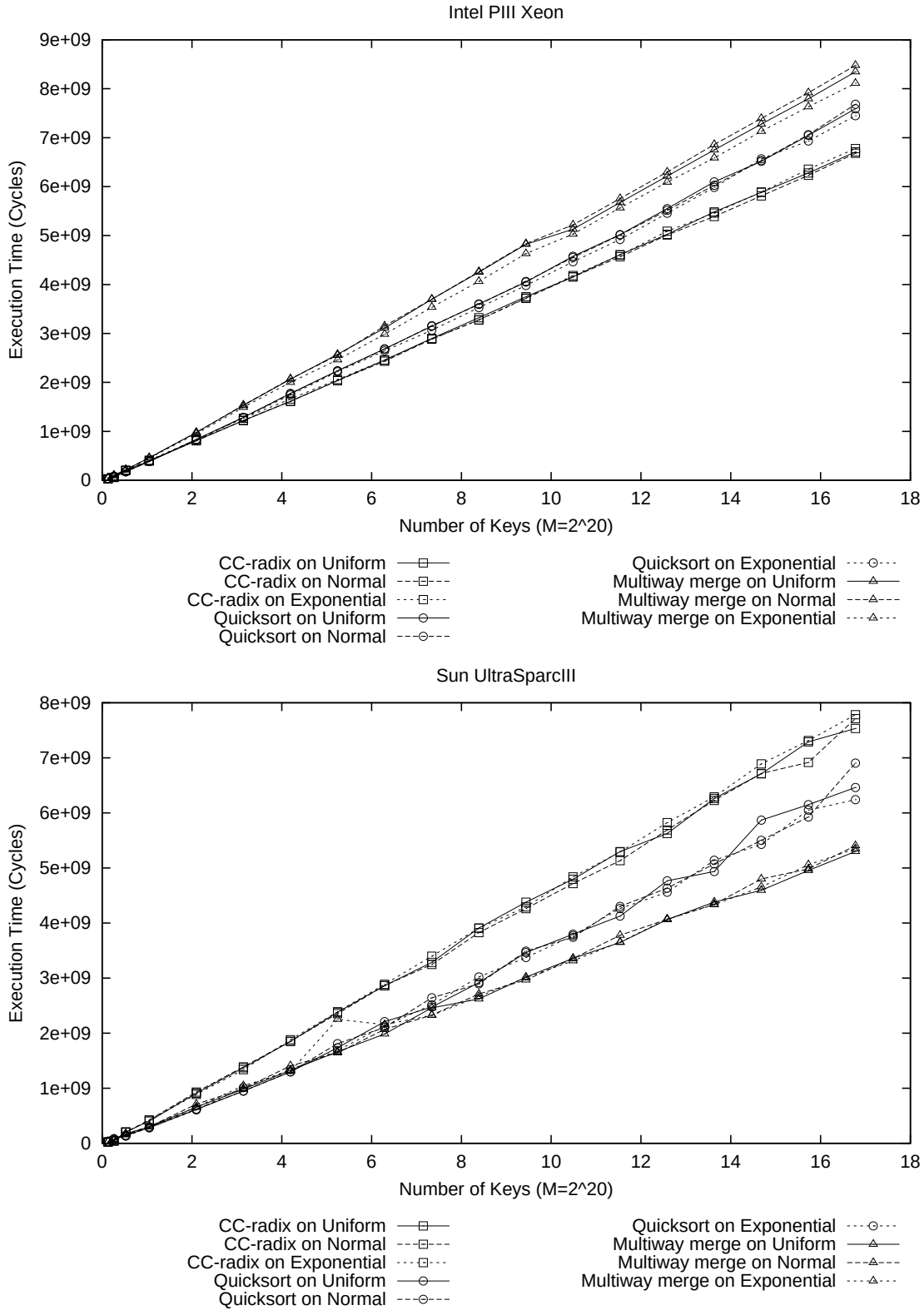


Figure 2.1. Effect of the distribution and number of keys on the performance of sorting algorithms

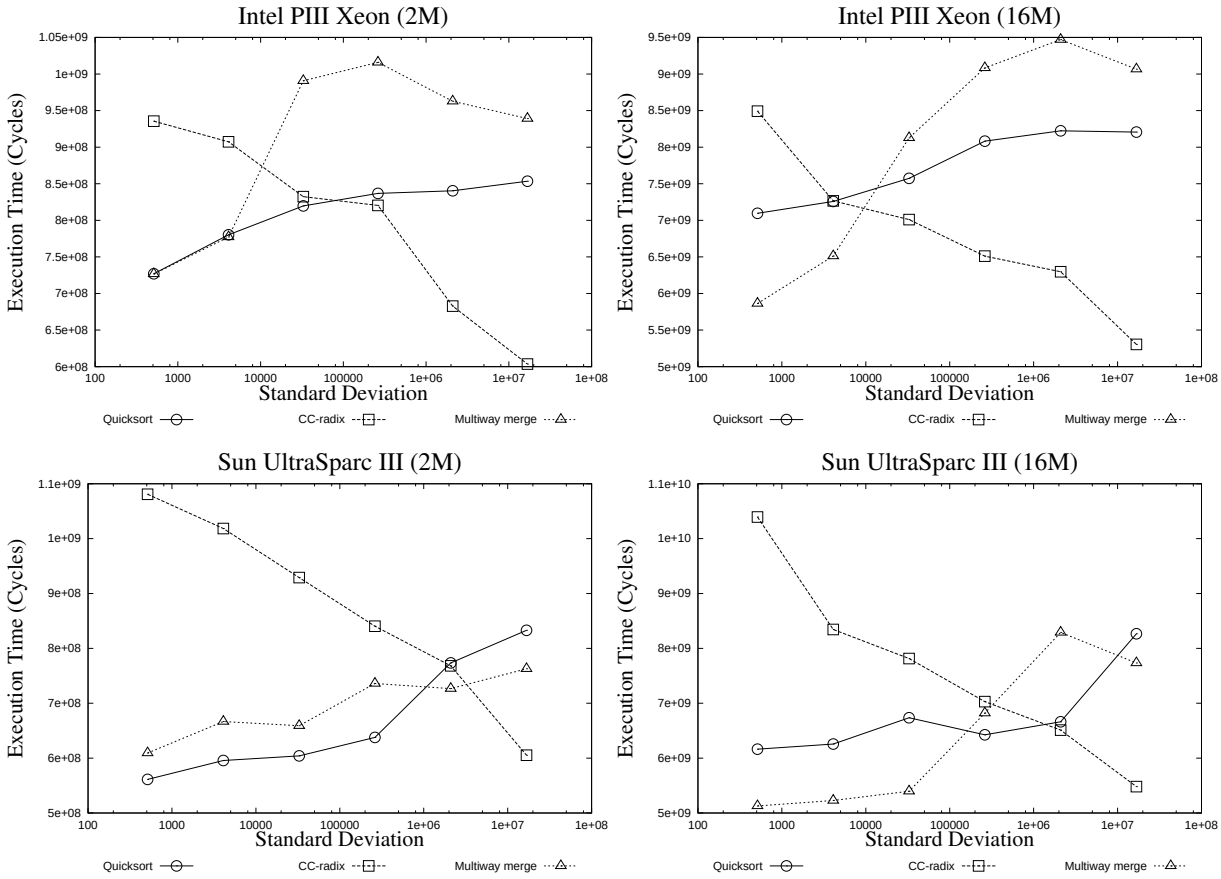


Figure 2.2. Effect of the standard deviation on the performance of the sorting algorithms. Left: 2M keys. Right: 16M keys.

We have used very tuned implementations of several sorting algorithms. In all the cases, the sorting algorithms that we use can be classified into two categories: comparison-based and radix-based.

Comparison-based sorting algorithms, such as Quicksort, Mergesort, compare the data elements as a whole. The lowest possible complexity of comparison-based sorting algorithms to sort n elements is $O(n \log(n))$.

Radix-based sorting algorithms set up the order among the input data array by comparing the digits, from the most significant position to the least significant position, of every element. The complexity of radix sort is linear in the product of the number of inputs and the number of digits (radices). This often results in fewer instructions than for comparison sort; also the code has fewer branches and ofter executes faster.

Our sorting library is based on Quicksort, Mergesort and a fast Radixsort CC-Radix Sort (Cache-Conscious Sort) [32]. Quicksort is supposed to have the best average performance. It is very simple and has good data locality. Mergesort is the traditional solution to the problem of external sorting. Mergesort can easily control the size of the active data set. This feature is useful to reduce the number of I/O operation for the external sorting and the cache misses for the problem of cache conscious sorting. CC-Radix Sort (Cache-Conscious Sort) [32] is a fast radix-based sorting algorithm optimized for reducing cache misses.

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.

2.1.1 Quicksort

Quicksort is an in-place divide-and-conquer algorithm. The algorithm is based on a partitioning 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 algorithm recursively works on the region to the left of the pivot and on the region to its right.

Sedgewick [59] 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 $O(N^2)$.

Sorting the whole set of the elements at the end eliminates the overhead of calling the Insertion Sort procedure during the recursion, and as a result reduces the number of executed instructions. However, sorting these small partitions as they appear during the recursion procedure, may reduce the data cache misses because the elements should be in the cache, since they have just been part of a recent partition. It is not clear which of these two strategies results in a lower execution time. Thus, in our experiments we try both of them. In addition, we also try the sorting network algorithm described in Section 2.1.5 when these small partitions are found. The pseudo-code for our implementation of Quicksort is illustrated in Figure 2.3.

```
Quicksort(bucket)

if below_threshold (bucket) then
    Register_sorting_network (bucket)
else
    Select the median of three randomly selected elements as pivot
    sub-buckets = Sort_around_pivot (bucket)
    for each sub-bucket in sub-buckets
        Quicksort(sub-buckets)
    endfor
endif
```

Figure 2.3. Pseudocode for Quicksort

2.1.2 A Cache-Conscious Radix Sort

Radix sort is the most important non-comparison algorithm [32]. 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 *i*th phase sorts the key on the value of the *i*th 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 [32]. To sort the keys in each phase, their implementation relies on a counting algorithm [35] that proceeds in three steps for each phase. First, a vector containing the histogram of the number of records for each digit value is computed. Thus, during phase *i*, the first step computes vector element $v(j)$ which contains the number of keys whose *i*th 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 copies 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 copied 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, for fixed number of digits, b. 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 [32] have proposed a cache-conscious radix sort (CC-radix sort) algorithm shown in Figure 2.4.

```
CC-radix(bucket)

if Below_register_threshold (bucket) then
    InsertSort (bucket)
else
if fits_in_cache (bucket) then
    Normal_Radix_sort (bucket)
else
    sub-buckets = Reverse_radix_sorting(bucket)
    for each sub-bucket in sub-buckets
        CC-radix(sub-buckets)
    endfor
endif
```

Figure 2.4. Pseudocode for CC-radix

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 lowest level cache. If they do, a simple radix sort algorithm is used. If, however, the data structures do not fit in the lowest level cache, the algorithm partitions the bucket into sub-buckets using *reverse radix sorting*. In reverse radix sorting, a bucket is partitioned applying the highest order digit, with variable digit radix, of the key that has not been used yet to partition the data. Keys in each sub-bucket are later sorted using a simple

radix sort to 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 that 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 set of memory locations that can be represented at a given time in the TLB, and that all the 2^r buckets are simultaneously accessed. The 2^r buckets and the original sequence are accessed sequentially. The $2^r + 1$ locations can be far apart in memory. In this case, TLB need to hold the $2^r + 1$ page entries. 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.

The reason for this constraint is that if the number of elements of a bucket is larger than the set of memory locations that the TLB can map, and the number of the resulting sub-buckets is larger than the number of TLB entries, the number of TLB misses could be high. More details about CC-radix sort can be found in [32]. For our experiments we used an implementation of CC-radix sort generously provided by the authors of [32]. Also, the values of r in our experiments was that assumed by that implementation.

2.1.3 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 [35]. At the beginning, the leaves in the heap are the first elements of all the sorted 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 leave. 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 2.5 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. We use NB to denote the heap size. In our implementation we use only $NB - 1$ tuples for the heap. The reason is that the tuples in the leaves are always the top elements of each sorted subset, so there is no need place them in the heap. 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 the cache are the

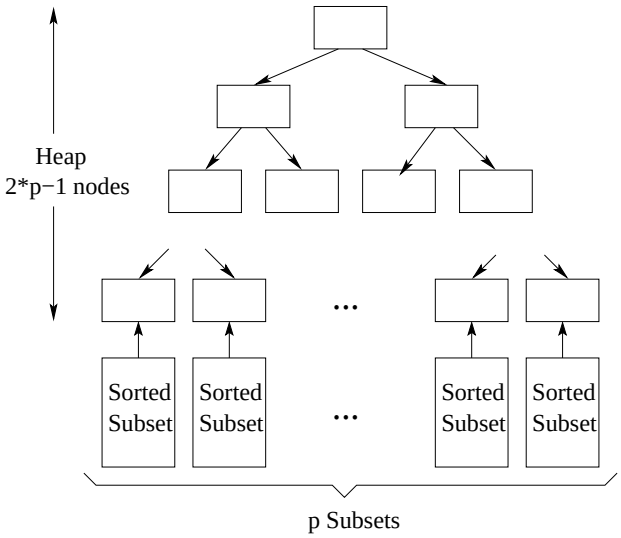


Figure 2.5. Multiway merge sort.

records in each subset to be sorted. 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 $N * log_2(2 * p - 1)$.

Our memory-hierarchy conscious Multiway Merge Sorting algorithm (Memsort) applies different optimizations for the cache level and the register level. The purpose of these optimization is (1) when the element is still required for sorting, it is in caches or in registers; (2) when the order of the element has been determined, it will be evicted. Figure 2.6 shows the pseudo-code of Memsort optimizing for L1 cache and L2 cache.

```
MEMSORT(src, dest, n, NB) :
  ; src:   the input data array
  ; dest:  the array to store the sorted data
  ; n:     the length of src and dest
  ; NB:    the size of the heap (the number of partitions)
  Partition src evenly into NB subsets;
  Sort each subset;
  Construct a heap (priority queue) with size NB;
  Input the first element of each subset into heap;
  While not all elements are sorted
      Remove the root of the heap to dest;
      Add the next element in the subset from which
          the original root comes from into the heap;
```

Figure 2.6. Memsort

At one level of memory hierarchy, Memsort works as in Figure 2.5. The input data array is partitioned into n subsets. Each subset will be sorted either by applying Memsort recursively or, if at the lowest level, Quicksort or CC-radix sort. The key in Memsort is to use a heap (priority queue) to merge sorted subset (call "run"). By choosing the appropriate heap size (NB), we can control the number of active elements currently used in the cache. It is clear the root of the heap is the smallest(biggest) element of all the subsets. So after it is stored into the $dest$ array, it will never be needed again for this level. As long as the next element of the same queue as the root is inputed into the heap, this property can be maintained. So when we need to compare an element to merge, it is in the heap. When we don't need it, it is stored into the destination array.

2.1.4 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 $O(N^2)$

2.1.5 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 not implemented as a hardware device, but instead it is implemented in software by performing in sequence each physical layer of the network on processor registers. This algorithm can perform better than insertion sort since practical 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 appropriate to handle small partitions. In fact, sorting networks can be used as part of loop unrolling transformations. The reason is that, when the number of keys is small, we can generate a fully unrolled sequence code corresponding to the sorting network where the data is stored in processor registers. The sequence has no nested branches, so that operations in the sequence can be scheduled better. Figure 2.7-(a) shows a diagram of a sorting networks similar to those used by Knuth [35], and Figure 2.7-(b) shows the corresponding code to sort eight elements.

At the lowest level of memory hierarchy, the registers, Quicksort and Memsort use a different optimization. We already know the recursive nature of Quicksort makes it not very efficient for small data sets. Therefore Hoare suggested a more efficient method be used when the data set is small[31]. Sedgewick recommended to use insertion

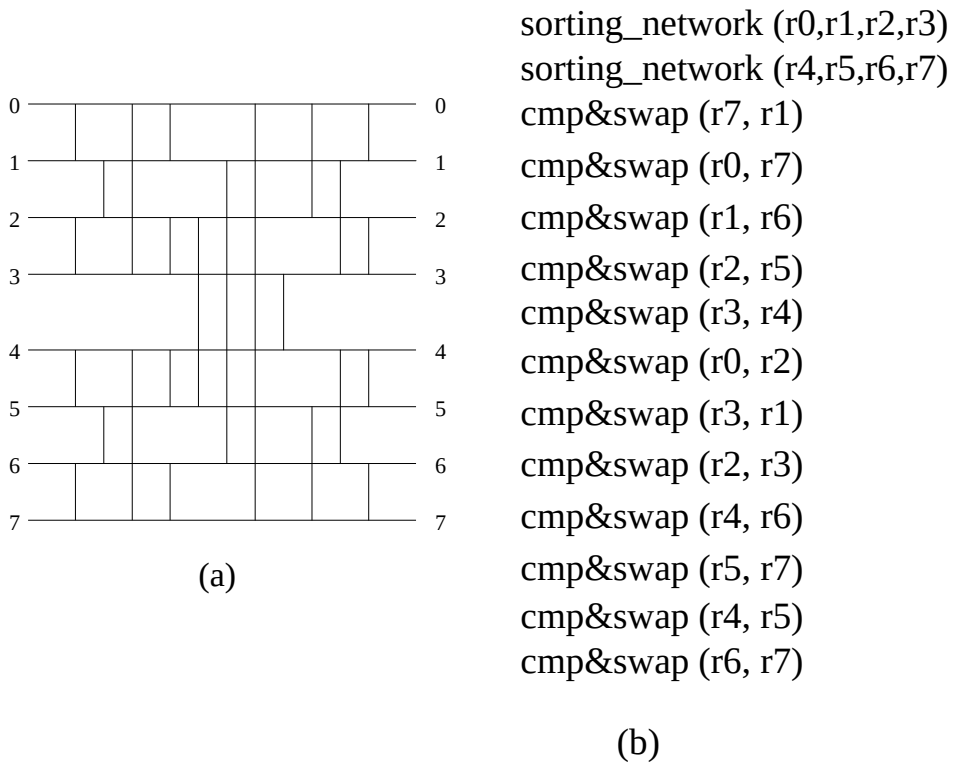


Figure 2.7. Sorting Network.(a)-Diagram of a sorting network. (b)-Code corresponding to (a).

sorting, which is known to be very efficient for small files[59]. Can we do better? The idea to treat small data set differently is similar to the compiler technology of register blocking. We can apply a similar technology here that a limited number of elements can be stored in registers, sorted and then store back to the original array. For register sorting, we can achieve a better performance than Insertion Sort, whose worse case is $O(n^2)$. Sorting network is known to have a fixed $O(n \log n)$ complexity. Though there are other sorting algorithms with worst case equal to $O(n \log n)$, e.g., Heapsort, Sorting network is the simplest, therefore, sorting network is more appropriate to handle the small data set. The instructions executed by sorting network are regular. When the number of elements is small, we can generate the sequential sorting network without any loop, that is, fully unroll sorting network. Figure 2.8.a is a generated sequential sorting network to sort eight elements.

By unrolling the sorting network fully, we expose more opportunities for compiler optimizations. A scheduling technology may help compilers a bit further. Without violating the dependence between instructions of the sorting network, the sorting network generator can rearrange the instructions so that the next instruction using the same register(s) will be lat instructions away. This directive scheduling may help compiler to hide the branch delay, though it is highly platform dependent and compiler dependent.

Table 2.4 shows briefly the performance difference between the Quicksort using Insert sort and the Quicksort using Sequential Sorting Network on three platforms. The values are the seconds to sorting 4 million and 16 million 32-bit

; CMPSWAP(a,b) compares elements a and b. After the execution,
; $a = \min(a, b)$, $b = \max(a, b)$.

```
CMPSWAP(s0, s1);
CMPSWAP(s2, s3);
CMPSWAP(s0, s3);
CMPSWAP(s1, s2);
CMPSWAP(s0, s1);
CMPSWAP(s2, s3);
CMPSWAP(s4, s5);
CMPSWAP(s6, s7);
CMPSWAP(s4, s7);
CMPSWAP(s5, s6);
CMPSWAP(s4, s5);
CMPSWAP(s6, s7);
CMPSWAP(s0, s7);
CMPSWAP(s1, s6);
CMPSWAP(s2, s5);
CMPSWAP(s3, s4);
CMPSWAP(s0, s2);
CMPSWAP(s1, s3);
CMPSWAP(s0, s1);
CMPSWAP(s2, s3);
CMPSWAP(s4, s6);
CMPSWAP(s5, s7);
CMPSWAP(s4, s5);
CMPSWAP(s6, s7);
```

(a) lat=0

```
CMPSWAP(s0, s1);
CMPSWAP(s2, s3);
CMPSWAP(s4, s5);
CMPSWAP(s0, s3);
CMPSWAP(s6, s7);
CMPSWAP(s1, s2);
CMPSWAP(s5, s6);
CMPSWAP(s4, s7);
CMPSWAP(s0, s1);
CMPSWAP(s2, s3);
CMPSWAP(s6, s7);
CMPSWAP(s4, s5);
CMPSWAP(s0, s7);
CMPSWAP(s1, s6);
CMPSWAP(s3, s4);
CMPSWAP(s2, s5);
CMPSWAP(s4, s6);
CMPSWAP(s1, s3);
CMPSWAP(s5, s7);
CMPSWAP(s0, s2);
CMPSWAP(s6, s7);
CMPSWAP(s4, s5);
CMPSWAP(s2, s3);
CMPSWAP(s0, s1);
```

(b) lat=1

Figure 2.8. Sequential sorting networks with different latencies

integers with the two Quicksorts. We can see there is always a $3\% \sim 5\%$ gaps between the two methods. This result proves that our Sequential Sorting Network sort small data set faster than Insert Sort. Considering the two versions of Quicksort are same except the lowest level, plus the Insert sort and the Sequential Sorting Network are just used to sort less than 20 elements for each invocation, the consistent improvement from Insert sort to Sequential Sorting Network is satisfying.

2.2 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. For example, the size of the heap in the multiway merge sorting algorithm can be chosen so that it fits in the cache. 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 not well understood. 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 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 [71], but the development of an automatic method for sorting remains an open problem.

However, when generating a library of sorting algorithms, one may think that the the parameters of the sorting algorithms that depend on the architectural characteristics like cache size can be tuned at installation time, while the selection of the appropriate algorithm needs to be delayed until the characteristics of the data to be sorted are known. This is true in many cases, but not in all cases. Thus, at installation time, our library generator will run some experiments, and empirically will search for the parameter values of each algorithm that result in the best performance. Those values that only depend on the architectural parameters will be set up at install time. The results of these experiments will be used as feedback information at run time, when deciding the sorting algorithm to execute

and the most appropriate parameters for that algorithm. In this Section, we explain the main important factors that affect the performance of the sorting algorithms described in Section 2.1, and we outline why some parameters that depend on architectural characteristics of the machine needs to be chosen at run time.

2.2.1 Architectural Factors

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

Size of the lowest level cache

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 [11].

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 the way tiling is expressed in 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 [39, 40] 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. We implement the multi-Quicksort algorithms proposed in [39], and the results 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. For that reason, we solely use Quicksort for the experiments in this chapter.

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 [39]. We consider 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. Thus, when installing our sorting library

we empirically try both options and will decide which is the best one for each particular platform. We will also search for a threshold to decide when insersort should be applied.

Lagoudakis [36, 37] uses a Markov Decision Process (MDP) to model recursions in several sorting algorithms, so that an appropriate cut-off point between different algorithms can be selected. That approach successfully finds better cut-off of input size between insertion sort and quicksort than that found using the empirical search they implement. However, the work of Lagoudaki's only tries on very small inputs (input size < 100), and they do not verify empirically that the cut-off point found by their learning technique is indeed the best. Lagoudakis's approach does not take into account architectural features or input distribution.

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, the reverse radix sort phase of 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. Notice that CC-radix must keep in the cache, the source vector, the destination vector, and also the counters that record locations of buckets.

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 also be applied recursively, but we did not consider this freedom of the algorithm for the experiments reported in this chapter.

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 depend on the size of the cache but also on the characteristics of the input data. The reason is that for some input data sets the best performance will be obtained for small values of p, which implies small heap sizes, while most of the work is done by radix sort or Quicksort. For other input data sets the best performance will be obtained for large values of p. Thus p is a parameter whose value needs to be decided at runtime when the characteristics of the data to sort are known. However, at installation time we need to gather information that will be used as feedback information to decide at runtime.

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 [66] or just by evaluating an expression involving the cache size [71]. However, as we have just outlined,this cannot be done for sorting algorithms because their performance depends on the characteristics of the input data. The performance of the sorting algorithms will also depend on the characteristics of the input data, and, as a result, the tile size, or the optimal value for the parameters in our case, will change depending on the input data.

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 2.7-(b) shows the resulting code when register tiling is applied to a sorting network algorithm that sorts eight elements. Figure 2.7-(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 is executed when a partition is smaller than a certain threshold. This threshold depends on the number of registers that are available to the program. 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. We examine the assembly code generated by the background compiler for our library, and we find that the compiler does generate instructions that use the special stack registers. In this case, compare and exchange operations in our library are translated into the stack instructions. Thus, the performance of the stack, instead of the number of registers, will determine the performance.

Table 2.1 shows how the threshold that determines when to use register sorting network affects the sorting performance of integers. For two of the three platforms, SGI R12K and Intel PIII Xeon, different rb's don't make much difference. We can see any value from 8 to 24 performs similarly well. For Sun UltraSparcIII, $rb = 8$ is the prominent

	SGI R12k		Sun UltraSparcIII		Intel PIII Xeon	
	4M	16M	4M	16M	4M	16M
rb = 8	2.266s	10.558s	1.352s	6.273s	2.124s	9.530s
rb = 16	2.264s	10.543s	1.368s	6.359s	2.103s	9.450s
rb = 24	2.259s	10.544s	1.804s	8.177s	2.188s	9.788s

Table 2.1. Performance vs. register blocking size. Numbers of elements are 4M and 16M.

best choice. This may related to the fact that Sun Compiler can only use 8 registers. If more registers are requested, unnecessary register spills will happen.

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 the elements in the cache line. In this way we exploit spatial locality. Sorting algorithms that scan the data like Insertion 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 only when writing the keys into the continuous locations of 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 2.9 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.

There are atomic operations for the heap: $shiftup$ and $shiftdown$. The heap is implemented as an array of nodes. All the child nodes of a parent node occupy continuous positions in the array. So to get the address of the parent node or to get the address of the child node is just a matter of arithmetic calculation. In $shiftup$, the basic action is to find the parent node of a child node. If the parent node is not in the cache, the cache line must be loaded. In $shiftdown$, every operation to move down a node needs to find the child node with the minimum key. Again, the child nodes needs to be loaded into the cache line if they are not present in the cache. If the number of node residing in the cache line is larger than the fanout of the heap, the cache line is under-utilized. For example, if cache line is 32 bytes, a node is 8 bytes. The fanout of the heap is 2. We load 32 bytes from the memory to the cache, but we only use $2 * 8 = 16$ bytes in a $shiftdown$ operation.

A natural optimization to make full use of the cache line is to fill as many as possible child nodes into a cache as in Figure 2.9, that is, make $fanout = \frac{\text{cache line size}}{\text{node size}}$. So in a $shiftdown$ operation, all the cache line can be used.

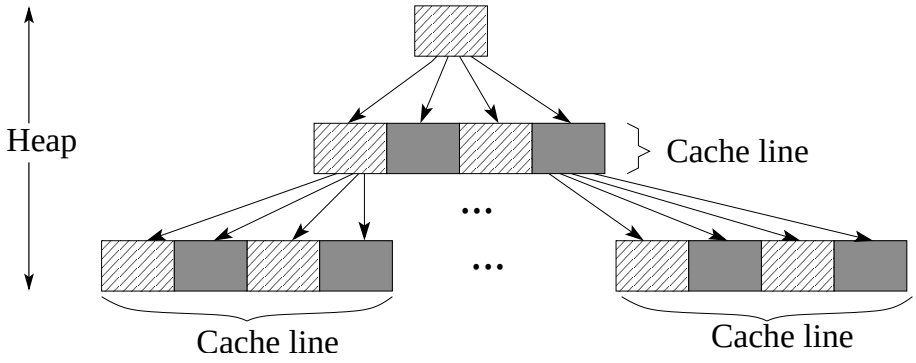


Figure 2.9. Fill as many as possible child nodes into a cache line.

2.2.2 Input Data Factors

In the past, the number of keys to be sorted has been typically used to select the algorithm to apply, and hence sorting selection techniques are evaluated with only different input sizes. In our work, we take into account other characteristics of the input data that are usually ignored before. Next, we discuss the characteristics of the input data that we have found to affect the performance of the sorting algorithms.

On the other hand, since different sorting algorithms perform better for different input data, then, why not build an algorithm that at runtime selects the best one among a series of sorting algorithms that are proved to be the best ones across a wide range of input data sets? The problem of this approach is that a simple quantitative description of the performance of the sorting algorithms does not provide enough information to choose the best algorithm at run time.

One of the statistical properties that determines the frequency of individual inputs is the *distribution*. Most of previous studies that have worked on sorting algorithms have only considered uniform distribution. However, other distributions are also possible. For example, in the TPC-H database benchmark [4] both the exponential and the uniform distributions are used to generate the database. Figure 2.1 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%. Thus, it seems that input properties determined by distribution are not useful to select the best algorithm.

Results in Figure 2.1 show that the characteristics of the input data to sort will determine the behavior of the different algorithms. In particular, it shows that considering a single factor such as number of keys to sort may result in wrong conclusions, since the best algorithm will also depend on other characteristics of the input data. However, we

need to identify the statistical properties of inputs that affect the performance of the different algorithms. In particular, we need those properties that help us to distinguish the performance of the comparison based algorithms versus the radix based ones.

Another statistical property that can determine some characteristics of the data is the standard deviation. This parameter measures the average distance from the mean of all the keys. A low value for standard deviation means that that values tend to be close to the mean, while a high value means that they are very spread out. Figure 2.2 shows the performance of Quicksort, CC-radix and multiway merge sort when applied to 2M and 16M keys with normal distribution and several values of standard deviation. To generate a set of keys for each particular value of standard deviation, we adjusted the random number generator to produce integers whose standard deviation is close to standard deviation. Since the Figure 2.1 showed that the performance did not depend on the distribution, we only show results for the normal distribution. The Figure shows that the relative performance of sorting algorithms do not change with input of different distributions. However, the relative performance follows similar pattern along the axis of standard deviation for different platforms.

We now discuss the reason why the performance of CC-radix depends on the value of standard deviation. CC-radix 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 evenly distributed between the buckets, fewer partitions will be necessary to fit the buckets in the cache. As we can see from the analysis, what really determines the performance of CC-radix is the distribution of digits, in particular the distribution of the most significant digits. We need a direct metric for such characteristics. Next we discuss in more detail how number of keys and the standard deviation affect the performance of sorting algorithms.

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.

Standard Deviation

Another property is the standard deviation. Figure 2.2 shows the effect of standard deviation in performance. The keys were generated using a Normal distribution and we average the performance of three runs on same input. The standard deviations are measured from input, though it is very close to the values we specify. 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.

Quicksort is a good algorithm when sorting inputs that fit into cache, because Quicksort is an in-place algorithm, and it can sort larger number of keys than the other two algorithms without increasing the data miss ratio. However, its performance decreases as the number of keys to sort increases and overflows the data cache. Thus, Quicksort is never the best algorithm when sorting large data sets. As the number of keys to sort increases, the complexity, and as result, the number of instructions executed increases. Also as the number of keys to sort increases, the data cache misses tend to increase. Thus, multiway merge is better than Quicksort for 16M keys and small values of the standard deviation. In principle, the performance of quicksort (and in general of any comparison algorithm) should not depend on the standard deviation. However, we noticed that the performance of quicksort varies as que change standard deviation. The reason is that for the Normal distribution that we use to generate the inputs, the variation of the standard deviation also controls the number of distinct values of the keys. In particular, when the standard deviation is low, the keys will mostly take a few number of values, since we are generating 32 bit integers. Smaller number of key values reduces the number of key swapping, which improves performance. A similar phenomenon also affects the performance of multiway merge sort, when sorting the keys in the heap.

Let us see why the performance of CC-radix sort depends on the value of standard deviation. CC-radix sort partitions the data of those buckets that do 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 these buckets fit into the cache. On the other hand, when values are distributed more evenly among the buckets, fewer partitions will be necessary to fit any of the buckets in the cache, and as a result CC-radix sort will perform better. Standard deviation is not a direct measurement of frequency of each digit values. As described in Section 2.3, we use entropy to measure such characteristics. Here, for the distribution we consider, increased standard deviation means increased entropy and increased entropy means better distribution across buckets. As a result, we use standard deviation to control the generation of inputs with different entropies, while it is entropy that is used to distinguish the performance of CC-radix.

Finally, besides the normal distribution that we used to generate the experiments in Figure 2.2, 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 affect 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 2.1 show.

However, in Figure 2.1 we show the execution time of the three algorithms as the standard deviation changes. We show results for two platforms. The plots on the left show results when sorting 2M keys, while the plots on the right show results for 16M keys.

From these results, it can be seen that the number of keys to sort may determine which is the best algorithm. For example, when sorting 2M, for small values of standard deviation Quicksort is the best algorithm in both platforms. However, for 16M data, Quicksort is never the best algorithm, and multiway merge is the best algorithm for small values of standard deviation. Also, as the standard deviation, increases, CC-radix is always the best algorithm independently of the number of keys to sort.

Also, notice that as the number of keys to sort increases, the complexity, and, as result, the number of instructions executed increases. Also as the number of keys to sort increases, the data cache misses tend to increase.

Figure 2.1 shows the performance of CC-radix, Quicksort, and multiway merge sort as the amount of data to sort changes from 128K to 16M, and the standard deviation and the distribution are constant. Results are shown for two platforms when sorting 32 bit integers. It can be seen that the relative performance of the different algorithms does not change for any of the two platform shown. Results using other platforms, and other standard deviation have resulted in the same conclusions. (Standard deviation determine which is the best algorithm, but the relative performance of the algorithms stays the same as the number of keys increase). Only when the number of items to be sorted is small, several algorithms behave similarly, or Quicksort seems to be very often the best algorithm. Quicksort is an in-place algorithm, and as result, larger amount of keys can be sorted without increasing the data miss ratio. However, its performance degrades as the number of keys to sort increases and overflows the data cache.

On the other hand, as the number of records to sort increases, the number of instructions executed increases. Also as the amount of data to sort increases, the data cache misses tend to increase.

Quicksort: Our Quicksort algorithm does not support tiling, thus as the number of keys to sort increases cache misses increase. However, since data is partitioned around the pivot, a good selection of the pivot increases the probability of an even distribution of the data into subsets, and would tend to reduce the number of passes required for all subsets to fit in the cache.

CC-radix: If the data bucket overflows the cache, CC-radix will partition the input data into sub-buckets. For each sub-bucket, the partition will continue until it fits into the cache. Executing more partitions implies executing more instructions. In addition, these partitions will cause many cache misses, since the data do not fit in the cache.

Multiway merge: As explained before the value of p can be chosen to minimize data cache misses, however, it will be chosen to maximize performance. Again, since each run is sorted using CC-radix or Quicksort, the number of records to sort in each run will also affect the performance of multiway merge.

Degree of sortedness

The standard deviation of input values distinguishes the performance of comparison-based sorting algorithms, such as, Quick Sort, and the performance of radix-based sorting algorithms, such as, CC-Radix. However, the performance of comparison-based sorting algorithms also depends on how well the original input array is sorted. We use Degree of "sortedness" to represented how far away the input array is from fully sorted, that is, how many more passes are required to sort the input array fully. In this section, we discuss how the performance of comparison-based sorting algorithms changes with the degree of sortedness, and if there is a feasible metric that can predict the relative performance of those algorithms and weather it is fast to calculate at runtime.

First let us look at what is the ideal metric for the performance of comparison-based sorting algorithms for a given input array. Here the "ideal metric" relax the requirement that we must calculate the metric before sorting the given input array. In other words, we assume we know both the original array and its sorted form in order to predict the performance of comparison-based sorting algorithms for that array.

The two basic operations in comparison-based sorting algorithms are comparison and swapping. Theoretical complexity usually measures the number of comparisons when applying a sorting algorithm on a specific kind of inputs. Comparisons are translated into corresponding branch instructions of the target processor. The higher the number of comparisons is, the larger the total delay of branch instructions, assuming that the branch prediction rate remains relative constant. On the other hand, the swapping operation affects the performance of comparison-based sorting algorithms in two ways. First, swapping accesses at least two memory units. Hence, more swapping operations means the algorithm has more memory read/write operations, which drags the sorting performance. The second way in which swapping operations affect the performance is related to the distance of the two memory units accessed. If the two memory locations are close, they are likely to be part of the same cache line or the same page. Accesses to them will have higher probability to be a hit in the cache or the TLB. However, if the two memory units accessed are far apart, the swapping operation will access more cache lines or memory pages, with the obvious effect of lower performance.

We want to use the degree of sortedness to predict performance of comparison-based sorting algorithms. From the above analysis of factors that affect the performance of such algorithms, it is clear that the measurements of the degree of sortedness must take into account both the number of operations and the locality of the operations. There are many definitions of degree of sortedness. However, some definitions have only theoretical value, because they can only be calculated after the original array is sorted and they can not be approximated with statistic methods. A metric that is widely used in the theoretical analysis of sorting algorithms, $k - sorted$ [14], is one of such definitions. $k - sorted$ is defined as in Definition 2.1. $k - sorted$ confines the number of comparison-and-swapping in the sorting and the locality of the operation. It is a good metric for the performance of comparison based sorting algorithms. However,

we can not calculate $k - sorted$ before sorting the array. Hence, we can not use $k - sorted$ as the metric to predict the relative performance of comparison based sorting algorithms.

Definition 2.1 *An array is said to be $k - sorted$ if no item is more than k positions from its position in the sorted array.*

The definition we use to measure the degree of sortedness is $gap - sorted$. The $gap - sorted$ is formally defined as in Definition 2.2. Intuitively, a $k - gap - sorted$ array consists of k sorted sub-sequences. A special case is that when $k = 1$, the whole array is sorted.

Definition 2.2 *An array $A[0 : n]$ is said to be $k-gap-sorted$ if the sequence $A[i : n : k]$ is sorted for $i \in [0 : k-1]$.*

We first look at in theory how the concept of $gap - sorted$ is related to the performance of comparison-based sorting algorithms. If an array is $k - gap - sorted$, the number of comparison required to fully sort the original array is limited. Theorem 2.3 provides the theoretical ground to calculate an upper-bound of the number of comparisons. Furthermore, when $h - sorting$ a $k - gap - sorted$ array, an element is compared with elements that are not farther than $h * k$ positions. This in effect caps the locality of swapping operation.

Theorem 2.3 *A g-gap-sorted sequence remains g-gap-sorted after sorting it into h-gap-sorted.*

Proof: see [35], p. 90. □

Next we look at how the performance of Quicksort is related to the $gap - sortedness$ in real world. We run experiments on Intel Xeon and Intel Itanium 2. The algorithm that we use is an optimized implementation of Quicksort. In all the experiments, the test input data is of same length(2^{22}) and same standard deviation. However, the $gap - sortedness$ of the test input data changes from 2 to 9. Figure 2.10 shows the performance of Quicksort versus the different $gap - sortedness$. The inputs with different gap-sortedness are randomly generated and then partially sorted to reach the desired gap-sortedness. Each point on the plot represents the average execution time of three runs of Quick sort on different inputs with same gap-sortedness. The plot proves our analysis that the performance of Quicksort decreases with bigger gap-sortedness. The conclusion is true on both Intel Xeon and Itanium 2.

However, this is just the first conclusive result we get about how the performance of comparison-based sorting algorithms changes with different input characteristics. We do not exclude other orthogonal input metrics that can also affect the performance, for example, the pivots selected. Also the performance of radix-based sorting algorithms is not affected by degree of sortedness. Moreover, though $gap - sortedness$ can be statistically approached, it is still expensive to calculate. In our experiment, the calculation of $gap - sortedness$ take about 20% of the sorting time.

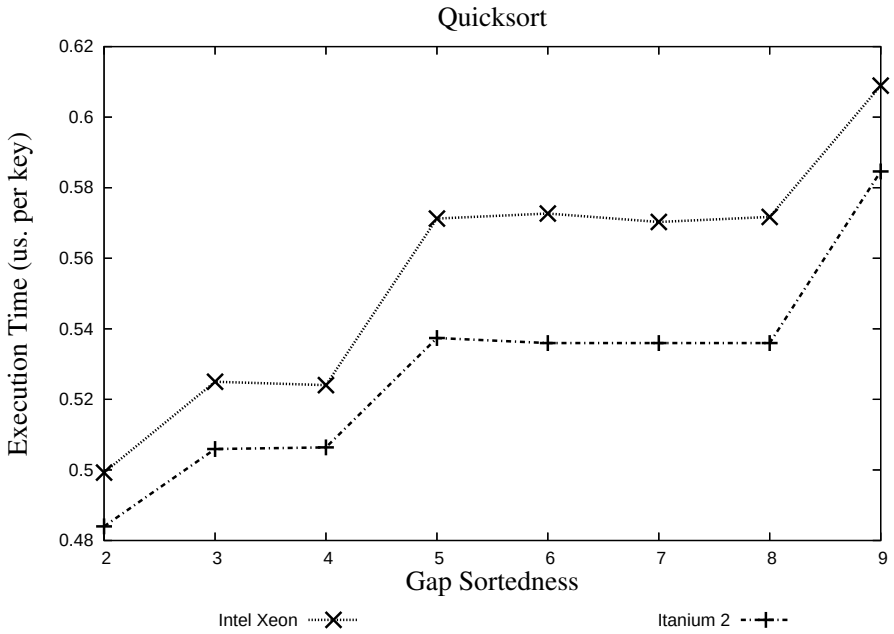


Figure 2.10. Performance of Quicksort vs. Gap-sortedness

Given this, and the fact that the empirical method to distinguish the performance of Quick sort and Merge sort is quite successful, we do not proceed to predict the performance of comparison-based sorting algorithms.

Finally, just notice that we did not cover all possible input characteristics that affect the performance of sorting algorithms. Also, some algorithms such as those in [19, 23] are very efficient when sorting inputs that are almost sorted. However, those algorithms are not covered in this thesis. About the problem of selecting input characteristics and their impact on performance, Leyton-Brown [42] proposes a mechanism to determine how to select the input characteristics to make a decision. In particular, Leyton-Brown proposes to sort input characteristics in asending order of their cost to calculate, so that the total cost of making a decision can be reduced. In our approach using genetic algorithms in the next chapter that is indirectly taken into account by the genetic algorithm, since the trade-off cost in computing versus performance will determine if an input characteristic is computed or not determine the algorithm to choose.

2.3 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.

	Algorithm parameter	Architectural feature
Quicksort	Threshold to use sorting network	Number of register
Multiway Merge Sort	Block size Fan-out	Lowest Cache size Lowest Cache line size
CC-radix	Radix of digit Block size	TLB size Lowest Cache size

Table 2.2. Algorithm parameters vs. Architectural features

Table 2.2 lists parameters of Quicksort, Multiway Merge Sort and CC-radix that depend on architectural features. To determine these algorithm parameters whose optimal values 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 [66] or SPIRAL [69]. 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* [47], 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 2.3.1, and then the implementation details, including the Winnow algorithm are presented in Section 2.3.2.

2.3.1 Entropy

As explained in Section 2.2.2, our experiments indicate that 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 algorithms 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 ("The Reverse Radix Sorting" in Figure 2.4) 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 calculate an entropy for each digit, so we obtain a vector of entropies. Each element of the vector represents the entropy of a digit position in the key. The length of the vector is equal to the number of digits.

2.3.2 Implementation Details

In this section we explain the implementation of our library. We explain the empirical search of parameters that depend on the target architecture in Section 2.3.2, the learning procedure used to compute the selection function in Section 2.3.2, and the runtime selection procedure in Section 2.3.2.

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, we have determined experimentally that the best version of the algorithm does not change significantly with the characteristics of the input data set and therefore only one version is used through out this work. In the case of multiway merge sort, the best version 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). For each size of the input data tested at installation time, these configurations are stored in a table indexed by number of keys and values of the 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).

1. *Quicksort*

   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 partitions smaller than the threshold are found. In ii), iii) and iv) we need to find a threshold. Thus, we run these options with thresholds that range from 8 to 32 in steps of 4. The experiment is executed three times with each randomly generated input set, 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 2.2.1). However, the scheduling in the C code will interact with the scheduling that the compiler does.

2. *CC-radix*

   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.

3. *Multiway Merge Sort*

   For Multiway Merge Sort, 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.

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 $\vec{E} =< E_0, E_1, ..., E_{d-1} >$.

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 obtained 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. The sizes of input sets change from 1M to 16M in step of 1M. It is very difficult to generate an input set with a given $entropy$ vector, so we use different values of the standard deviations 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 2.3.2, we keep this information in a table indexed by the amount of data to sort and the entropy vector.

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 2.11) 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 2.11-(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 2.11-(b)

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

Sample the input array and compute the entropy vector E_i
Compute $S = \sum_i W_i * E_i$
if $S \geq Threshold$
    choose CC-radix
else
    choose Algorithm using the size of Src

(a)

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

(b)

Figure 2.11. Runtime algorithms. (a)-Select Algorithm (b)- Select Multiway Merge Parameters.

2.4 Evaluation of Code Selection

In this section we present some measurements of the behavior of our sorting library generator. In Section 2.4.1, we describe the environmental setup that we used for the evaluation. Section 2.4.2 presents the performance results, and in Section 2.4.3 two sensitivity analysis experiments are discussed.

2.4.1 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 2.3 lists for each platform the architectural parameters, the version of the operating system, the compiler, and the compiler options used for the experiments.

	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
L2 Cache Line	64B	32B	32/64B	128B	128B	64B	128B
TLB	32	256	128	32	64	2*64	128
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 2.3. 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.

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 [48]. 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 5 512 to 8M in multiplicative steps of 8 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 [32]. Also, the values of r in our experiments was that assumed by that implementation.

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.

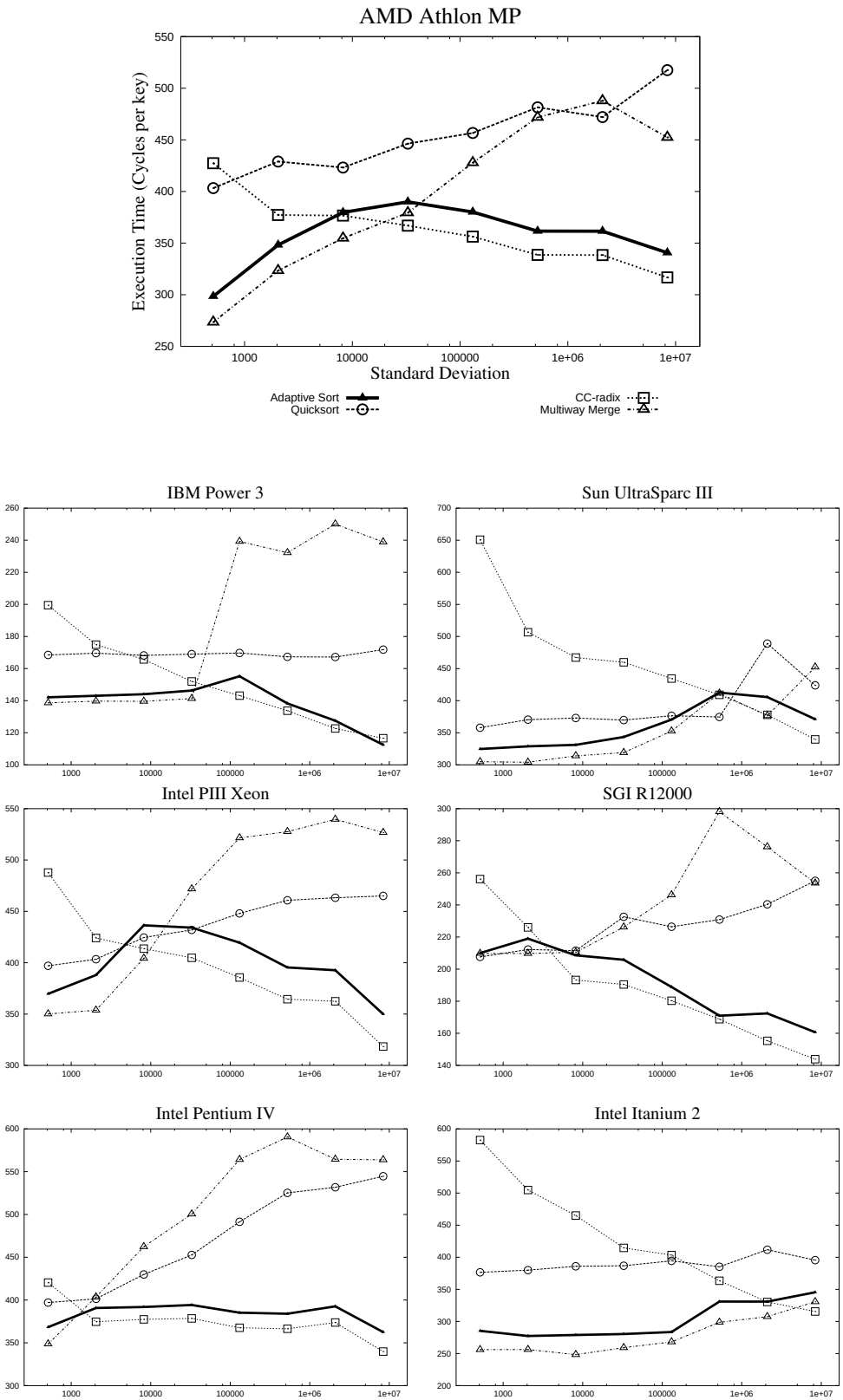


Figure 2.12. Execution time versus standard deviation for our library and the different sorting algorithms when sorting 12M records.

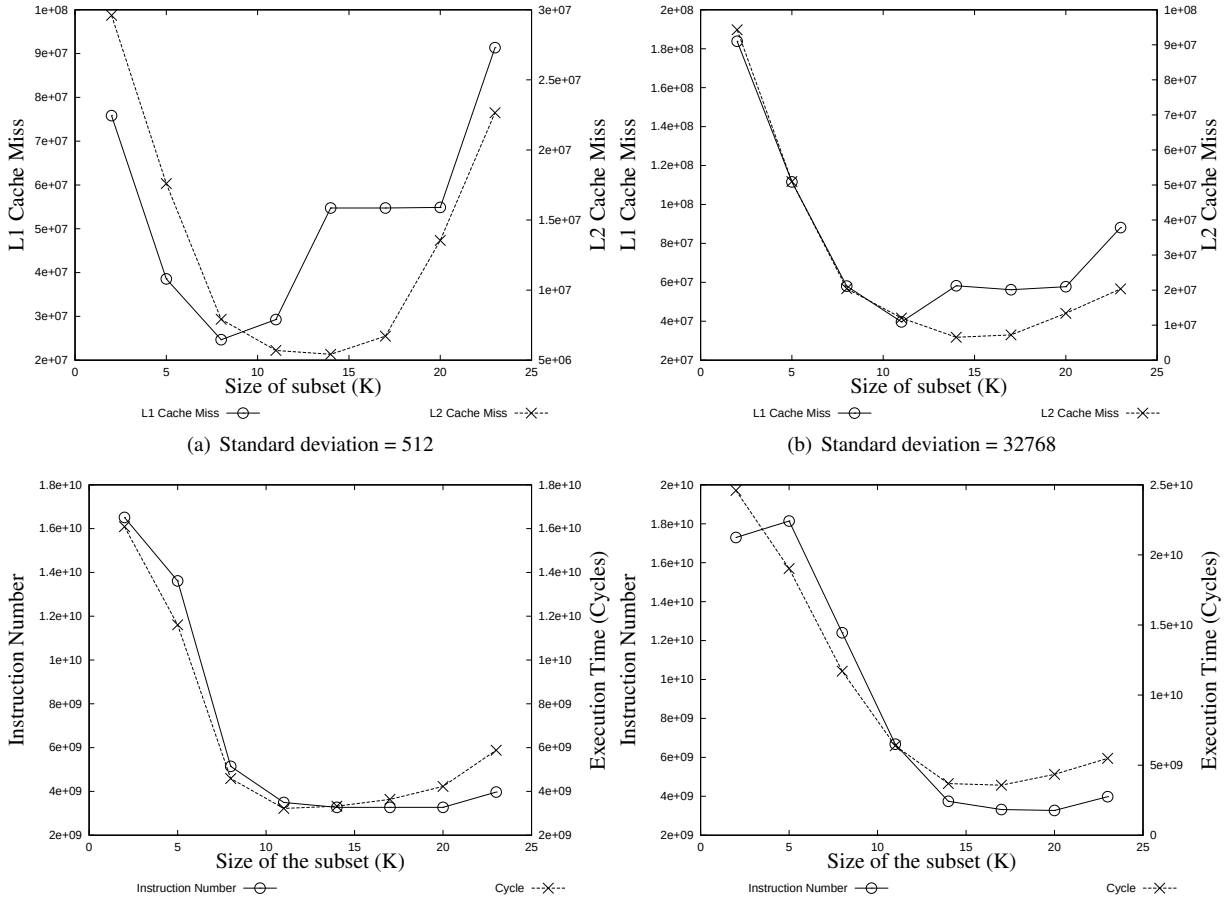
2.4.2 Performance Results

Figure 2.12 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 2.3. The input data sets used for the experiments in Figure 2.12 are different 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. from 512 to 8M in multiplicative steps of 4. 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).

Figure 2.12 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. For multiway merge, however, our library did not search for the best parameters after the cross-point between the two algorithms was reached. This was done to reduce installation time. Thus, in figure 2.12, the performance of multiway merge sort could be somewhat better in the interval where it is not the best algorithm.

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 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



(a) Standard deviation = 512

(b) Standard deviation = 32768

(c) Instruction number and Cycle vs. run size. standard deviation = 512

(d) Instruction number and Cycle vs. run size. standard deviation = 32768

Figure 2.13. Effect of changing the run size when sorting 12M tuples on SGI R12000.

our experiments we used the CC-radix sort provided by Jimenez et al. [32] and we did not modify their code.

If we compare Quicksort, CC-radix, and multiway merge sort on our inputs, 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, which means the values of the input array spread more densely around the mean value of the array, 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 2.12 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 that we have presented in this chapter is well suited to the problem for which it was used.

2.4.3 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 used for small partitions by Quicksort and for the heap size in the implementation of multiway merge sort.

Table 2.4 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 (which determine for what size partition Quicksort switches) that delivered the best result. Notice that the performance difference between Quicksort and the rest of the optimizations will depend on the input sets used to collect the results, since different values can result in a higher or a smaller number of small partitions. Thus, to collect the results in table 2.4 we ran three input sets, and we show the average performance. Since the performance of Quicksort is not significantly affected by the different values of standard deviation, we think this is the best way to show the sensitivity results. The results show that Quicksort plus sorting networks is the optimization that delivers the best performance. The only exception is 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.

Table 2.1 shows how the register blocking size affects the sorting performance of integers. For two of the three platforms, SGI R12K and Intel PIII Xeon, different rb's don't make much difference. We can see any value from 8 to 24 performs similarly well. For Sun UltraSparcIII, $rb = 8$ is the prominent best choice. This may related to the fact

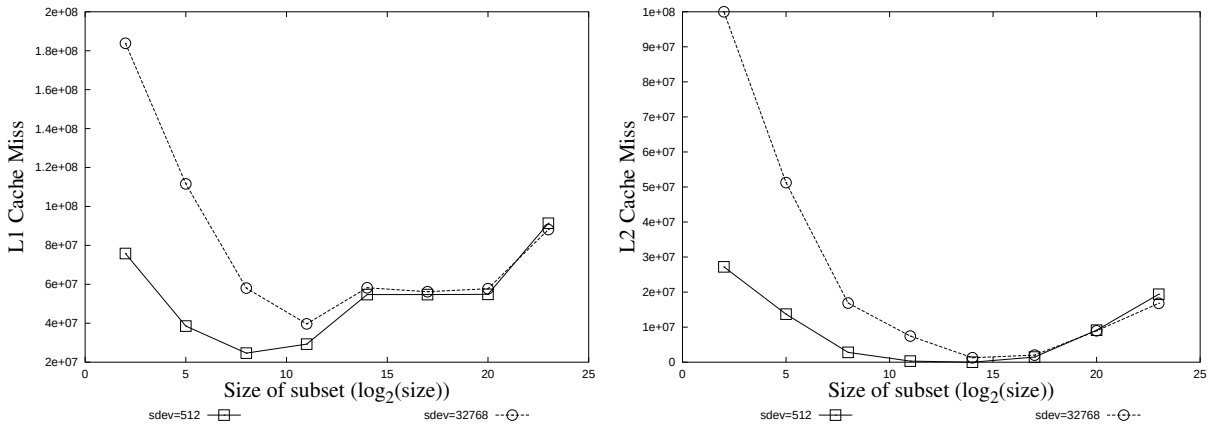
	SGI R12000		Sun UltraSparcIII		Intel PIII-Xeon	
	4M	16M	4M	16M	4M	16M
Quicksort	2.960	13.915	1.579	6.498	2.429	10.886
Insert Sort at the end	2.676	11.576	1.406	6.098	2.174	9.759
Insert Sort at each partition	2.859	11.976	1.475	6.415	2.321	10.486
Sorting Networks	2.275	10.994	1.372	6.160	2.081	9.207

Table 2.4. Execution time in seconds for Quicksort and Quicksort plus some optimizations using insert sort or sorting networks.

that Sun Compiler can only use 8 registers. If more registers are requested, unnecessary register spills will happen.

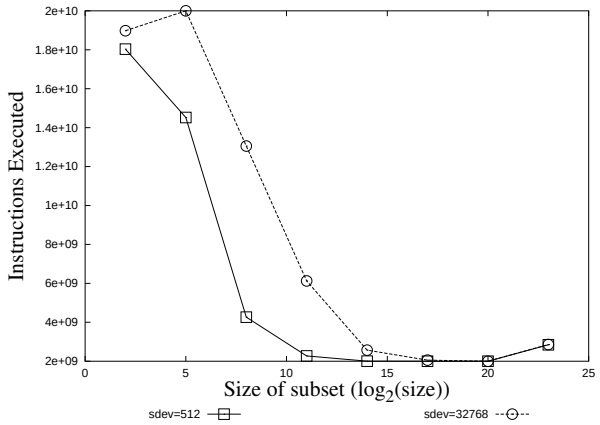
We now discuss how different sizes of the heap affect the performance of multiway merge sort. Notice that for a fixed number of keys to sort, the heap size determines the size of each sorted subset and vice versa. To analyze whether the best size of the subsets depends only on the cache size or on both 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 with different sizes. Figure 2.15 shows the results. The plots on the left corresponds to data sets with a standard deviation of 512, while for the ones of the right the standard deviation is 32768. 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 X axis is using a logarithmic 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. The four events were measured at the same time. To collect the data for each value of standard deviation, we repeat three times the same experiment with different input data sets that had the same standard deviation but different values. The plots use error bars to show the average, the lowest, and the highest value measured for each event. Because the variation of experiments using the same input configuration is small, below 3% in most cases, the corresponding error bars are short or appear almost as a dot in the plots.

Comparing the results in Figure 2.14 for the two values of the standard deviation we can observe that the plots of L1 cache miss, L2 cache miss, and execution time behave differently. The figure shows that although the amount of data to sort and size of the subset are the same, different standard deviations result in different data misses and number of executed instructions. Notice that in this situation, where data cache misses and number of executed instructions change depending on the standard deviation of the input data, the best performance (shown in the number of cycles plot) is obtained when both data misses (L2 misses for the R12000) and the instructions executed are minimized. As a result, we can conclude that the point where the best performance is obtained depends not only on the cache size, but also on the standard deviation. On the other side, to verify that the differences in cache misses, instructions and cycles

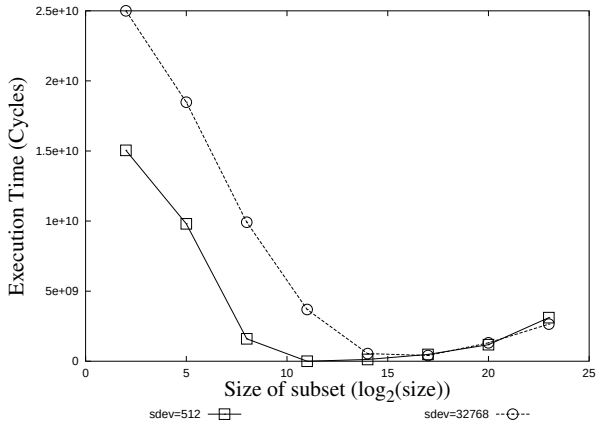


(a) L1 cache misses versus size of the subset.

(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 2.14. 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.

executed depend on the standard deviation, and not on the values to be sorted, we ran three 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 event were always less than 5%.

The foregoing discussion shows that as expected sorting behaves differently than dense numerical linear algebra. For example, given a tiled implementation of matrix multiplication, the size of the matrix is the only factor affecting the number of cache misses if the size of the tile is kept constant. As the tile size changes the number of cache misses will change, but the number of executed instructions remains approximately constant. Thus, in this case the problem of searching the optimal tile size can be simplified to that of finding the tile size that minimizes the data cache misses. Since the the optimal tile size only depends on the cache size, a simple expression of the cache size can be used to compute the optimal value of the tile size [71]. In sorting, the problem is more complex and a more complex approach involving runtime decisions such as the one discussed above seems to be necessary. In the example shown in Figure 2.13, our runtime algorithms used the size of the subset obtained with the 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.

For standard deviation = 512, the run time choose "run size" = 16k. The difference in performance is less than 5%. When standard deviation = 32768, the run time choose "run size" = 32k. It is almost the best value in the "cycle vs. run size" plot. In the example shown in Figure 2.14, our runtime algorithm will use the heap size obtained using a training set containing 16M records. When the standard deviation is 2^16 this size is 128K, while for the standard deviation 2^20 the size is 256K. These values of heap size are within the 7% of the best value shown in Figure 2.14.

2.5 Conclusion

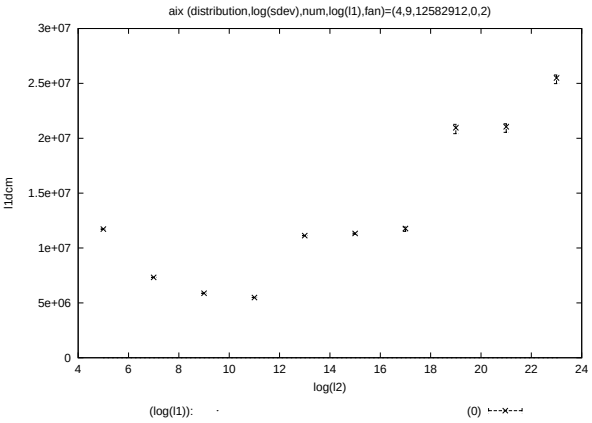
In this chapter, we study how to tune sorting for a specific platform through compile time optimization and runtime adaptation.

At compile time, we apply compiler optimizing technologies, including tiling, register blocking and cache line aligning to a memory-hierarchy conscious sorting algorithm. We use a heuristic-directed search to obtain the best parameters for the algorithm.

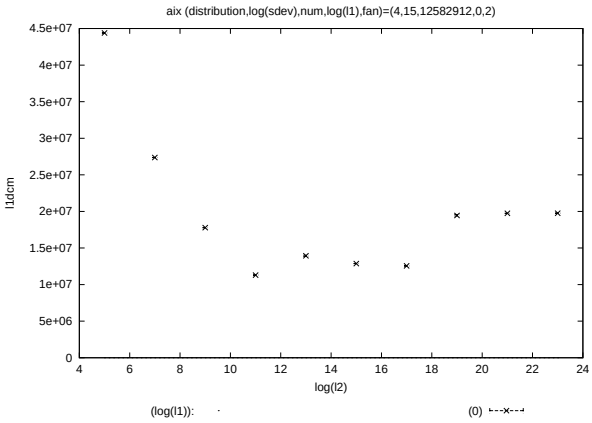
Runtime optimization of sorting algorithms has not been studied in great detail in previous research. We propose the idea to build a composite a sorting algorithm from comparison-based sorting algorithm and a radix-based sorting algorithm. We take advantage of the best part of the two algorithms at runtime. We show that the entropy of the input data is a good criteria for runtime algorithm selection. During the installation of the library, a machine learning technology, the linear separator algorithm, is used to train the runtime adaptation for the platform. After that, the runtime adaptation code is generated and compiled into the sorting library. The result of the composite sorting algorithm with runtime adaptation shows the optimized algorithm get the best parts of two algorithm candidates.

On the other hand, our approach, being an empirical study, has its own limitations. First, in order to generate training data with different entropies, we use inputs that we generate using a random generator and a Normal distribution. However, Normal distribution in effect skews the probabilities of possible combinations of entropies, that is, different combinations may have different probabilities to be generated. Our training hence suffers from the biased training inputs and the result code selection function may overfit to the training sets that we use. Our code selection mechanism can be further improved if we complement the training phase with inputs from unconstrained Uniform distribution, which will enable every combinations of entropies to have equal probability of being generated. Second, our approach does not take into account other input features that may affect the relative performance of sorting algorithms, for example, the number of distinct key values. In addition, since the keys are generated using a random

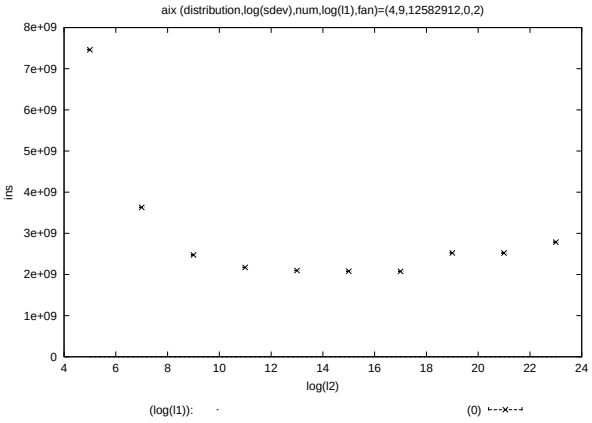
number generator, the value of each key is independent of the value of another key. Consequently, our input data does not reflect the situation where keys are correlated. Including those input characteristics will require much more complex learning techniques to select an optimal sorting algorithm, because the learning technique should be able to handle a much larger input space and because the selection may have to consider input characteristics that may not be orthogonal.



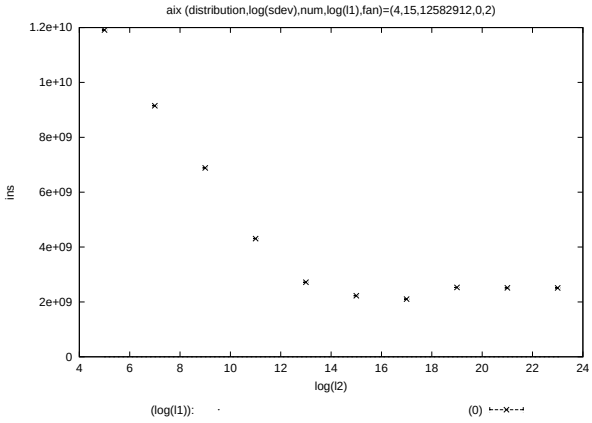
(a) L1 cache miss vs. NB



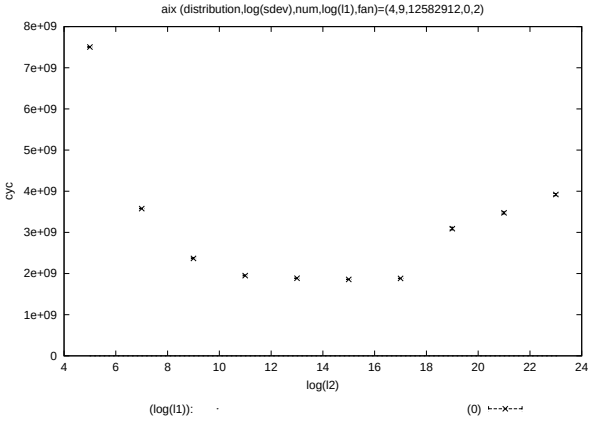
(b) L1 cache miss vs. NB



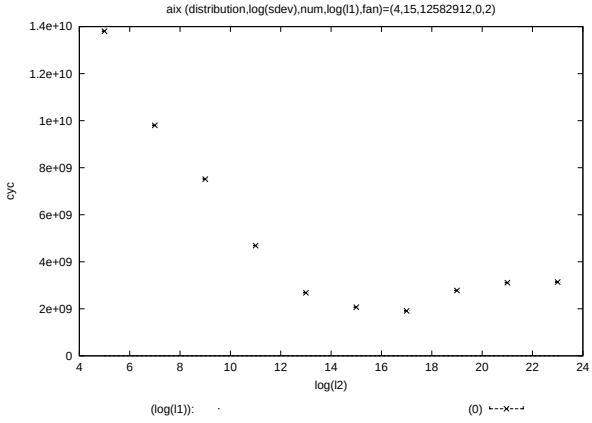
(c) Instruction number vs. NB



(d) Instruction number vs. NB



(e) Performance (cycle) vs. NB



(f) Performance (cycle) vs. NB

Figure 2.15. The effect of block size sorting 16M tuples on IBM Power3

Chapter 3

Code Synthesis

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 [66], PHiPAC [15], FFTW [26] and SPIRAL [69] 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. A signal processing transform can be represented by many different, but mathematically equivalent, formulas. The programs implementing these transforms have different running times. Since the number of formulas can be quite large, exhaustive search is usually not possible. 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 [26, 33], or genetic algorithms [62].

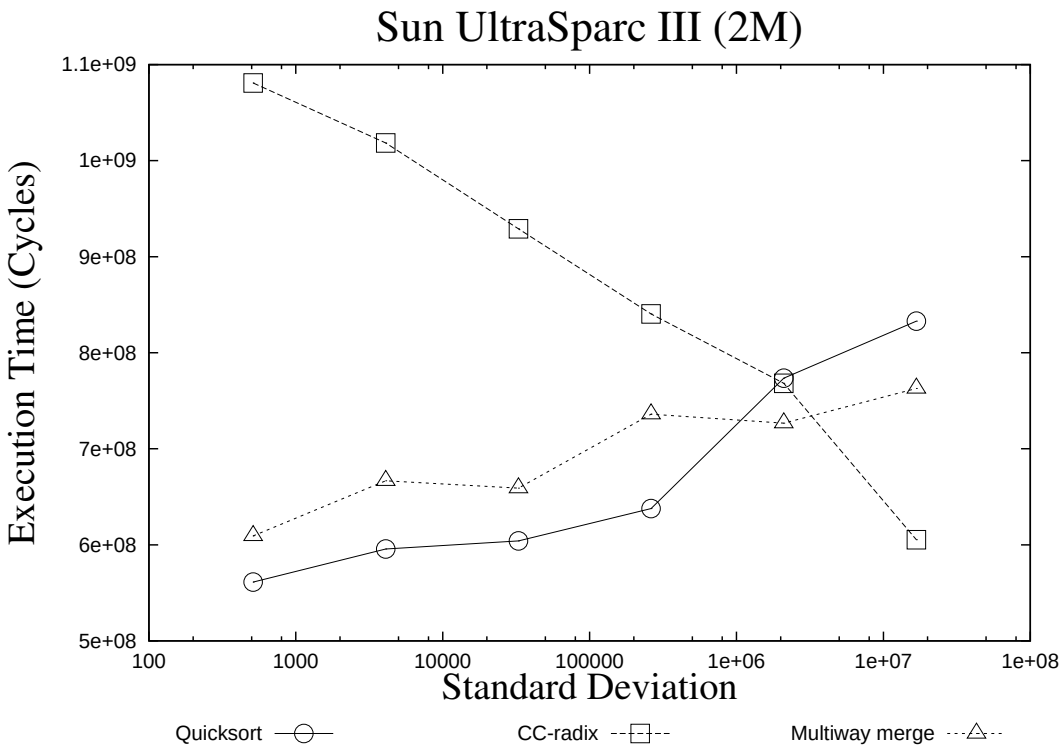
In this chapter, we explore the problem of generating high-quality sorting algorithms. A difference between sorting algorithm and the algorithms implemented by the most popular library generators such as ATLAS is that the performance of the algorithms these generators implement implement is completely 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 a radix

sort performs 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 Chapter 2 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 Chapter 2 - that can be selected at run time.

In this chapter, we extend and generalize our earlier approach discussed in the previous chapter. Our goal here is to produce 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 as shown in the previous chapter, 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. In the previous chapter, composition was achieved by selecting a single "pure" sorting method to handle the whole data set. In this chapter, we generalize that approach so that it becomes possible for different sections of the data to be processed by different algorithms. Besides the sorting primitives, which are the kernels of "pure" sorting algorithms, code generated by the methods of this chapter 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 [16, 28, 52, 67]. Genetic algorithms have also been used to search for the appropriate formula in SPIRAL [62] and for traditional compiler optimizations [20, 24, 63]. The idea of building a composite algorithm by dynamically selecting the algorithm to use has also been used by Lagoudakis [36, 37]. In this work, the selection is done based on a Markov decision process (MDP). The work shows a small example where the learning mechanism decides the best size of the input to switch to insertion sort when sorting using quicksort.

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 chapter is organized as follows. Section 3.1 discusses the primitives that we use to build sorting algorithms. Section 3.2 explains why we chose genetic algorithms for the search and explains some details of the algorithm that we implemented. Section 3.3 shows performance results. Section 3.4 outlines how to use genetic algorithms to generate a classifier system for sorting routines, and finally Section 3.5 presents out conclusion.



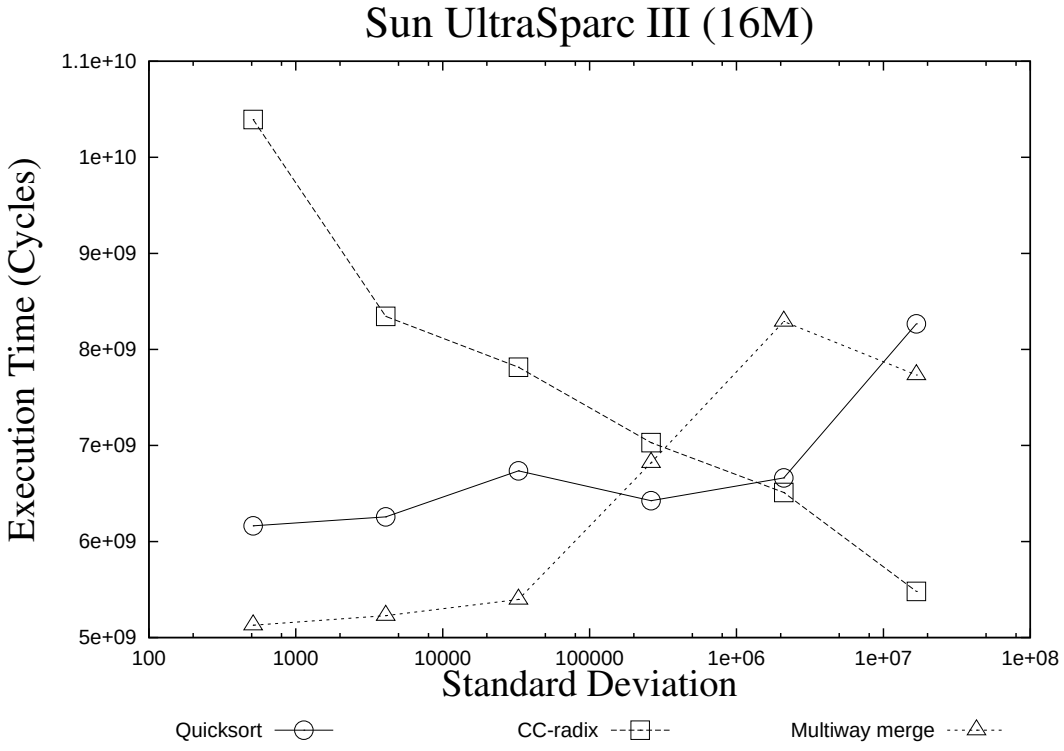


Figure 3.1. Performance impact of the standard deviation when sorting 2M and 16M keys.

3.1 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 3.1, which plots the execution time of three sorting algorithms against the standard deviation of the keys to be sorted. Results are shown for Sun Ultra-Sparc III, and for two data sets sizes, 2 million(M) and 16 million(M). The three algorithms are: Quicksort [31, 59], a cache-conscious radix sort (CC-radix) [32], and multiway merge sort [35]. Figure 3.1 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 entropy vector of the records to be sorted. CC-radix is better when the entropy of the records is high, especially the entropy of the most significant digits, 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. On the other hand, if we compare the performance on the two platforms we can see that, although the general trend of the algorithms is always the same, the performance crossover point occurs at different points.

It has been known for many years that the performance of Quicksort can be improved when combined with other algorithms [59]. 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, 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 [35]. The idea of using insertion sort to sort the small partitions that appear after the recursive calls of quicksort was also used in [59] to improve the performance of quicksort.

Darlington [21] introduced the idea of sorting primitives and identify merge sort and quicksort as two sort primitives. In this chapter, 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 algorithms we consider 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 its original form, 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 so that each key in partition i is smaller than or equal to any key in partition i+1, for i=1,...,np.

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 [35]. 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 3.2 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. As discussed in Chapter 2, 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. 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.

   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 reverse radix sort algorithm. The DR primitive distributes the records to be sorted into buckets depending on the value of one of the digits in the record. Thus, if we use a

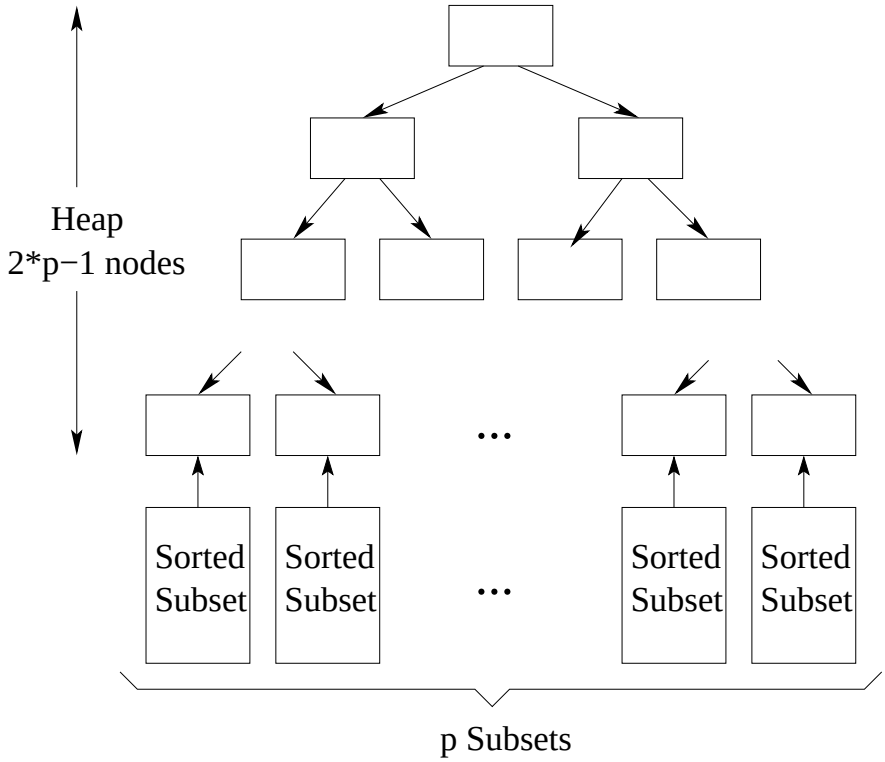


Figure 3.2. Multiway Merge.

radix of r bits, the records will be distributed into 2^r sub-buckets based on the value of the k*th* digit of r bits for some value k. Our implementation relies on the counting algorithm [35] 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. Figure 3.3 presents an example where the DR primitive is applied to the second most significant digit. The example assumes base 8 numbers, and the radix size is therefore 3 bits. An important feature of this primitive is that it is a non-comparison based algorithm.

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, which represents the most significant digit, 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 above are used to determine the number

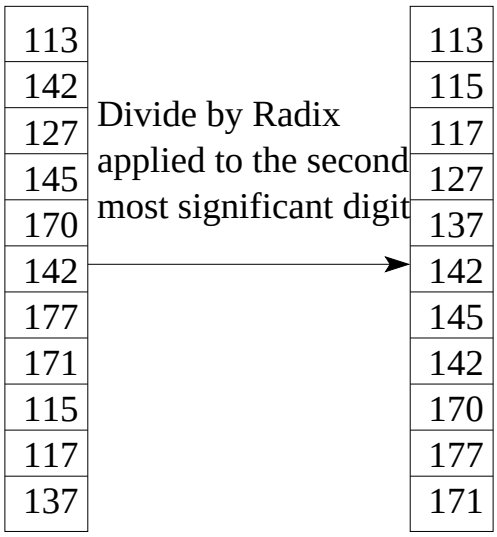


Figure 3.3. Divide by radix primitive applied to the second most significant digit.

of keys of each possible value and reserve the corresponding space in the output vector. However, these steps (in particular computing the histogram) are costly. If it can be assumed that a digit is uniformly distributed and that the number of keys for each possible value is the same, thus this overhead can be avoided. 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 may 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. Moreover, to reduce the probability of bucket overflow, we allocate buckets that are slightly larger than its expected size. As in DR, the DU primitive has a $radix$ parameter.

The DV, DP and DR primitives can be combined and applied recursively. Therefore, we need an exit primitive to stop the recursive application of the above primitives. Since quicksort and radix perform well when sorting small amount of data, we have chosen the corresponding DV and DR primitives to stop the recursion and sort the remaining elements. These primitives we call leaf primitives.

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

5. $Leaf - Quicksort$ (LDV)

$Leaf - Quicksort$ can be thought as a special case of DV, so it is named as LDV. This primitive specifies that Quicksort 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. $Leaf - Radix - Sort$ (LDR)

   $Leaf - Radix - Sort$ is a special case of DR, just like LDV as to DV. This primitive specifies that Radix Sort 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 a competitive performance. Merge sort is not considered as an alternative to sort leaves because we have found it to be slower than quicksort and radix sort for small partitions.

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 3.1, 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 3.1, 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 Chapter 2, the notion of entropy from information theory.

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 Chapter 2.

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-thresholds for the $n+1$ branches and the weight vector.

Table 3.1. Summary of primitives and their parameters.

The primitives and their parameters are listed in Table 3.1. We will use the eight primitives presented here (six sorting primitives and two selection primitives) to build sorting algorithms. Figure 3.4 shows an example where different sorting algorithms are encoded as a tree of primitives. Figure 3.4-(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 3.4-(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 3.4-(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 3.2, with a $fanout$ of 4, which is the parameter value of the DP primitive.

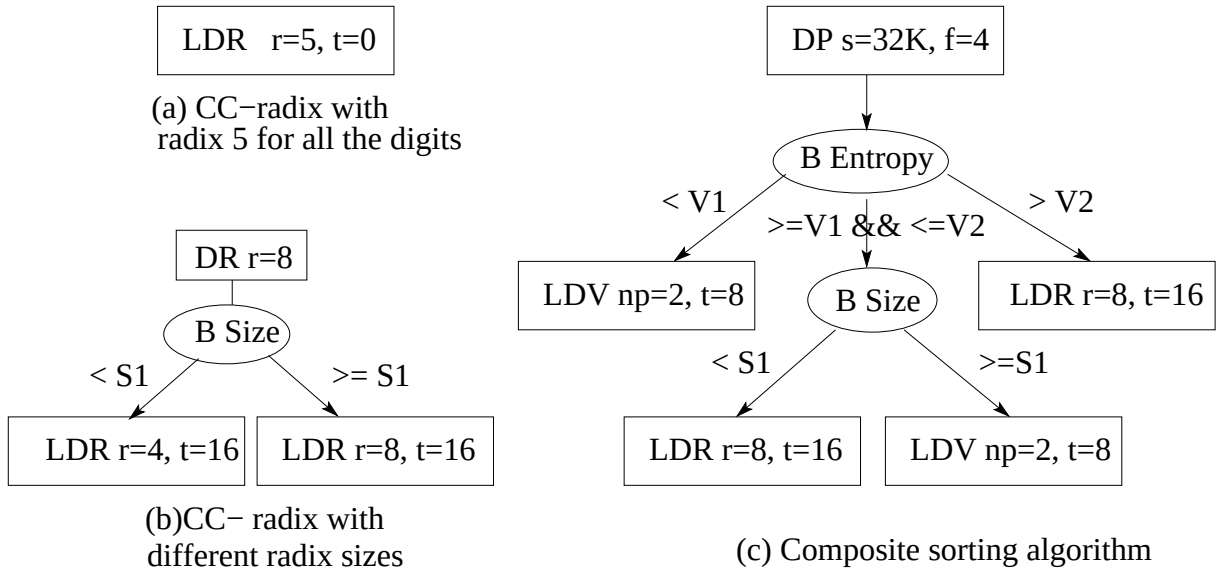


Figure 3.4. Tree based encoding of different sorting algorithms.

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.

Figure 2.12 is the experiment result of the runtime adaption mechanism as in Chapter 2. The runtime adaption mechanism succeeds in selecting the best part of two sorting algorithms respectively. However, for each part, the chosen algorithm can be further improved. We need a more fundamental mechanism to tuning the algorithmic structure and parameters of sorting algorithms.

Some sorting algorithms, e.g., Quick Sort, Merge Sort, can be synthesized from several primitives. John Darlington [22] introduced a series of primitives that can systematically construct Quicks Sort, Selection Sort, Merge Sort and Insertion Sort. For example, Quick Sort and Selection Sort are the same algorithm with different parameters. Quick

Sort partitions the input array based on the value of a randomly selected pivot, while Selection Sort also partitions the input array based on value but it makes sure the partition with smaller values has only one element (the smallest element).

The idea to build sorting algorithms from primitives is a good starting point because if we find a way to methodically synthesize sorting algorithms with primitives, we can explore a much larger sorting solution space than traditional sorting algorithms.

DV, DP, DR, LQ, LR, BN and BE are the seven primitives we used to synthesize sorting algorithms. They can not compose all the possible sorting algorithms. But by combining them in different structures and with different parameters, they can build a much larger space of sorting algorithms than just several traditional sorting algorithms, like Quick Sort or Radix Sort. Also the algorithm in this space could have the built-in capabilities to adapt to the input data at run time, which is not a feature for traditional algorithms.

The next section describes the methodology that we use to search better sorting algorithms on a given platform.

3.2 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.

3.2.1 Why Use Genetic Algorithms?

Traditionally, the complexity of sorting algorithms has been studied in terms of the number of comparisons executed assuming a specific distribution of the input, such as the uniform distribution [35]. 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. There are analytical models of the performance of sorting algorithms in terms of architectural features of the machine. However, the only way to identify the best algorithm is by applying empirical search, because the models do not take into account properties of the inputs and because they do not reflect accurately real machines. Furthermore, we will need to find the parameter values of a sorting algorithm that deliver the best performance on a platform, given its cache size, number of registers, and so on. Recent experimental results with ATLAS have proven that doing an empirical search of parameter values such as tile size and degree of loop unrolling can improve performance significantly relative to using the values selected by compilers [71]. For applications such as sorting, the difference should be even greater since current analysis techniques are not powerful enough to support automatic restructuring of sorting algorithms. For example, it is not generally feasible, and as far as we know, no compiler today attempts , to automatically determine that it is possible to change the partition size for multiway merge

sort.

Our approach is to use genetic algorithms to search for an improved sorting algorithm. The search space is defined by composition of the *sorting and selection primitives* described in Section 4.3 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.

The reason to use the sorting and selection primitives to build an effective sorting algorithm is that a hybrid algorithm often perform better than pure algorithms. A good sorting algorithm is more likely a composite algorithm consisting of a hierarchy of sorting algorithms whose particular shape depends on the characteristics of the input to sort and the architecture of the target machine. The intuition behind this is that different partitions will benefit more from different sorting algorithms, and as a result a better sorting algorithm can be obtained from the composition of these different sorting algorithms.

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

- Using the primitives in Section 3.1, the sorting algorithms can be encoded in the form of trees such as those in Figure 3.4. Genetic algorithms can be easily used to search in this space for the most appropriate tree shape and parameter values. Other loop-based search mechanisms, such as hill climbing, can search for the appropriate parameter values, but they are not adequate for searching for the optimal tree.

- The search space of sorting algorithms that can be derived using the eight primitives in Section 3.1 is too large for exhaustive search. For example suppose that each primitive has only one parameter, and each parameter can only take two different values. Assuming a tree with 6 nodes, and ignoring the structural differences, the number of possible trees is $(8 * 2)^6 = 16777216$.

- 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. Suppose that a sub-tree is optimal when sorting small number of records. It can then be used as a building block for a sorting algorithm for larger number of records.

  This is the idea behind dynamic programming, where the solution to a large problem is composed of the best solutions found for smaller problems. However, Common practices of dynamic programming employ fixed strategy of how to construct solutions for larger problems from sub-solutions. Genetic algorithms are more flexible. Genetic algorithms can, like dynamic programming, obtain an good sorting algorithm for 16M records which uses the best sorting algorithm for 4M records. However, genetic algorithms are not restricted to using the solution found for a smaller problem as the building blocks to solve a larger problem. They may also find that the best solution for sorting 16M records has nothing to do with the one found for 4M records.

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.

3.2.2 Optimization of Sorting with Genetic Algorithms

Encoding of Sorting Genomes

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

Operators of Genetic Algorithm

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. Here we use single-point crossover and we choose the crossover point randomly. Figure 3.5 shows an example of single-point crossover.

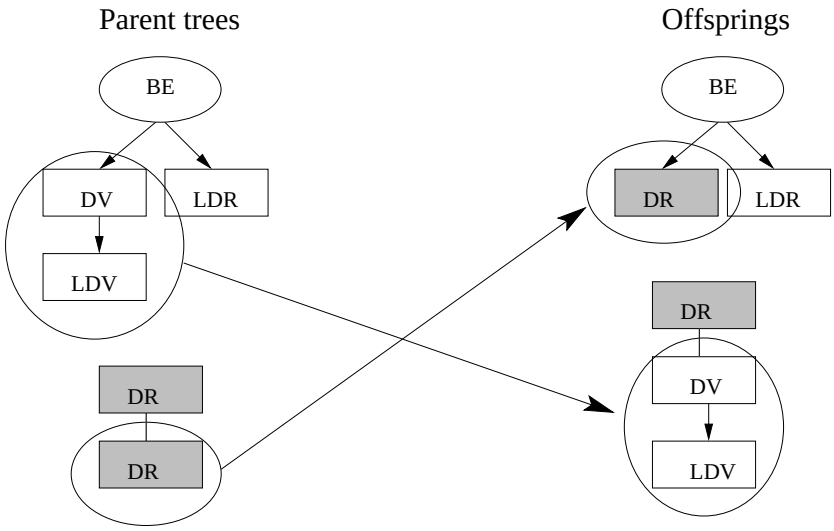


Figure 3.5. Crossover of sorting trees.

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 [13]. 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. The distance from the new value to the old value is randomly selected from a Normal distribution with mean equal to zero.

2. Exchange two subtrees. This type of mutation can help in cases like the one shown in Figure 3.6-(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 3.6-(b) shows an example of this mutation.

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

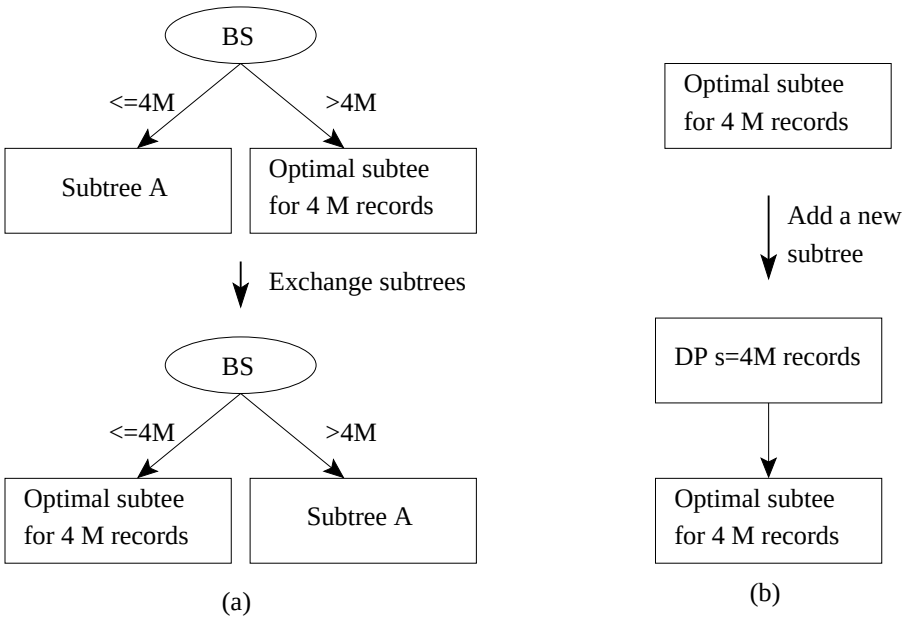


Figure 3.6. Mutation operator. (a)-Exchange subtrees. (b)-Add a new subtree.

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 sorting all training inputs in a generation 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 training inputs.

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

Evolution Algorithm

An important decision is to choose the appropriate evolution algorithm. The evolution algorithm determines how many offsprings will be generated, how many individuals of the current generation will be replaced and so on. The search for the optimal sorting algorithm is an incremental learning problem and, as such, what has already been learned should be remembered and used later.

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 3.7 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. In this way, the number of individuals remains constant across generations.

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

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 = Generate input sets randomly with different sizes and different standard deviations
- Use each genome of P to sort each element of S and store fitnesses with corresponding genomes
- Apply fitness function to remove the k least fit individuals from P.

}
}

Figure 3.7. Genetic Algorithm

and initial population that we use will be discussed in the next section.

3.3 Evaluation of Gene Sort

In this section we evaluate our approach of using genetic algorithms to optimize sorting algorithms. In Section 3.3.1 we discuss the environmental setup. Section 3.3.2 presents performance results, and Section 3.3.3 presents the sorting trees produced for each target platform and analyzes their characteristics.

3.3.1 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 3.3 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 3.7. Table 3.2 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 show that 30 generations suffice in most cases to obtain a stable solution.

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).

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

Table 3.2. Parameters for one generation of the Genetic Algorithm.

For the initial population we have chosen trees representing the pure sorting algorithms of CC-radix [32], quicksort [59], multiway merge [35], the adaptive algorithm that we presented in the previous chapter, and random variations of these trees. 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.

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

3.3.2 Experimental Results

In this Section we present the performance of sorting routines generated using genetic algorithms. We first compare with other sorting routines with commercial libraries such as INTEL MKL, C++ STL, and IBM ESSL. At last we evaluate the robustness of our approach.

Performance Results

We use genetic algorithms to search for either a specialized or a general sorting algorithm. A specialized sorting algorithm is an algorithm that has been optimized to sort data with some fixed characteristics like number of records and/or standard deviation. A general sorting algorithm is one that has been optimized to perform well on the average

for a wide range of input data values. We used the genetic algorithm to generate a general 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}. We also use the genetic algorithm to search for a specialized sorting algorithm. This specialized algorithm also sorts 32 bit integers, and has been tuned using data sets sized from 8M to 16M, and standard deviation fixed at 2^{19}.

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 [48, 60, 54]. We assume that this array has been created before our library routines are called.

Figure 3.8 shows the performance of five different sorting algorithms: quicksort, CC-radix, multiway merge, the adaptive sort algorithm that we presented in Chapter 2, and the sorting algorithm generated using the genetic algorithm (*Gene Sort*). For CC-radix sort we use the implementation provided by the authors of [32]. For quicksort, multiway merge sort, and adaptive sort we use the implementations that we presented in the previous chapter. 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 before. *Gene Sort* is the algorithm generated using the approach presented in this chapter.

Figure 3.8 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 3.8 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 in all cases smaller than 3%. The test inputs have a normal distribution, which was also the distribution of the training inputs. However, we have shown that show that the sorting routines in Figure 3.8 obtain similar performance when sorting inputs with uniform or exponential distribution. This agrees with the results that we reported in the previous chapter.

Figure 3.8 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 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 the next section, in the

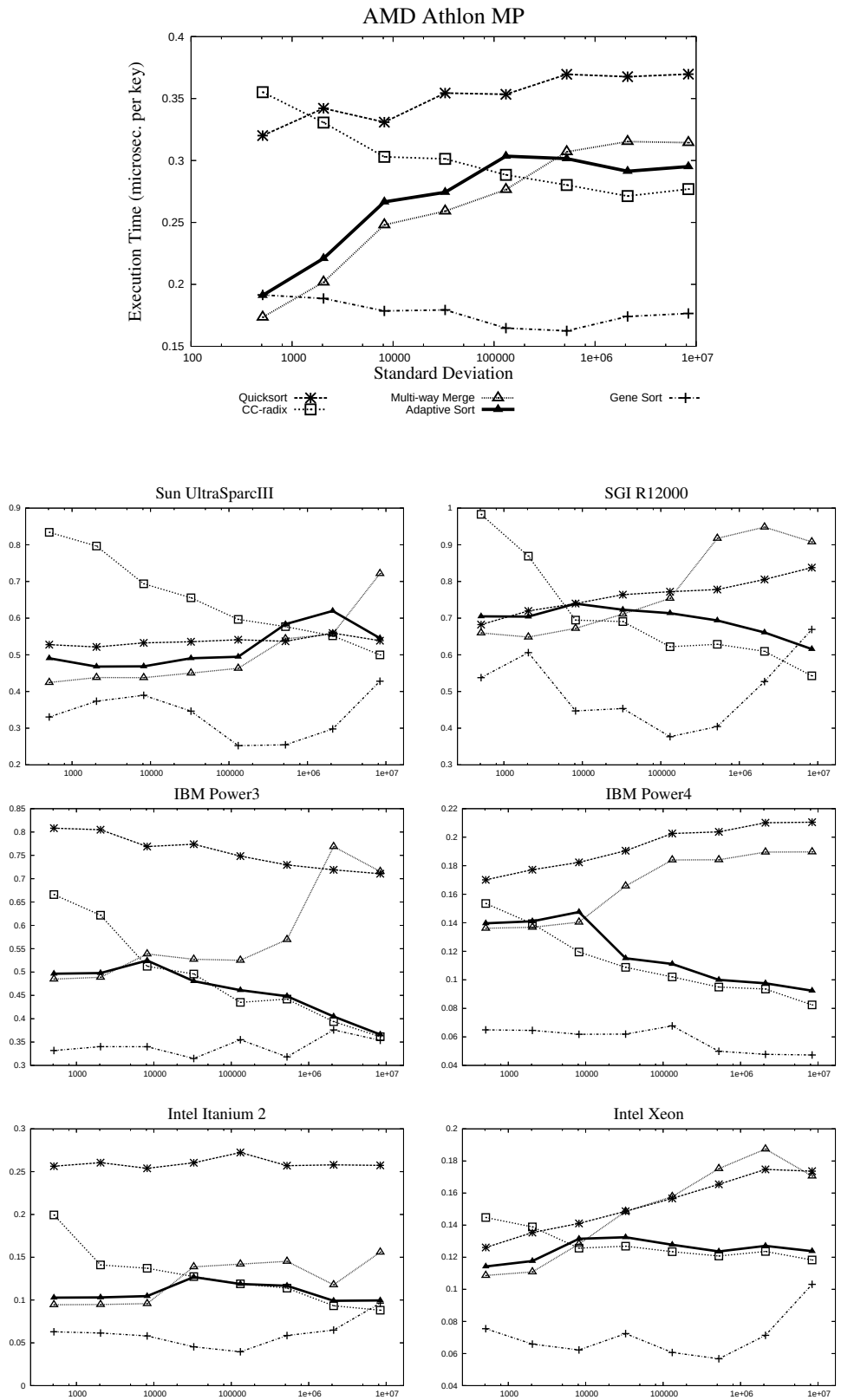


Figure 3.8. Performance of sorting algorithms as the standard deviation changes

end none of the algorithms found by our genetic algorithm used this primitive. The result is that for each particular platform the genetic algorithm has been able to find a sorting algorithm that performs better than any of the base pure sorting algorithms.

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 3.2.2). 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% for the inputs tested.

We have also investigated the additional speedups that can be obtained when generating a specialized algorithm. Our specialized algorithm *Special Gene sort* has been been tuned for inputs with fixed standard deviation 2^{19}, which is shown in the plots with an arrow. For this value of standard deviation, the specialized algorithm performs better than the general algorithm in all the platforms, with the only exception of Intel Xeon. Notice that this specialized algorithm may run slow when sorting inputs with standard deviation very different from the ones used in the training process, specially for low values of the standard deviation. The specialized algorithm achieves an average 5% performance improvement over the general one for the specific standard deviation for which it was trained.

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 [29]. 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 3.9 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 3.8, 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 4.4 we present a slightly different approach that generates the Xsort routine. As can be seen in Figure 3.9 Xsort is faster than any of the commercial libraries that we considered (including the IBM ESSL library in the Power 3). On the average of the performance on all test inputs, 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.

3.3.3 Analyzing The Best Sorting Genomes

Table 3.4 presents the best sorting genome found in the experiments of Section 3.3.2 for the Gene Sort routines in Figure 3.8. The string representation of the genome is of the form *(primitive parameters (child 1) (child 2)...)*, where parameters are those shown in Table 3.1. 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 4.3).

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 3.4. Gene Sort algorithm for each platform.

We are unable to verify that the sorting genomes in Table 3.4 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 3.4 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.

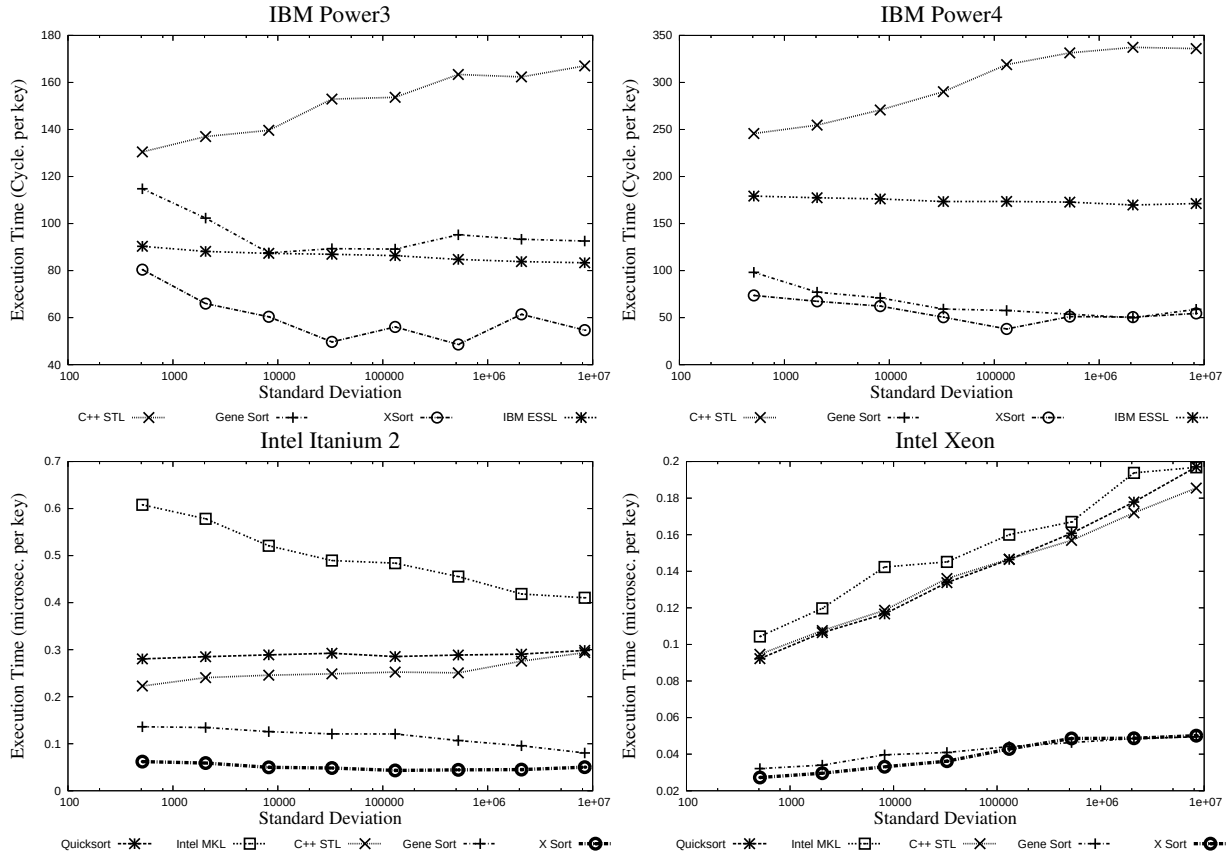


Figure 3.9. Performance comparison with commercial libraries.

On the other hand, the observation that selection primitives are rare in the sorting algorithms, leads us to think that it might be because either a) a single algorithm is the best for all standard deviation and sizes or b) the approach we use is not effective at deriving a composite algorithm where different algorithms are selected based on input size

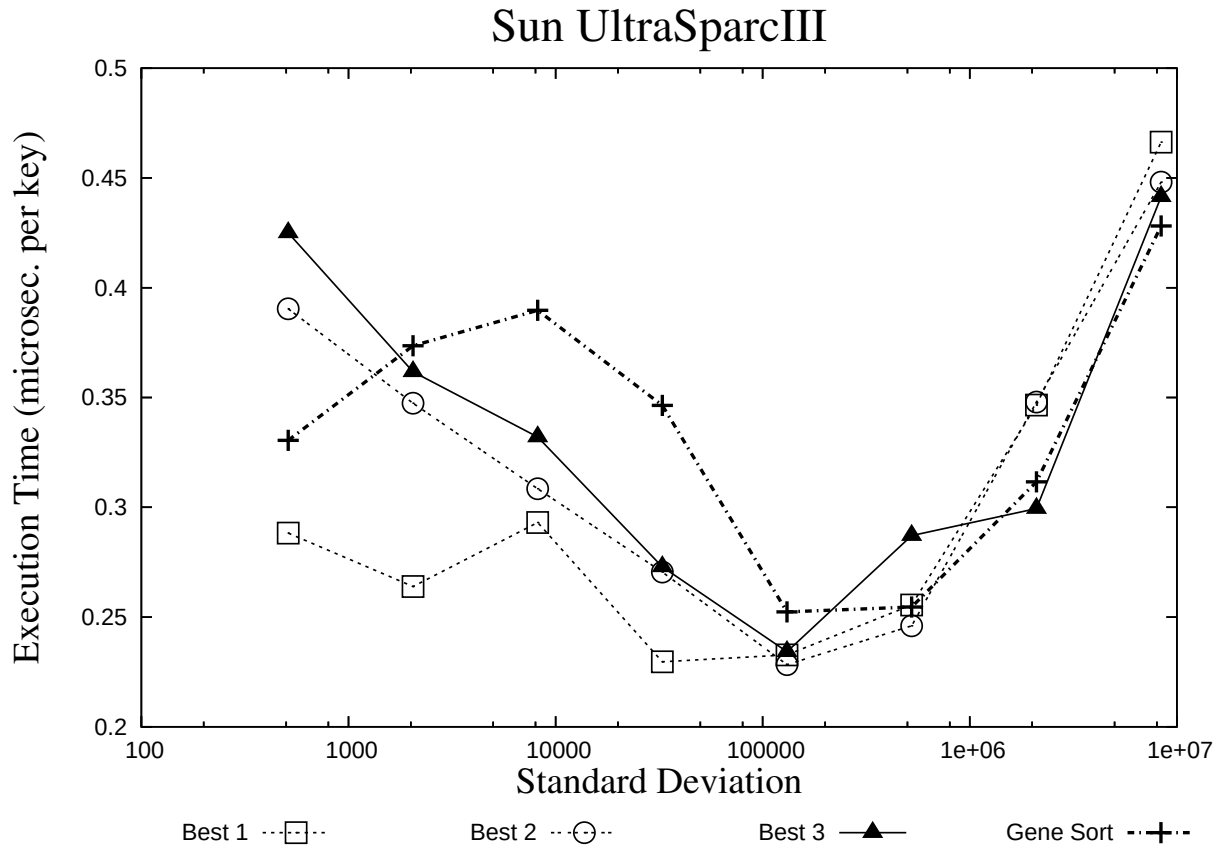


Figure 3.10. Performance variation of sorting genomes.

and entropy. To study this problem we selected some of the algorithms that our genetic algorithm found to be the fastest during the training process when sorting some inputs, but that were not selected as the best sorting algorithm in Table 3.4. Figure 3.10 shows execution times on Sun Ultrasparc III for three of these sorting genomes and Gene Sort as standard deviation changes. Figure 3.10 shows that Best 1 is the fastest for low values of standard deviation, but it performs poorly for high values. Best 2 is the fastest for middle values while Best 3 is the fastest for high values. Thus, these results show that a sorting algorithm composed of the best sorting genomes and the appropriate selection primitives can perform better than Gene Sort.

One of the limitations of our genetic algorithm is the fitness function that we use (Section 3.2.2). Our fitness function uses the average performance to select the fittest genome of the population. As a result, genomes with a stable performance are more fit than the genomes with a higher variability. On the other hand, we expected that the genetic algorithm would generate trees containing selection primitives to select the best algorithm for each range of input parameters. However, the results in Figure 3.10 show that our fitness function and the random search implemented in the genetic algorithm are not sufficient to discover this composite algorithm. In the next Section, we explain a possible solution to select the appropriate sorting genome based on the input characteristics.

$Divide - by - value$ and $divide - by - position$ are only used in two cases, specialized Sun UltraSparcIII, and general Intel Itanium 2. Branch primitives are also only used in two cases, specialized IBM Power 3 and general Intel Itanium 2. Also, the specialized algorithm is more complicated than the general one: the general sorting genome contains at the most three primitives, while the specialized one usually contains four primitives (Intel Xeon is the only exception since the specialized sorting genome is simpler than the general one).

$divide - by - radix$ with a large radix is the most frequently used primitive. A possible reason is that given that the keys being sorted are only 32 bits, choosing a large radix in the first pass, partitions the data in a way that the resulting partitions will most likely fit into the L2 cache and reduces the total number of passes required to sort the input. This first pass, though, is expensive. A large radix implies that the counters require large amount of memory. Large values of the radix may result in a high cache and TLB miss ratio [32]. However, the sorting routines generated with our genetic algorithm run faster than CC-radix, which used a radix of 5 in all the passes.

3.4 Classifier Sorting

In this Section we discuss how to use genetic algorithms to generate a classifier system [16, 52, 67] 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 $\vec{i} =< i_0, ..., i_k >$ and $\vec{c} =< c_0, ..., c_k >$ 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 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.

3.4.1 Representation

As we explained in Chapter 2 and outlined in Section 4.3, 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. Therefore, 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 algorithm selection 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. since we are using two bits to encode N, a maximum of four different sorting routines can be executed based on N, where each partition contains 3M keys ($\frac{16M-4M}{4} = 3M$). Thus, the algorithm executed when N is between $4M \sim 7M$ can be different from the one executed when N is between $7M \sim 10M$, and so on.

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.

3.4.2 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 [16, 67]. 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 [16, 67].

3.4.3 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 ($\geq 4M$).

3.4.4 Experimental Results

Standard deviation	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 3.5. Best genomes selected by the classifier sorting library.

Figure 3.11 compares the execution time of the algorithm generated in the form of a classifier (*Xsort*) versus Gene Sort and Adaptive Sort with the standard deviation = $2^n, n = 8, ..., 24$ when sorting 14M records. Xsort is almost always better than Adaptive sort. On the average of performance on all the test inputs, 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 3.5 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.

Let's assume we just use the number of keys encoded in four bits as the condition. Each bit can have three possible values, '0', '1', and '#'. '#' means don't care. Each rule is associated with two properties, predicted fitness and accuracy of the prediction. Assume we have the following rule A:

condition: 010#

action: (dr 21 (du 5 (ldr 6 20)))

fitness: 10

accuracy: 0.2

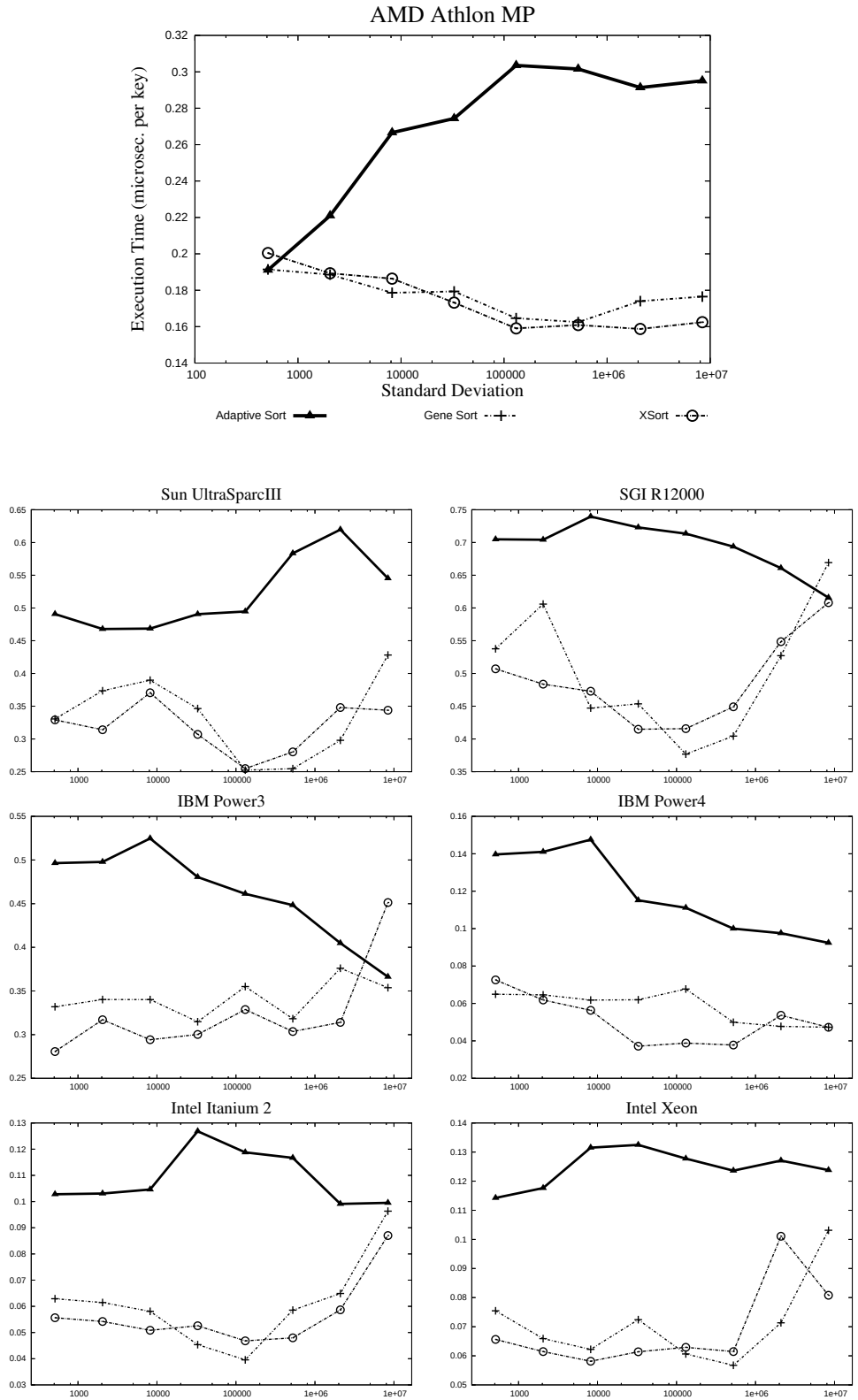


Figure 3.11. Xsort versus Gene and Adaptive Sort.

Which means if the number of keys falls into bucket 4 and 5 (9.5M-10.5M), rule A will predict that when using the genome ¨(dr 21 (du 5 (ldr 6 20)))¨to sort that particular input, we can expect the relative sorting performance to be 10.

Let's assume "(dr 21 (du 5 (ldr 6 20)))" is in fact the best genome only for bucket 5. The condition of this rule covers incorrect wide range. During exploration, the condition can be mutated with certain probability. Assume the mutated condition is 0101, which means the rule will now match the number of keys that falls into bucket 5. This random mutation corrects the error in the condition. The reverse can also be true. This is an example how the matching range of a rule can be changed through mutation.

During exploitation, we will test the performance of matching rules against some test inputs. The real performance will be used to update the predicted fitness and the accuracy of the matching rules. Those rules matching incorrect input cases will suffer from lower accuracy. The rule matching correct or smaller range of input cases will be awarded with higher accuracy. Let's assume the condition of the rule isn't affected by the mutation. After applying this rule on input with number of keys fall into bucket 4. The fitness and the accuracy will possibly decrease to 9 and 0.1. This will make this rule less likely to be selected at runtime when it matches inputs in bucket 4 and bucket 5.

Notice that the accuracy of the classifier learning system is somehow related to the transformed performance proposed by Leyton-Brown in [42] to minimize the error in the decision. However, our classifier system tries not only to minimize the error, if not also to improve performance.

3.5 Conclusion

In this chapter, 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 on average than any commercial routine that we have tried including the IBM ESSL, the INTEL MKL and the STL of C++, on inputs generated by selecting keys independently, using a Normal distribution. In term of the average performance on all inputs tested, our generated routines are 26% and 62% faster than the IBM ESSL in an IBM Power 3 and IBM Power 4, respectively.

Chapter 4

Matrix Multiplication

4.1 Introduction

Compilers have been very successful on automating the process of program optimization, but there is still a significant difference in performance between the code generated by the compiler and the hand-optimized code. The growing complexity of the architectural features of modern processors makes it very difficult to optimize performance. An approach that some researchers have followed is to use library generators to generate high performance code for some specific problem domains.

Examples of well-known library generators are ATLAS [66], PHiPAC [15], FFTW [26] and SPIRAL [69]. ATLAS and PHiPAC generate linear algebra routines and focus the optimization process on the matrix multiplication routine. During 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.

In the previous chapters we have built a library generator for sorting [43, 44]. Sorting is different from the algorithms implemented by the previous library generators in that performance of sorting depends not only on the target platform but also on characteristics of the input data, which are only known at runtime. In the work presented in Chapter 3 we used a classifier learning system to generate algorithms capable of adapting to the input data. In the work discussed herein, we follow a similar approach and use a classifier learning system to generate high performance libraries for matrix-matrix multiplication (MMM). Our library generator generates MMM routines that used recursive layouts [17, 61] and several levels of tiling. Our approach is to use a classifier learning system to search among all the different ways to partition the input matrices, the one that performs the best. The MMM routine generated with our classifier learning system uses different levels of tiling and tile sizes based on the dimensions of the matrices and the architectural features of the target machine.

ATLAS is a library generator that also produces a MMM routine. The difference between our approach and the

one followed by ATLAS is that we use recursive layouts to place the blocks in consecutive memory locations and focus the search on levels of tiling and size of each tile. ATLAS does not search for the number of levels of tiling. In fact, ATLAS only searches for the tile size for a single level of tiling, although a second level of tiling can be implemented [1]. Also, notice that the performance delivered by ATLAS in some platforms is still far from the one delivered by the vendor provided libraries [72], mainly because ATLAS does not take into account all the levels of the memory hierarchy and does not take advantage of some optimizations like prefetching. Our objective is to reduce the performance gap between the hand-optimized code and the automatic generated code by extending the search to consider parameters ignored by ATLAS.

When using a single level of tiling, it has been shown that a model can predict the best value of the tile size almost as well as the empirical search of ATLAS by simply taking into account certain cache parameters [71, 72]. However, when tiling for the different levels of the memory hierarchy, the size of the matrices becomes important. If the matrices are not a multiple of the tile sizes, we need to use padding or cleanup code. With padding, the size of the matrices is increased with additional rows or columns of zeros. Arithmetic operations are usually blindly performed on them. With cleanup, additional code (which is usually suboptimal) is executed to multiply the remainder rows or columns. With recursive layouts, padding is the method usually preferred. Given the large sizes of the second and third level of caches of current machines (6 to 8 MB), padding can represent a significant overhead if the tile sizes are computed without taking into account the matrix sizes. On the other hand, choosing the tile sizes based on the matrix sizes and disregarding the cache sizes will result in poor cache utilization. In addition, choosing the number of levels of tiling based on the number of caches of the machine may result in slow-downs. In some platforms it is better to use a single level of tiling because additional levels of tiling introduce additional instructions such as branches that may execute slowly.

We compared the MMM routine generated using a classifier learning system with the MMM routine generated by ATLAS when multiplying matrices of sizes 1000 to 5000. Our results show that the MMM routine generated using the approach we follow in this chapter runs always faster than ATLAS in a Sun UltraSparc III and faster on the average by 18%. In the case of Intel Pentium Xeon, our routine is almost always faster than ATLAS and the average speedup is 5%. However, ATLAS runs on average 14% faster than our routine in Intel Itanium II. Our experiments also show that padding is important to obtain high performance, and we plan to implement more sophisticated padding strategies to improve the performance of the generated library.

This chapter is organized as follows. Section 4.2 revises some of the compiler optimizations that are applied to MMM. Section 4.3 presents the partition primitives that will be used by the classifier learning system, which is presented in Section 4.4. Section 4.5 presents our experimental setup and results. Finally, Section 4.6 concludes.

4.2 Matrix-Matrix Multiplication

In this Section we present an overview of an automatic tiling and discuss copying and recursive layouts in the context of matrix-matrix multiplication.

A naïve implementation of matrix-matrix multiplication is shown in Figure 4.1. Usually this code runs slowly because of the poor utilization of cache memories. A transformation used to increase cache locality is loop tiling. This transformation was first introduced by McKellar and Coffman [45] and discussed in the context of compilers by Abu-Sufah [6] and later by Wolfe [68]. Tiling can be used to reduce the amount of data accessed between two references to the same memory location so that the data can be reused from the cache when referenced the second time. Figure 4.2-(a) shows the code for a tiled matrix-matrix multiplication using a square tile of size $NB \times NB$. Instead of operating on the whole rows or columns of the matrices, tiling operates on smaller matrices, as shown in Figure 4.2-(b). The Figure shows with black dots the data accessed when executing the two inner-most loops of the tiled code in Figure 4.2-(a).

```
for (j = 0; j < M; j++)
  for (i = 0; i < N; i++)
    for (k = 0; k < K; k++)
      C[i][j] = C[i][j] + A[i][k] * B[k][j]
```

Figure 4.1. Matrix Multiplication Code.

The size of the tile must be chosen to minimize capacity misses. In the example in Figure 4.2-(a), the best tile size is such that the $NB \times NB$ tile of A accessed by each iteration of the jj loop remains in the cache after it is brought into the cache by the first iteration of the loop.

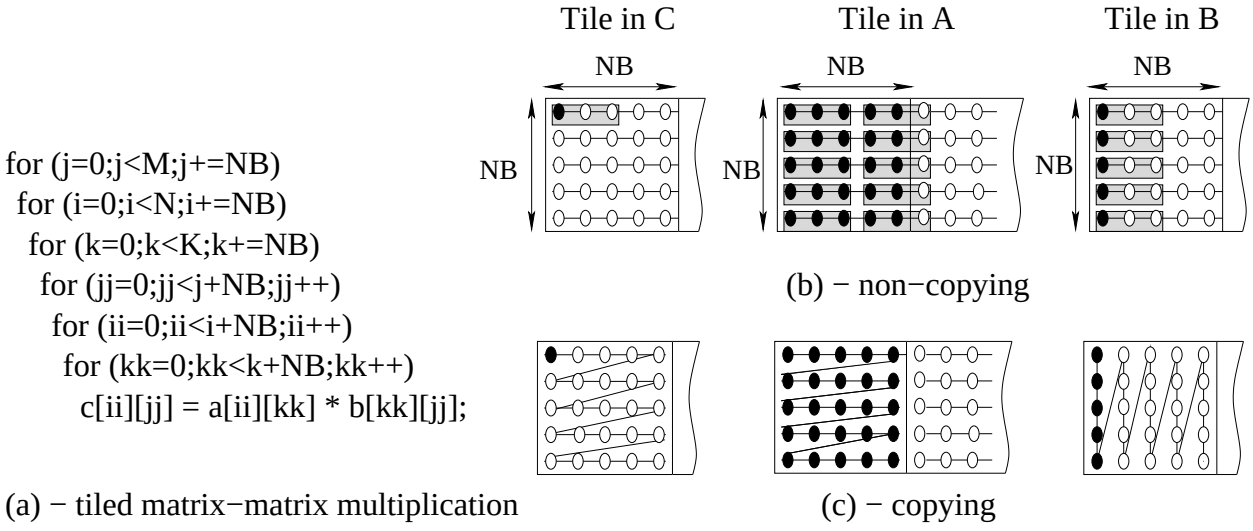


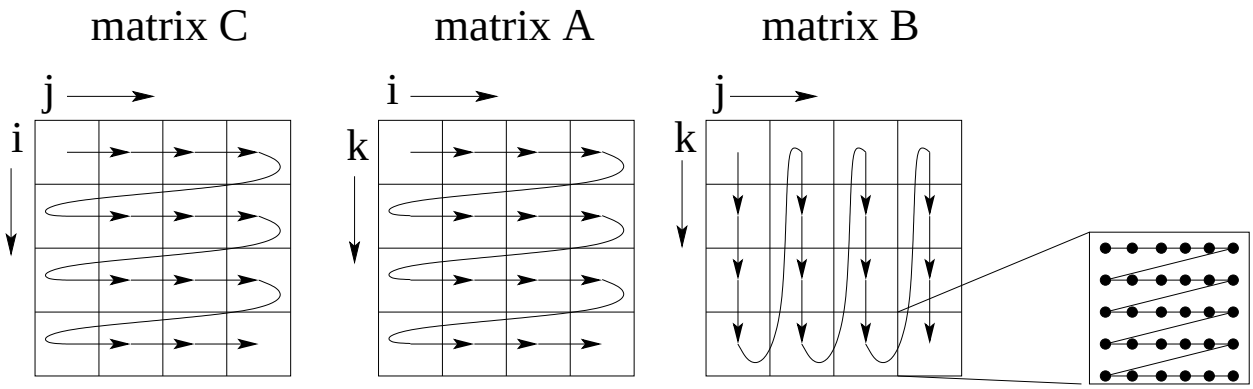
Figure 4.2. Tiled matrix matrix multiplication. (a)-code. (b)-tiles accessed and memory layout when using a row major layout and non copying. (c)-memory layout when elements in the tiles are copied to consecutive memory locations.

Tiling decreases capacity misses in MMM, but when used with a row or column major layouts, cache memories can be underutilized. In fact, cache lines contain several elements and, as a result, elements that are never referenced may be brought to the cache due to a cache miss. Figure 4.2-(b) shows an example for the row-major layout where cache lines contain 3 elements (shown in a shadow rectangle). The Figure shows in white dots the elements brought to the cache that are never referenced. This results in poor spatial locality. In addition, row or column major layout may also cause cache conflict and TLB misses. Conflict misses occur because some cache lines are mapped to the same set of the cache. TLB misses can occur if the size of matrices is large enough so that each row (in a row major layout) of matrices is in a different physical page. This problem can be avoided if the tile selection considers the number of entries in the TLB in conjunction with the cache size [46]. In any case, to reduce conflict and TLB misses, tiling is usually used in combination with copying [38, 65] where the elements of each $NB \times NB$ submatrix are copied into contiguous memory locations, as shown in Figure 4.2-(c). Then, this layout is usually refereed as block data layout. Copying can also be done in a tiled way, where NB cache lines are read and transposed in each tile, though no significant performance can be gained from this approach for large matrices.

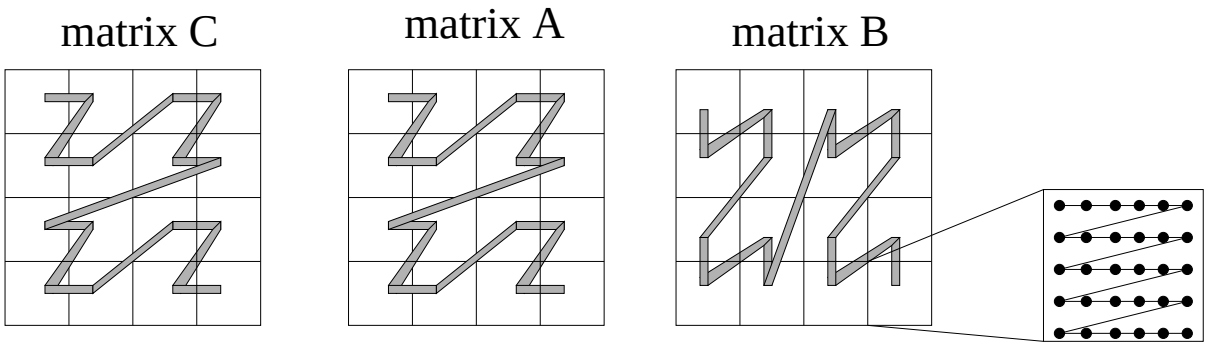
Tiling has been extensively considered in the literature when applied to a single cache level [18, 38, 49, 56, 71]. However, when tiling for a single level of cache, we do not exploit all the cache levels. For example, Figure 4.3-(a), shows the order in which the submatrices of A, B and C are accessed when executing the code of Figure 4.2-(a). Each iteration of the outermost loop (j) will traverse the 16 blocks of matrix A. Unfortunately, if matrix A is large, it will not fit in the second level cache. Therefore each j iteration will have to bring all the A blocks back to the second and first cache level. A solution to this problem is to apply another level of tiling [56, 70, 10].

Suppose that we apply another level of tiling to the code in Figure 4.2-(a) by adding three additional loops with the same order JIK. The outer loops would operate on blocks consisting of 2×2 tiles so that the blocks of matrix A will be traversed in the order shown in Figure 4.3-(b). The blocks of the second level of tiling are no longer consecutive in memory and, as a result, these accesses can result in cache conflicts and TLB misses [50]. To avoid this problem, nonlinear array layout or recursive layouts together with tiling have been used [17, 61]. The idea is to copy these blocks into consecutive memory locations. These array layouts are described as based on quadtrees [25] or on space-filling-curves[30, 51, 58]. Instances of this family are familiar in parallel computing under the names Morton ordering and Hilbert Ordering. The layout shown in Figure 4.3-(b) for matrix A is known as Z-Morton. These recursive layouts were shown to deliver high performance [17, 61], but some considerations need to be taken into account in their implementation:

- These nonlinear layouts can be applied recursively down to the level of individual matrix elements [25]. However, Chatterjee et al. [61] showed that this was counter-productive, and that it is better to follow a recursive layout only until the tile fits in the cache.



(a) One level of Tiling



(b) Two levels of tiling. Recursive layouts

Figure 4.3. Memory layouts for tiled matrix-matrix multiplication. (a)- One level of tiling and block data layout. (b)- Two levels of tiling and recursive layout.

- These recursive layouts require that for a matrix of size $M \times N$ and a tile of size $tm \times tn$, the following equations be satisfied: $\frac{M}{tm} = \frac{N}{tn} = 2^d$. Sometimes it is necessary to add padding to the matrix in order to satisfy this equation. The general idea is to select the appropriate tile $tm \times tn$ for the cache of the machine, insert a zero padding and perform the arithmetic operations on the zero padding.

4.3 Partition Primitives

The library generator used in this study produces a matrix-matrix multiplication (MMM) routine that computes $C = \alpha AB + \beta C$, where A, B and C are matrices of dimensions $M \times K$, $K \times N$ and $M \times N$ respectively. The generated MMM routine uses multilevel tiling and recursive layouts as discussed above. The matrices are decomposed into submatrices of matching sizes, with the help of padding, and the matrix product is represented as a product of matrices whose elements are submatrices. The elements of each submatrix is stored in consecutive memory locations. Then the same transformation is recursively applied to the submatrices, which elements are also reordered accordingly. After how the matrices are partitioned is settled, the routine first copies the original matrices from row or column major layout to the recursive layout. Then, it multiplies the matrices and transforms the resulting C matrix back to the row or column major layout. The copy and multiplication procedures are determined by the number of levels of tiling and tile sizes. These values will be selected using empirical search as discussed below. This Section describes the partition primitives which will be used by the search procedure to determine the best number of levels of tiles and tile sizes for the dimensions of the input matrices and target architecture. Before explaining the primitive partitions, we briefly describe the procedures for copying and padding.

We denote the matrix dimensions at level i as M_i, N_i and K_i, for where i ranges from 1 to the number of levels of tiling. If the matrices at level i are partitioned with factors $p(m)_i$, $p(n)_i$ and $p(k)_i$, the dimensions of each submatrix in the next recursion level will be $M_{i-1} = \frac{M_i}{p(m)_i}$, $N_{i-1} = \frac{N_i}{p(n)_i}$ and $K_{i-1} = \frac{K_i}{p(k)_i}$ respectively. The partition factors determine how the sub-blocks must be copied from row (or column) major layout to the recursive layout. An example of these recursive layouts has been shown in Figure 4.3-(b).

As an example Figure 4.4 shows how matrix A is partitioned using two levels of partition. The partition vector for level 2 is (4,2,4) and for level 1 is (2,4,2). We apply first the partition vector for the second level. As a result, each dimension of matrix A is partitioned in 4 pieces. Then, each of the resulting sub-matrices is divided in 2 pieces. The figure shows the partitioning and the recursive layout using the Z-Morton layout. The elements in each block will be consecutive in memory, that is, they will use a block layout.

When the factors in the partition vector are not a divisor of the matrix dimensions we need to use padding. For example suppose A is a matrix of 2000×1000, and we divide it first by (3,2) and then (4,1). Since 3 is not a divisor

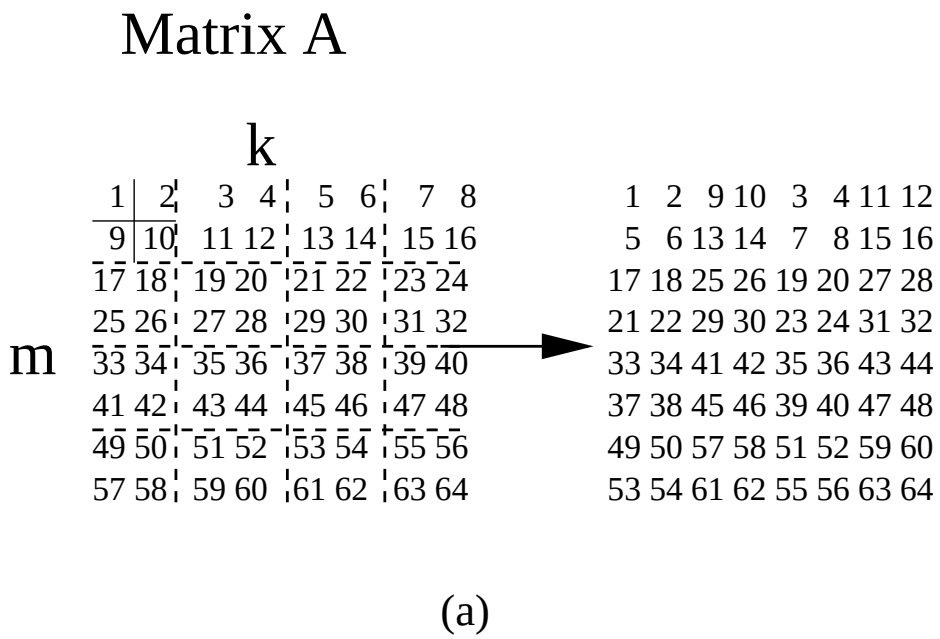


Figure 4.4. Example of recursive layout

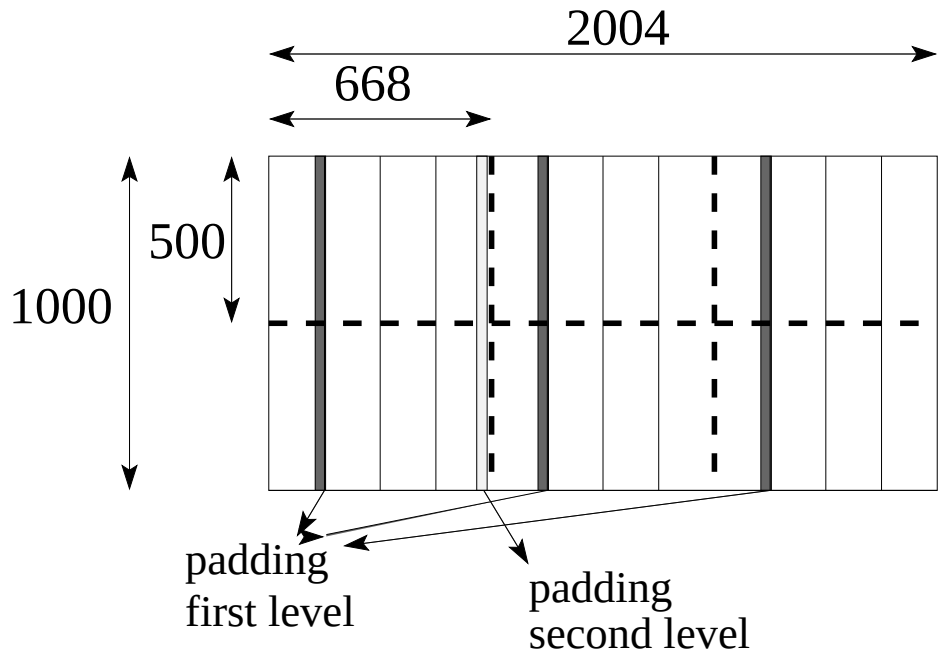


Figure 4.5. Example of padding

of 2000, we need to add padding so that we can divide the matrix in exactly 3 pieces. Each resulting submatrix will be of size 667×500. Now, the 667 elements of the X dimension need to be divided by 4. Since 4 is not a divisor of 667, we need to pad each submatrix, and make them to be 668. Thus, we end up with a matrix of size 2004×500. We have 4 additional columns of zeroes which will be blindly multiplied. The example is shown in Figure 4.5.

Next, we describe the partition primitives that we use in this work.

1. `Partition by Block`(PB)

This primitive specifies the tile or block size. It has three parameters, which are the block size for each M, N and K dimension. So, consider $M = 100, N = 100, K = 40$. If we want tiles of sizes 50, 50, and 20 for the dimensions M, N, and K, respectively, we would specify this as follows `Partition By Block (50,50,20)`. The `Partition By Block` primitive will compute the partition factors (p(m),p(n),p(k)) as follows:

$$p(m) = \lfloor \tfrac{m}{bm} \rfloor, p(n) = \lfloor \tfrac{n}{bn} \rfloor, p(k) = \lfloor \tfrac{k}{bk} \rfloor$$

The `Partition by Block` primitive allows to specify tiles of any size, not only square tiles.

2. `Partition by Size`(PS)

This partition primitive specifies the size of a block and partitions the different dimensions of the matrix until the resulting submatrices are equal or smaller than the size of the specified block. The primitive guarantees that the ratio

between the dimensions is kept constant. The primitive allows the specification of the dimensions to be partitioned. It has four parameters. The first three parameters specify if a given dimension M, N and K needs to be partitioned. The fourth parameter specifies the block size. The algorithm used by this primitive is shown in Figure 4.6.

```
Input Parameters:
    m,n,k: input matrix dimensions
    muse, nuse, kuse: boolean variables indicating the
                      dimensions to be partitioned
    size: the block size

begin
    maxratio = MIN(m,n,k)/2
    for(ratio=maxratio; ratio ≥ 2;ratio--){
       if(muse) tmp(m)=⌈m/ratio⌉
       if(nuse) tmp(n)=⌈n/ratio⌉
       if(kuse) tmp(k)=⌈k/ratio⌉
       tmpsize=tmp(m)*tmp(k)+tmp(k)*tmp(n)+tmp(m)*tmp(n)
       if(tmpsize ≤ size)
          break;
    }
    if(muse) p(m) = ratio;
    if(nuse) p(n) = ratio;
    if(kuse) p(k) = ratio;
end
```

Figure 4.6. Partition by Size Primitive

Notice that most of the previous research on recursive layouts works by dividing each dimension by half. The `Partition by Size` primitive is a generalization of the `divide by half` strategy which can be implemented by setting $muse = nuse = kuse = true$ and $size = \frac{m*k+k*n+m*n}{4}$. In some studies, the recursion is carried down all the way to the individual elements [25, 27]. The work in [27] showed that this strategy resulted in minimum number of cache misses. Unfortunately in this case minimizing the number of misses does not necessarily results in better performance, because of the additional instructions that need to be executed. In fact, the work by Chaterjee et al. [17, 61] showed that stopping the recursion at tiles of the appropriate size returned better performance. In this chapter, when generating the kernel routine for the MMM we will follow the approach of Chatterjee et al.(Section 4.5).

4.4 Classifier Learning System

To build a high performance library we need to determine how the input matrices should be partitioned along the M, N and K dimensions. The best partitioning is a function of architectural features such as number of caches and size of

each cache and the dimensions of the input matrices. Choosing the correct partition is hard. For some machines, we need to apply a single level of tiling, since the overhead of the additional instructions executed when more levels of tiling are applied results in lower performance. Even when tiling for a single level of cache we need to decide whether to tile for L1 or L2 [2, 71]. When tiling for L2 and L3, it is important to take into account the dimensions of the matrices. Since L2 and L3 tend to be large (sometimes 6 or 8 MB), when a dimension of the matrix is not a multiple of the tile size, the amount of padding can be substantial.

We plan to use the partition primitives described in the previous Section as the building blocks to generate a MMM library. By combining the different primitives and selecting different parameter values, the space of the different algorithms that we can generate is very large. As a result, exhaustive search is unfeasible. Our approach is to use a classifier learning system [16, 52, 67] to search the space of possible algorithms. The main reason to use a classifier learning system is that with this mechanism input characteristics can be used to create a table of the best partitioning parameters that can be used at runtime to enable dynamic adaptation.

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 $match(i, c)$ as follows:

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

If there is only one $match(i, c)$ which is $true$, the action corresponding to the condition bit string c is selected. However, for a given input several matches are possible. In this case, we will choose one action among all the rules that match. The mechanism for the selection is explained below (in Section 4.4.3).

Next we explain how the classifier learning system is tuned for each platform and input

4.4.1 Representation

Encoding of the Rule Condition

The input characteristic that will determine the parameter values of the partition primitives is the dimension of the matrices. Thus, we will encode possible values of the dimensions of the matrices A, B and C in the condition of the rules.

Action of the Rule.

The action part will be a list of the partition primitives `partition by size (PS)` or `partition by block (PB)` with their corresponding parameter values. For example, an action will have the shape `(PS param-list (PB param-list))`, where `param-list` is the list of parameters. This action will return a *single function* that will decompose the input matrices of size $M \times N \times K$ into submatrices of size $M' \times N' \times K'$, that result from applying first the `PS` primitive and them the `PB` primitive.

Notice that each action, even if it contains several partition primitives, correspond to a single level of tiling. To apply several levels of tiling, we can recursively invoke the rule set of the classifier system with the size of the resulting submatrices. The recursion will finish when the number of levels of tiling has already reach the maximum number of levels allowed, or when the size of the submatrices is within a predefined range.

4.4.2 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 partition scheme based on the input characteristics.

During the training process we generate matrices of different sizes. Given a training input, we have a match rule set, which are the set of rules where the condition matches the bit string that encodes the input characteristics. We use a XCS classifier learning system as the one in [16, 67]. In this type of classifier systems, each rule has two attributes. The first attribute is the fitness. The fitness is an estimation of the performance of this rule on the inputs that match the associated condition. The second attribute is the accuracy. The accuracy measures the confidence of the fitness attribute in predicting the correct performance.

In our approach we use a multi-step classifier system, since the output of an invocation can be used as the input for the next invocation. This system works as follows. The first time we invoke the rule set with a training input we have a match rule set. All the actions in the matching rules are the set of strategies that can be used to partition the input matrices. During the training process, all the actions in the matching rules are applied. Thus, given an input of size $M \times \mathbb{N} \times K$, the result will be submatrices of sizes $M'_i \times \mathbb{N}'_i \times K'_i$, where i=1..number of matching rules. Each of the $M'_i \times \mathbb{N}'_i \times K'_i$ generated outputs can be used as the input to the next invocation to the learning classifier system. The system, as explained above, will stop when the maximum level of calls is reached or when the size of the submatrices is within a specified range. At the end, we have many different partition strategies, each of them blocking the matrices with tiles of different sizes, and possibly different levels of tiling. We generate the MMM routine for each partition strategy and measure the execution time. Based on the results obtained, we update the fitness and accuracy of all matching rules used to generate each of the MMM routines. The algorithm is shown in Figure 4.7.

Multi_Step_Classifier_Learning
Inputs:
M,N,K: dimensions of the input matrices
l: current level of recursion
Outputs:
$p(m)_i, p(n)_i, p(k)_i$, i=[0..max-num-levels]: partition factors
$exec$: execution time
begin
P= variable that contains the partition factors —$p(m)_i, p(n)_i, p(k)_i$, i=[0..max-num-levels]
Encode M,N,K into the bit string $\overrightarrow{in}$
$mset = \varnothing$
for each rule r
$\overrightarrow{rcond}$ = condition of r
if $match(\overrightarrow{in}, \overrightarrow{rcond})$
add r to $mset$
while ($mset \neq \varnothing$)
extract r from $mset$
act= action part in r
$p(m)_i, p(n)_i, p(k)_i$= result of applying act on M, N, K
Update P with the new $p(m)_i, p(n)_i, p(k)_i$
M', N', K'= result of applying $p(m)_i, p(n)_i, p(k)_i$ on M, N, K
if $notend$ then
call $Multi\_Step\_Classifier\_Learning$ ($M', N', K', l + 1$)
else
Run matrix multiply with M, N, K using P
Measure execution time $exec$
Use $exec$ to update $fitness$ and $accuracy$ of r
return exec **end**

Figure 4.7. Classifier learning algorithm

To generate new conditions and actions, transformations such as mutation and crossover applied in genetic algorithms [28, 44] are also used here. Details about how this algorithm works can be found in [16, 67].

4.4.3 Runtime

At the end of the training phase we have a tuned rule set. At runtime, the bit string encoding the input characteristics will be used to extract all the rules whose condition matches the input. Among all these rules, the one selected will depend on a function that rewards low execution time and penalizes low accuracy. The runtime overhead includes the computation of the input bit string, and the scan of the rule set to select the best one.

4.5 Experiments

In this section we evaluate our approach of using a classifier learning system to optimize a MMM routine. In Section 4.5.1 we discuss the environmental setup that we use for the evaluation and in Section 4.5.2 we present performance results.

4.5.1 Environmental Setup

We evaluated our approach on three different platforms: Sun UltraSparc III, Intel Itanium 2, and Intel Xeon. Table 4.1 lists for each platform the main architectural parameters, the operating system, the compiler and the compiler options used for the experiments.

	Sun	Intel	Intel
CPU	UltraSparcIII	Itanium 2	P4 Intel Xeon
Frequency	750MHz	1.5GHz	3GHz
L1d/L1i Cache	64KB/32KB	16KB/16KB	8KB/12KB (1)
L2 Cache	1MB	256KB (2)	512KB
Memory	4GB	8GB	2GB
OS	SunOS5.8	RedHat7.2	RedHat3.2.3
Compiler	Workshop cc 5.0	gcc3.3.2	gcc3.4.1
Options	-native -xO5	-O3	-O3

Table 4.1. Test Platforms. (1) Intel Xeon has a 8KB trace cache instead of a L1 instruction cache. (2) Intel Itanium2 has a L3 cache of 6MB.

To generate the MMM library we used the classifier learning system. We trained the classifier with the algorithm of Figure 4.7. The classifier determines the number of levels of tiling and the tile size for each matrix size. For the implementation of the MMM at the last level of tiling we used the kernel generated by ATLAS. ATLAS generates a MMM routine and uses empirical search to look for the best parameter values of certain compiler transformations

such as tile size, loop unrolling and software pipelining [66, 71, 72]. The kernel in ATLAS produces code for a MMM routine with a single level of tiling and square tiles. Thus, in our MMM library the submatrices in the last level of tiling must also be square. We allow these submatrices to be in the range of 40 - 120, since this range cover most of the different values that ATLAS finds for current platforms [72]. ATLAS generates a single MMM routine and searches for the tile size that obtains the best performance results. In our system, the tile size of the last level is determined by the classifier learning system, but we use ATLAS to search for the rest of the other parameters for each tile size in the range 40 - 120. We limited the maximum number of levels of tiling to be 3, since current architectures have three or less levels of cache, and our experiments showed that increasing the level of tiling beyond 3, resulted in less performance. Apart from this, after we determine the partitioning strategy, we need to copy the tiles to the corresponding recursive layout. In this work we use the Z-Morton layout. In particular, The tiles of matrix A is stored in "row major" and the tiles of matrix B is stored in "column major". Then the same order is used recursively within tiles. In a longer study we could also search for the best layout. When the matrix is not a multiple of the tiling we insert padding, as shown in Figure 4.5. Padding can also be necessary to obtain a square tile at the last level of tiling.

To encode the size of the matrices, we used 13 bits per dimension, though we consider only matrices with size up to 5000. Since we have 3 dimensions $M \times N \times K$, we used a total of 39 bits. Initially we generated 1000 rules, and we randomly generated the condition and the action part of each rule. For the training we randomly generated matrices whose sizes were between 1000 and 5000. We did not specify any condition to end the training process. Instead, we let the training run for a certain amount of time. In the experiments reported here, we let it run for 1 week.

We compare the MMM routine generated by our classifier learning system with three different approaches:

- L1, where the MMM routine has a single level of tiling.
- L2, where the MMM routine has two levels of tiling.
- ATLAS.

To make a fair comparison with L1 and L2 approaches we used ATLAS to generate the kernel of the MMM routine. In both cases we used the same copying strategy and padding as the one used in the MMM routine generated using the classifier. For the L1 approach we used the tile size that ATLAS found to be the best. For the L2 approach we used the value found by ATLAS for the first level of tiling. For the second level of tiling we chose the size so that $Tile2 = K \times Tile1$. We selected K so that $Tile2$ is multiple of $Tile1$, and smaller than the value that results from resolving the inequality $3 * Tile2^2 \leq CacheSize$. The exception is Sun UltraSparc III. This machine has a large L2 cache (1 MB) and selecting the $Tile2$ using the previous formula resulted in low performance, since padding represented a large overhead in some cases. We decided to select for the Sun UltraSparc a tile of size 1/3 of the computed value using the previous formulas. Table 4.2 shows the values used for each $Tile1$ and $Tile2$. In both L1

and L2 we allowed the $Tile1$ to vary within the value reported in the Table and $+/-10$. We varied the size of the $Tile1$ based on the matrix size to minimize the amount of padding.

	UltraSparcIII	Itanium 2	P4 Intel Xeon
L1_Tile	68	120	60
L2_Tile	380	240	240

Table 4.2. Tile Sizes.

For ATLAS we used the code produced by the ATLAS Code Generator using empirical search. ATLAS can also use hand tuned BLAS routines. When ATLAS is installed these hand-coded routines are also executed and evaluated. However, since in this work we are only interested on the comparison on the MMM routine generated by ATLAS, we only used the code generator, without hand-coded code. Notice, that ATLAS can have a L2 Cache Blocking parameter by setting a variable called CacheEdge. For the ATLAS experiments, we set this variable to the appropriate value as reported in [1].

4.5.2 Experimental Results

Figure 4.8 presents the performance results of the four MMM routines described in the previous Section: L1, L2, Classifier and ATLAS. For the experiments we multiplied square matrices whose sizes vary from 1000 to 5000, in steps of 100.

The results vary from platform to platform. In the case of the Sun UltraSparc, Classifier is always the best. For this platform L2 is also better than ATLAS and L1. For Itanium 2, the code generated by ATLAS performs better than any of the other routines. Only in a few points that of the Classifier is equal or better. For Intel Xeon, the code generated by the Classifier is usually the fastest, followed by that of ATLAS.

It has been stated [57] that tiling for L1 was enough and that multi-level tiling was not necessary. However, our results for Sun UltraSparc III show that multi-level tiling can improve performance over one level of tiling, since L2 and Classifier are always the best approaches for this platform. For the other two platforms it is not clear if multilevel tiling is better.

The performance results for the Intel platforms Itanium 2 and Xeon shows high variability in performance for the code generated by Classifier, L1 and L2. Since these 3 approaches use padding when the dimensions of the matrices are not multiple of the tile sizes, while ATLAS (whose performance is very stable) uses cleanup code, we think that the variability is due to the fact that the amount of padding changes for the different matrices being multiplied. We need to conduct further experiments to verify this. Also, in the future we plan to study different strategies to pad the matrices more efficiently. For example, we can concentrate all the padding at the end of the matrix, instead of distribute it in each tile, as we have done in the routine in this chapter. We will also study the possibility of combining cleanup code

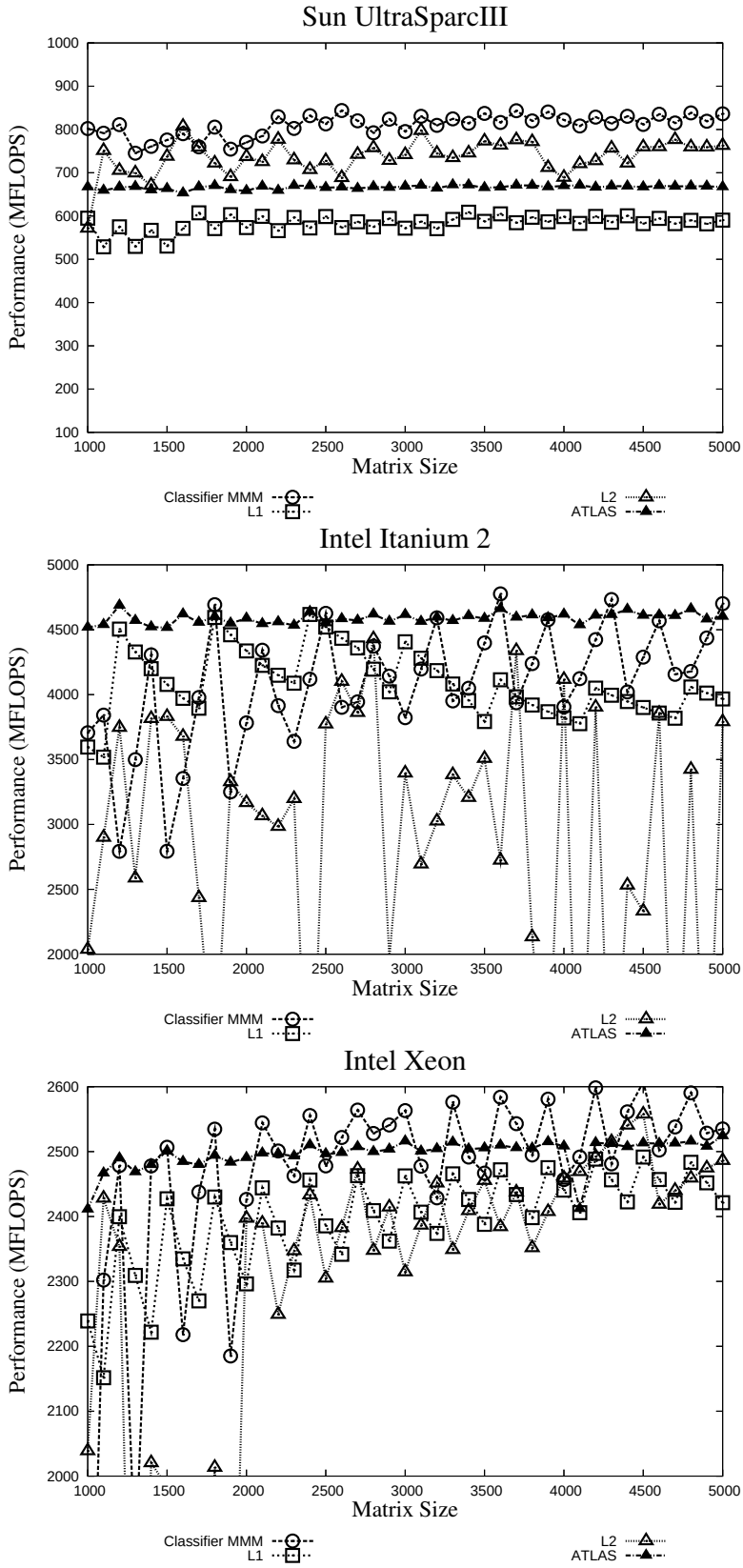


Figure 4.8. Performance Results

with recursive layouts. If we find out that performance is highly dependent on the padding or clean up strategies, we can also search in this space.

Overall, our results show that the MMM routine generated using the approach we follow in this chapter runs always faster than the code generated by ATLAS in a Sun UltraSparc III and the average speedup across all matrix sizes is 18%. In the case of an Intel Pentium Xeon, our routine runs almost always faster than ATLAS by an average of 5%. However, ATLAS runs 14% faster than our routine in Intel Itanium II. In the future, we will also add more platforms to this study.

4.6 Conclusions

We have presented how we have generated a MMM routine using a classifier learning system. The MMM routine generated with our classifier learning system uses different levels of tiling and tile sizes based on the dimensions of the matrices and the architectural features of the machine where it is installed.

We compared the MMM routine generated using a classifier learning system with the MMM routine generated by ATLAS when multiplying matrices of sizes 1000 to 5000. Our results show that the MMM routine generated using the classifier runs always faster than ATLAS in a Sun UltraSparc III by an average of 18%. In the case of an Intel Pentium Xeon, our routine runs almost always faster than ATLAS by an average of 5%. However, ATLAS runs on average 14% faster than our routine in Intel Itanium II. Our experiments also show that padding is important to obtain high performance, and we plan to implement more sophisticated padding strategies to improve the performance of the generated library.

Chapter 5

Conclusions and directions for future research

A major challenge faced by compiler researcher is how to generate efficient code for increasingly complex architectures. If we look into the future, the task is even more daunting. At the horizon are problems like extreme levels of on-chip parallelism, deep, partitioned cache hierarchies and re-configurable architecture. These novel technologies will complicate the analysis, modeling and generation of efficient code. Library generation is a promising technique for generating high performance code for existing and future platforms. Previous studies, such as ATLAS and SPIRAL, have achieved significant successes in generating efficient code for problems whose performance does not depend on input values. However, the introduction of input values greatly complicates the efforts to generate efficient code for problems such as sorting. The complexity comes from both the uncertainty of input values at runtime and the difficulty of analyzing the relationship between performance and input values statically.

Machine learning techniques bridges the gap between traditional code generation techniques and the desired adaptability to input data. Why machine learning is suitable for this task? The reasons are that machine learning inherently adapts to complex relationships, as the relationship between performance and input values, and that machine learning makes empirical predictions and predicts better with more data. These two features of machine learning are just the goal of the tuning process in library generation.

5.1 Contributions

This thesis presents novel techniques that combine machine learning and empirical code generation, and applies the new method on sorting and recursive matrix-matrix multiplication. We use machine learning techniques to learn the mapping between the performance of alternative algorithms and the input characteristics. We are able to select the best algorithm at the runtime based on architectural features and real input. Moreover, we propose the idea to build hybrid algorithms based on a set of pure algorithm candidates and adapt the hybrid algorithm to factors that affect the performance, including architecture and input characteristics. Machine learning techniques serve as the backbone in our hybrid library generation system.

Our techniques that marry machine learning and empirical library generation deliver good performance. The

sorting library generated using our techniques outperforms the best sorting routine in vendor-provided libraries on the inputs we tested. Our work on recursive matrix-matrix multiplication is among the first efforts to automatically tune matrix-matrix multiplication routines based on recursive layout. The performance of our library matches that of empirically tuned libraries, ATLAS. Equally important is that our machine learning based techniques tune and generate the highly-efficient code automatically, using only a small fraction of manual tuning time.

5.2 Future Directions

The major theme of future work along the direction portrayed in this thesis is to develop machine learning techniques that can understand program behavior. We are facing major challenges if we want to squeeze the last ounce of performance from future architectures. First, architectures are increasingly complex. In particular, speculative features, such as complex prefetch patterns and smart buffers in the memory hierarchy, blur the picture how a program runs. We no longer have the luxury to use simple models to formulate the interaction between programs and architectures. We need new methods to describe and give us insight on how programs run in speculative environments.

Next we discuss how the techniques presented in this thesis can be applied in the two other directions.

5.2.1 Multi-core Parallelism

Multi-core processors, such as Intel Core [5] and Sun Niagara [53], is a new technology that emerged not long ago. Multi-core is a result of the facts that architectural tricks no longer increase IPC, also that the performance gain in single-core processor is at the price of exponential increase of power consumption. In theory, multi-core processors can deliver higher performance at lower frequency, which reduce the pressure from the increasingly tight power constraint. Moreover, However, we can not achieve the performance potential in multi-core processors freely. The main difference between current multi-core architecture and previous parallel architecture is that key components in a processor, including cache, memory I/O unit, are shared between multiple cores. Cores in a processor communicate with each other much faster and more efficiently than previous SMP systems. This extensive resource sharing adds an additional level of complexity to the analysis and generation of efficient code on multi-core processors. Libraries on multi-core processors are almost exclusively manually tuned. Automatic library generators, which are highly successful on single processor, have no or only limited support for these new parallel platforms.

The techniques proposed in this thesis, in particular the technique that generates efficient hybrid code, has good potentials for generating high-performance libraries on multi-core processors. The advantage is that our technique can find the best tradeoffs of resource competition automatically and efficiently. The identification of tradeoffs usually is the major barrier that keeps traditional library generation techniques from being ported to multi-core processors. On

the other hand, the major challenge in applying the technique to multi-core processors is that we need to design basic operations that can represent competing relationship between resources. In addition to that, more basic operations means larger space. We need to design machine learning technique that can work in the larger space efficiently.

5.2.2 Software/Architecture Co-design

We usually look at only how to make software more efficient for an architecture. Now more and more attentions are paid to the question on the other side, what is the best architecture for a program. For example, compiling for FPGA [64, 12] is in fact to design an architecture for program. Given the capability to design both hardware and software, we have more freedom to design a computing system that suits a specific problem. However, more freedom also means harder to find the best design tradeoffs, and achieve the full potential of the co-designed system. In other words, we need to find the best software implementation for every possible hardware design, and vice versa. To co-design software and architecture, we need fast knowledge how a program will behave on changing platforms. It is infeasible to manually make out all models we need.

This thesis has presented how machine learning can be used to generate efficient code for a given architecture. However, if we want to apply the similar techniques to the reverse problem, that is, search for the best hardware design for given code, we need to define what are primitives in hardware design that represent all alternatives in hardware design. Moreover, the relationship between the primitives and the performance of code needs to be well defined so that the goal of the search is clear and the search process can be geared towards the most efficient co-design.

5.2.3 Long-term Goal

Machine learning techniques will play an important role in helping us understand the interaction between programs, architectures and inputs. Machine learning can identify the structures from data, even the data is from speculative sources, helping us figure out models for underlying problems, and tell which factors are important in determining the best tradeoff. Moreover, this is an automatic process.

In the long run, we want to know how programs interact with architecture and input, and how to leverage machine learning technique to dig out the optimization opportunities. Our long-term goal is to generalize from the extensive study of individual applications and provide a fundamental understanding of the behavior of general programs and knowledges on how to generate efficient code from the understanding.

References

[1] ATLAS home page. [Online]. http://math-atlas.sourceforge.net/errata.html#tuneCE.

[2] ATLAS home page. [Online]. http://math-atlas.sourceforge.net/faq.html#NB80.

[3] Sprng: Scalable parallel pseudo random number generators library.

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

[5] Intel core architecture, http://www.intel.com/technology/computing/multi-core, 2006.

[6] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the Performance Enhancememt of Paging Systems through Program Analysis and Transformations. *IEEE Transactions on Computers*, 30(5):341–356, May 1981.

[7] Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. pages 305–314, 1987.

[8] Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. pages 204–216, 1987.

[9] A. Aggarwal and S. V. Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[10] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *International Journal of Parallel Programming*, 29(5):493–544, 2001.

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

[12] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 70–80, Los Alamitos, CA, 1999. IEEE Computer Society Press.

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

[14] K. Berman and J. Paul. *Fundamentals of Sequential and Parallel Algorithms*. PWS Publishing Co., 1997.

[15] 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.

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

[17] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. In *International Conference on Supercomputing*, pages 444–453, 1999.

[18] S. Coleman and K. s. McKinley. Tile Selection Using Cache Organization and Data Layout. In *Proc. of Int. Conference Programming Language Design and Implementation*, pages 279–290, June 1995.

[19] C. Cook and D. Kim. Best sorting algorithms for nearly sorted lists. 23:620–624, 1980.

[20] 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.

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

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

[23] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.

[24] 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.

[25] J. Frens and D. Wise. Auto-blocking Matrix-Multiplication or Tracking BLAS3 Performance with Source Code. In *Proc. of the Intenational Symp. on Principles and Practice of Parallel programming (PPoPP)*, pages 206–216, June 1997.

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

[27] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proc. of the Intenational Symp. on Foundations of Computer Science (FOCS)*, October 1999.

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

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

[30] D. Hilbert. Über Stetige Abbildung einer Linie auf ein Flächenstrück. *Mathematische Annalen*, 38:459–60, 1891.

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

[32] 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.

[33] 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.

[34] 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.

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

[36] M. Lagoudakis, M. Littman, and R. Parr. Selecting the right algorithm. *Proceedings of the AAAI*, 2001.

[37] M. G. Lagoudakis and M. L. Littman. Algorithm selection using reinforcement learning. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 511–518, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[38] M. Lam, E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. of the Int. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, October 1991.

[39] 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.

[40] A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithms*, 1:4, 1996.

[41] 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.

[42] K. LeytonBrown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *Proceedings of IJCAI*, 2003.

[43] 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.

[44] X. Li, M. J. Garzarán, and D. Padua. Optimizing Sorting with Genetic Algorithms. In *In Proc. of the Int. Symp. on Code Generation and Optimization*, pages 99–110, March 2005.

[45] A. McKellar and E. Coffman. Organizing Matrices and Matrix Operations for Paged Memory Systems. *In Communications of the ACM*, 12(3):153–165, March 1969.

[46] N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante. Quantifying the Multi-Level Nature of Tiling Interactions. *Int. Journal of Parallel Programming*, 26(6):641–670, June 1998.

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

[48] 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.

[49] P. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting Loop Tiling with Data Alignment for Improved Cache Performance. *IEEE Trans. on Computers*, 48(2):142–149, February 1999.

[50] N. Park, B. Hong, and V. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):640–654, July 2003.

[51] G. Peano. Sur Une Curbe qui Remplit Toute une Aire Plaine. *Mathematische Annalen*, 36:157–160, 1890.

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

[53] K. A. Poonacha Kongetire and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. In *IEEE Micro*, pages 21–29, Mar/Apr 2005.

[54] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *The VLDB Journal*, pages 78–89, 1999.

[55] J. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[56] G. Rivera and C. Tseng. Data Transformations for Eliminating conflict Misses. In *Proc. of Int. Conference Programming Language Design and Implementation*, pages 38–49, June 1998.

[57] G. Rivera and C. Tseng. Locality Optimizations for Multi-Level Caches. In *Proc. of IEEE Supercomputing*, November 1999.

[58] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

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

[60] 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.

[61] P. K. P. Siddhartha Chatterjee, Alvin R. Lebeck and M. Thotterhodi. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 13:1105–1123, 2002.

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

[63] 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.

[64] J. Stockwood, R. Harr, T. Callahan, U. Kurkure, E. Darnell, and Y. Li. Hardware-software co-design of embedded reconfigurable architectures. *dac*, 00:507–512, 2000.

[65] O. Temam, E. Granston, and W. Jalby. To Copy or Not to Copy: A Compile–Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Proc. of the ACM/IEEE Supercomputing Conference*, November 1993.

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

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

[68] M. Wolfe. Iteration Space Tiling for Memory Hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

[69] 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.

[70] Q. Yi, V. Adve, and K. Kennedy. Transforming Loops To Recursion for Multi-Level Memory Hierarchies. In *Proc. of the Int. Conf. on Programming Language Design and Implementation (PLDI)*, pages 169–181, June 2000.

[71] 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.

[72] K. Yotov, X. Li, G. Ren, M. J. Garzarán, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate a High Performance Blas? *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, 23:358–386, February 2005.

Author's Biography

Xiaoming Li was born in Wuhan, China. Since his high school, he has been fascinated by computers. In 1994, he enrolled in Nanjing University and chose Computer science as his major. He graduated from Nanjing University in 1998, with a B.S. degree in Computer Science. He stayed in Nanjing for three more years and did research on compiling for parallelism. He received a Master's Degree, also in Computer Science, from Nanjing University in 2001. After that, he made a major decision to continue doctorate study in US. He joined the University of Illinois at Urbana-Champaign as a graduate student in 2001. His research has been focused on compilers and code generation/optimization. He completed his Ph.D. in July, 2006, with Prof. David Padua as his advisor. During his Ph.D. studies, he has been supported by Illinois Distinguished Fellowship and Illiac Fellowship from 2001 to 2004. He will join the faculty of the Department of Electrical and Computer Engineering at University of Delaware in August, 2006.